Spring IoC Container & Bean Lifecycle

BeanFactory/ApplicationContext, the bean lifecycle, @PostConstruct/@PreDestroy, bean scopes, @Lazy, and resolving circular dependencies.

Advanced 55 min

Last updated: August 11, 2026
Read online: https://www.learnforgex.com/en/topics/spring-ioc-container

---

Spring IoC Container & Bean Lifecycle

In the Dependency Injection & IoC lesson, "Manual Dependency Injection Without Spring (Composition Root)" showed us doing something entirely by hand -- knowing every concrete class and wiring objects together with `new`, in the right order. This lesson covers how a real Spring container automates that exact job. For the first time, we'll bring up and shut down a real `ApplicationContext`, and watch by hand when beans get created, in what order they start, when they shut down, and how many copies of each one exist (scope).

What Is the Spring IoC Container?

The Spring IoC container is the automated version of the composition root from the Dependency Injection & IoC lesson -- it reads classes/definitions, works out the dependency graph between them, builds objects in the right order, and manages their entire lifecycle (creation, initialization, being ready for use, shutdown):

```
// What we did by hand in "Manual Dependency Injection Without Spring
// (Composition Root)":
static OrderService buildOrderService() {
    NotificationSender sender = new EmailNotificationSender();
    return new OrderService(sender);
}

// The container doing the same job automatically:
ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
OrderService orderService = context.getBean(OrderService.class);
```

In the second version, you don't write the `new OrderService(...)` line -- the container does, by looking at the `@Bean` methods in `AppConfig` and working out which object needs which dependency on its own.

Why Does It Exist?

A hand-written setup method like `buildOrderService()` from "Manual Dependency Injection Without Spring (Composition Root)" is perfectly manageable for a handful of objects. But as an application grows -- hundreds of classes, complex dependencies between them, some needing exactly one shared copy for the whole app's lifetime while others need a fresh one every time, some needing to open a resource (like a database connection) at startup and release it at shutdown -- that hand-written setup code quickly turns into an error-prone layer that demands its own maintenance.

The container solves this by centrally managing three things: **resolving the dependency graph** (which object needs which, in what order they must be built), the **lifecycle** (when an object counts as "ready," what should run at shutdown), and **scope** (whether an object is a single instance for the whole app or a fresh copy per request). We'll cover all three, in order, in this lesson.

History

Spring's container started with Spring Framework 1.0 (2004) and the `BeanFactory` interface -- a minimal mechanism that just holds bean definitions and produces objects on demand. `ApplicationContext` followed shortly after: a richer interface that wraps (actually extends) `BeanFactory` and adds "enterprise" features on top, like event publishing, internationalization (message sources), and AOP-friendly proxy creation.

Bean definitions were originally written in XML files (read via `ClassPathXmlApplicationContext`); Spring 3.0 (2009) introduced Java-based configuration with `@Configuration` / `@Bean` (`AnnotationConfigApplicationContext`) -- the approach this project uses too. Spring Boot (2014), as we'll cover in depth in the next lesson (Spring Boot Auto-Configuration & Properties), all but eliminated the need to create an `ApplicationContext` by hand, by building and configuring it automatically behind `SpringApplication.run(...)`.

BeanFactory: The Root Interface

At the bottom of the container hierarchy sits `BeanFactory` -- the minimal interface that holds bean definitions and produces objects on demand, with no other "enterprise" features. One important trait: it's **lazy** -- registering a bean definition does not create the object:

```
import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;

interface NotificationSender {
    void send(String to, String message);
}

class EmailNotificationSender implements NotificationSender {
    EmailNotificationSender() {
        System.out.println("EmailNotificationSender constructed");
    }

    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

class BeanFactoryExample {
    public static void main(String[] args) {
        // The root container interface -- everything else (including
        // ApplicationContext) builds on top of this. Registering a definition
        // here does not create anything yet.
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();

        BeanDefinition definition = BeanDefinitionBuilder
                .genericBeanDefinition(EmailNotificationSender.class)
                .getBeanDefinition();
        factory.registerBeanDefinition("emailSender", definition);

        System.out.println("Bean definition registered -- nothing constructed yet.");
        // Bean definition registered -- nothing constructed yet.

        // BeanFactory is lazy by nature: EmailNotificationSender's constructor only
        // runs on this line, the first time the bean is actually asked for.
        NotificationSender sender = factory.getBean("emailSender", NotificationSender.
class);
        // EmailNotificationSender constructed

        sender.send("ayse@example.com", "Your order has been placed.");
        // [email to ayse@example.com] Your order has been placed.
    }
}
```

After the `registerBeanDefinition(...)` call, `EmailNotificationSender`'s constructor still hasn't run -- the output order in `main` shows this clearly. The object is only created the moment it's actually requested with `getBean(...)`. As we'll see in the next section, `ApplicationContext` changes this default.

ApplicationContext: The Layer Built on Top of BeanFactory

`ApplicationContext` extends `BeanFactory`, but with one important behavioral difference: **it creates singleton beans eagerly, the moment the context is built, instead of lazily**:

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

interface NotificationSender {
    void send(String to, String message);
}

class EmailNotificationSender implements NotificationSender {
    EmailNotificationSender() {
        System.out.println("EmailNotificationSender constructed");
    }

    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

@Configuration
class AppConfig {
    @Bean
    NotificationSender notificationSender() {
        return new EmailNotificationSender();
    }
}

class ApplicationContextExample {
    public static void main(String[] args) {
        // Unlike the raw BeanFactory, an ApplicationContext eagerly instantiates
        // every singleton bean the moment the context refreshes -- "EmailNotificationSender
        // constructed" prints right here, before any getBean(...) call at all.
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);
        // EmailNotificationSender constructed

        NotificationSender sender = context.getBean(NotificationSender.class);
        sender.send("ayse@example.com", "Your order has been placed.");
        // [email to ayse@example.com] Your order has been placed.

        context.close();
    }
}
```

Notice the exact reverse of `BeanFactoryExample`'s ordering: here the line `EmailNotificationSender constructed` is printed while `AnnotationConfigApplicationContext`'s constructor runs (while the context "refreshes"), before `getBean(...)` is ever called. That's why real Spring applications (Spring Boot included) almost always use `ApplicationContext` -- `BeanFactory` is valuable for understanding the container's conceptual foundation, but you rarely deal with it directly in everyday use.

What Is a Spring Bean?

A "bean" is any object created, configured, and managed throughout its lifecycle by the container -- it's no different from an ordinary Java class, only in how it gets created and managed:

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

interface ReceiptPrinter {
    void print(String item);
}

class ConsoleReceiptPrinter implements ReceiptPrinter {
    @Override
    public void print(String item) {
        System.out.println("[receipt] " + item);
    }
}

@Configuration
class AppConfig {
    @Bean
    ReceiptPrinter receiptPrinter() {
        return new ConsoleReceiptPrinter();
    }
}

class SpringBeanBasicsExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        // Every object the container manages -- not just the ones you wrote a
        // @Bean method for, but Spring's own internal infrastructure beans too --
        // shows up here.
        for (String name : context.getBeanDefinitionNames()) {
            System.out.println(name);
        }
        // receiptPrinter
        // ...plus several internal Spring infrastructure beans...

        System.out.println(context.containsBean("receiptPrinter")); // true

        ReceiptPrinter byType = context.getBean(ReceiptPrinter.class);
        ReceiptPrinter byName = (ReceiptPrinter) context.getBean("receiptPrinter");
        System.out.println(byType == byName); // true -- both point at the same singleton

        context.close();
    }
}
```

`getBeanDefinitionNames()`'s output shows not just `receiptPrinter`, which you defined with a `@Bean` method, but also beans Spring registers for its own internal infrastructure -- the container manages its own inner workings through the same mechanism. `byType == byName` comes out `true` because (as we saw in "ApplicationContext: The Layer Built on Top of BeanFactory") every bean is a single instance by default -- we dig deeper into this in "Bean Scope: Singleton (the Default)".

Defining Beans: Java Config with @Bean

`@Bean` methods inside a `@Configuration` class define how an object gets created -- when one `@Bean` method needs another `@Bean` as a parameter, Spring resolves it exactly the way it resolves a constructor parameter:

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

interface NotificationSender {
    void send(String to, String message);
}

class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

class OrderService {
    private final NotificationSender notificationSender;

    OrderService(NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

    void placeOrder(String customerContact, String item) {
        notificationSender.send(customerContact, "Your order for '" + item + "' has been
placed.");
    }
}

@Configuration
class AppConfig {
    @Bean
    NotificationSender notificationSender() {
        return new EmailNotificationSender();
    }

    @Bean
    OrderService orderService(NotificationSender notificationSender) {
        // Spring resolves this parameter exactly the way it resolves a constructor
        // parameter on a @Component -- by looking up a bean of the matching type.
        return new OrderService(notificationSender);
    }
}

class JavaConfigBeanExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        OrderService orderService = context.getBean(OrderService.class);
        orderService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.

        context.close();
    }
}
```

The `orderService(NotificationSender notificationSender)` method's parameter is identical to the `OrderService` constructor we wrote by hand in the Dependency Injection lesson, before touching Spring Boot at all -- the difference is that the container calls `new OrderService(notificationSender)` now, not you. In Component Scanning & Configuration, we'll compare this Java-config approach with `@Component` scanning, the second way to define beans.

Bean Naming and Multiple Beans

Once more than one implementation of the same interface is defined as a bean, `getBean(Type)` can no longer tell which one you mean -- that's where bean names (by default, the `@Bean` method's name) come in:

```
import org.springframework.beans.factory.NoUniqueBeanDefinitionException;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

interface NotificationSender {
    void send(String to, String message);
}

class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

class SmsNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[sms to " + to + "] " + message);
    }
}

@Configuration
class AppConfig {
    @Bean
    NotificationSender emailSender() {
        return new EmailNotificationSender();
    }

    @Bean
    NotificationSender smsSender() {
        return new SmsNotificationSender();
    }
}

class MultipleBeansExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);
```

```
        // Two NotificationSender beans exist now -- asking by type alone is ambiguous.
        try {
            context.getBean(NotificationSender.class);
        } catch (NoUniqueBeanDefinitionException e) {
            System.out.println("Ambiguous: " + e.getMessage());
            // Ambiguous: No qualifying bean of type 'NotificationSender' available:
            // expected single matching bean but found 2: emailSender,smsSender
        }

        // Asking by name (the @Bean method's name, by default) resolves it exactly.
        NotificationSender email = (NotificationSender) context.getBean("emailSender");
        NotificationSender sms = (NotificationSender) context.getBean("smsSender");
        email.send("ayse@example.com", "Hello via email");
        // [email to ayse@example.com] Hello via email
        sms.send("+90 555 000 00 00", "Hello via sms");
        // [sms to +90 555 000 00 00] Hello via sms

        context.close();
    }
}
```

With two `NotificationSender` beans defined, calling `context.getBean(NotificationSender.class)` throws `NoUniqueBeanDefinitionException` -- the container never tries to guess which one you want; it demands an explicit name. We'll cover how to resolve this ambiguity at the *injected constructor parameter* level with `@Qualifier` and `@Primary` in Component Scanning & Configuration -- the by-name `getBean(...)` call you see here is the exact mechanism those annotations sit on top of.

The Bean Lifecycle: How the Container Builds a Bean

A bean becoming "ready" isn't a single step -- the container follows a fixed sequence for every bean. Let's observe it with a special component that wraps every bean's initialization (`BeanPostProcessor`):

```
import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

// A BeanPostProcessor is infrastructure that wraps around EVERY bean's
// initialization -- it runs immediately before and after each bean's
// @PostConstruct step, for every bean the context manages.
class LoggingBeanPostProcessor implements BeanPostProcessor {
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException {
        System.out.println("[BeanPostProcessor] before init: " + beanName);
        return bean;
    }
```

```
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        System.out.println("[BeanPostProcessor] after init: " + beanName);
        return bean;
    }
}

class LifecycleLoggingBean {
    LifecycleLoggingBean() {
        System.out.println("1. Constructor");
    }

    @PostConstruct
    void init() {
        System.out.println("3. @PostConstruct");
    }

    @PreDestroy
    void cleanup() {
        System.out.println("5. @PreDestroy");
    }
}

@Configuration
class AppConfig {
    @Bean
    LoggingBeanPostProcessor loggingBeanPostProcessor() {
        return new LoggingBeanPostProcessor();
    }

    @Bean
    LifecycleLoggingBean lifecycleLoggingBean() {
        return new LifecycleLoggingBean();
    }
}

class BeanLifecyclePhasesExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);
        // 1. Constructor
        // [BeanPostProcessor] before init: lifecycleLoggingBean
        // 3. @PostConstruct
        // [BeanPostProcessor] after init: lifecycleLoggingBean

        System.out.println("4. Bean is fully ready and in use");
        // 4. Bean is fully ready and in use

        context.close();
        // 5. @PreDestroy
    }
}
```

The output order follows these exact steps: (1) the constructor runs, (2) dependencies are already set by the constructor, (3) `BeanPostProcessor.postProcessBeforeInitialization` runs for every bean, (4) the `@PostConstruct` method runs, (5) `postProcessAfterInitialization`

runs -- and the bean is now ready for use. At shutdown (`context.close()`), `@PreDestroy` runs in something close to the reverse order. The next two sections look more closely at step (4) and the shutdown step, from two different angles (annotation and interface).

@PostConstruct and @PreDestroy

If a bean needs a setup step that must run after its constructor finishes (once all its dependencies are set) -- `@PostConstruct` -- or needs to release a resource when the container shuts down -- `@PreDestroy` -- these two annotations exist for exactly that:

```
import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

// A resource that needs to be acquired once the bean is fully constructed
// (all its dependencies set) and released exactly once, when the container
// shuts down -- exactly what @PostConstruct/@PreDestroy are for.
class ConnectionPool {
    private boolean open;

    @PostConstruct
    void open() {
        open = true;
        System.out.println("ConnectionPool opened");
    }

    void borrowConnection() {
        if (!open) {
            throw new IllegalStateException("Pool is not open");
        }
        System.out.println("Connection borrowed");
    }

    @PreDestroy
    void close() {
        open = false;
        System.out.println("ConnectionPool closed");
    }
}

@Configuration
class AppConfig {
    @Bean
    ConnectionPool connectionPool() {
        return new ConnectionPool();
    }
}

class PostConstructPreDestroyExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);
        // ConnectionPool opened

        ConnectionPool pool = context.getBean(ConnectionPool.class);
        pool.borrowConnection();
        // Connection borrowed

        // Closing the context runs every managed bean's @PreDestroy method --
        // exactly why manually-created objects (via plain `new`) never get this
        // for free, only container-managed beans do.
        context.close();
        // ConnectionPool closed
    }
}
```

`ConnectionPool` itself doesn't implement any Spring interface -- it just marks two of its methods with annotations. When `context.close()` is called, every managed bean's `@PreDestroy` method runs automatically; that's a guarantee you never get for free with objects manually created via `new`, as in "Manual Dependency Injection Without Spring (Composition Root)" -- you'd have to track down who calls `close()` / `cleanup()` and when.

The InitializingBean and DisposableBean Interfaces

Before `@PostConstruct` / `@PreDestroy`, the only way to do the same job was implementing two Spring interfaces -- it still works, but this approach comes with a cost:

```
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

// The interface-based alternative to @PostConstruct/@PreDestroy -- it predates
// the annotations and still works, but ties this class's source code directly
// to Spring's own interfaces (compare with "@PostConstruct ve @PreDestroy",
// which needs no Spring-specific supertype at all).
class LegacyStyleConnectionPool implements InitializingBean, DisposableBean {
    @Override
    public void afterPropertiesSet() {
        System.out.println("ConnectionPool opened (InitializingBean)");
    }

    @Override
    public void destroy() {
        System.out.println("ConnectionPool closed (DisposableBean)");
    }
}

@Configuration
class AppConfig {
    @Bean
    LegacyStyleConnectionPool legacyStyleConnectionPool() {
        return new LegacyStyleConnectionPool();
    }
}

class InitializingDisposableBeanExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);
        // ConnectionPool opened (InitializingBean)

        context.close();
        // ConnectionPool closed (DisposableBean)
    }
}
```

The moment you write `LegacyStyleConnectionPool implements InitializingBean, DisposableBean`, that class becomes dependent on Spring -- it won't even compile without the container on the classpath. `@PostConstruct` / `@PreDestroy`, on the other hand, are just standard Java annotations (from the `jakarta.annotation` package) -- the class itself stays meaningful without ever importing Spring. That's why the annotation-based approach is almost always preferred today; you'll mostly see the interface-based one in older codebases.

Bean Scope: Singleton (the Default)

A bean's **scope** determines how many copies the container keeps. The default scope (the one you get even if you specify nothing) is singleton -- one instance per container:

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

class Counter {
    private int value;

    void increment() {
        value++;
    }

    int getValue() {
        return value;
    }
}

@Configuration
class AppConfig {
    @Bean
    // No scope annotation at all -- "singleton" is the default: the container
    // creates exactly one instance and hands out that same instance every time.
    Counter counter() {
        return new Counter();
    }
}

class SingletonScopeExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        Counter first = context.getBean(Counter.class);
        first.increment();
        first.increment();

        Counter second = context.getBean(Counter.class);
        second.increment();

        System.out.println(first == second); // true
        System.out.println(first.getValue()); // 3 -- both references share the same state

        context.close();
    }
}
```

`first` and `second` point at the same object (the `==` comparison is `true`) -- the change made by `first.increment()` is visible through `second` too, because both are the same `Counter`. This is why `byType == byName` came out `true` in "What Is a Spring Bean?".

> A singleton bean holding **mutable state** very easily leads to unexpected shared-state bugs -- `Counter` is kept deliberately simple here, but in a real application, multiple threads can access the same singleton bean at once (recall the race conditions from the Threads lesson), so singleton beans need to either be thread-safe or avoid mutable state entirely.

Bean Scope: Prototype

A bean marked `@Scope("prototype")` flips this default -- every `getBean(...)` call means the container creates a **new instance**:

```
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;

class Counter {
    private int value;

    void increment() {
        value++;
    }

    int getValue() {
        return value;
    }
}

@Configuration
class AppConfig {
    @Bean
    @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    // Every getBean() call now returns a BRAND NEW instance -- the container
    // still creates it and runs its lifecycle callbacks, but ownership (and
    // @PreDestroy) passes to the caller from that point on.
    Counter counter() {
        return new Counter();
    }
}

class PrototypeScopeExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);

        Counter first = context.getBean(Counter.class);
        first.increment();
        first.increment();

        Counter second = context.getBean(Counter.class);
        second.increment();

        System.out.println(first == second); // false -- two independent instances
        System.out.println(first.getValue());  // 2
        System.out.println(second.getValue()); // 1

        context.close();
    }
}
```

The exact same `Counter` class from "Bean Scope: Singleton (the Default)" behaves completely differently once the `@Scope` annotation is added -- `first` and `second` are now independent of each other; incrementing `first` doesn't affect `second` at all. This is preferred for state that shouldn't be shared across the whole application (for example, state that needs to be kept separate per user action).

Web Scopes: Request, Session, Application (A Quick Look)

Besides singleton and prototype, there are three more scopes that only make sense within a web application's context (like a Spring MVC app such as this one) -- these can't be tested with a standalone `AnnotationConfigApplicationContext`, because their existence depends on an HTTP request:

```
@Bean
@RequestScope   // one instance per single HTTP request
ShoppingCart requestScopedCart() { return new ShoppingCart(); }

@Bean
@SessionScope   // one instance per single user session
ShoppingCart sessionScopedCart() { return new ShoppingCart(); }

@Bean
@ApplicationScope   // one instance for the whole ServletContext (very close to singleton)
ShoppingCart applicationScopedCart() { return new ShoppingCart(); }
```

`@RequestScope` gives a different instance for each HTTP request (the old one is gone by the next request); `@SessionScope` keeps the same instance across different requests from the same user (a shopping cart, for example); `@ApplicationScope` is, in practice, very close to singleton, but is tied to the `ServletContext`. This project doesn't use any of these three scopes right now (`HomeController` / `TopicController` are stateless), but you'll run into them often in a real Spring MVC application.

Lazy Initialization: @Lazy

As we saw in "ApplicationContext: The Layer Built on Top of BeanFactory", `ApplicationContext` creates singleton beans eagerly by default. `@Lazy` reverses that default on a per-bean basis:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;

class EagerService {
    EagerService() {
        System.out.println("EagerService constructed");
    }
}

class LazyService {
    LazyService() {
        System.out.println("LazyService constructed");
    }
}

@Configuration
class AppConfig {
    @Bean
    EagerService eagerService() {
        return new EagerService();
    }

    @Bean
    @Lazy
    // This bean's constructor will NOT run when the context refreshes -- only
    // the first time something actually asks for it.
    LazyService lazyService() {
        return new LazyService();
    }
}

class LazyInitializationExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);
        // EagerService constructed

        System.out.println("Context refreshed -- LazyService not constructed yet.");
        // Context refreshed -- LazyService not constructed yet.

        context.getBean(LazyService.class);
        // LazyService constructed

        context.close();
    }
}
```

`EagerService`'s constructor runs immediately while the context is built, but `LazyService`'s -- marked `@Lazy` -- only runs the moment `getBean(LazyService.class)` is actually called -- just like the raw `BeanFactory`'s default behavior. This is used to shorten startup time for beans that are expensive to create but not guaranteed to be used on every run.

Circular Dependency: Why It Happens, How to Resolve It

`A` needs `B`; `B` also needs `A` -- if both use constructor injection, the container ends up in a deadlock it can't finish either one out of:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;

// ServiceA needs ServiceB, and ServiceB needs ServiceA -- neither can finish
// being constructed before the other exists. With plain constructor injection
// on both sides, Spring has no safe order to build them in and refuses outright
// (a BeanCurrentlyInCreationException wrapped in a BeanCreationException).
class ServiceA {
    private final ServiceB serviceB;

    ServiceA(ServiceB serviceB) {
        this.serviceB = serviceB;
    }
}

class ServiceB {
    private final ServiceA serviceA;

    ServiceB(@Lazy ServiceA serviceA) {
        // @Lazy here breaks the deadlock: instead of the real ServiceA, Spring
        // injects a proxy that only constructs the real ServiceA the first time
        // one of its methods is actually called -- by which point ServiceA's own
        // construction (which needed a finished ServiceB) has already completed.
        this.serviceA = serviceA;
    }
}

@Configuration
class AppConfig {
    @Bean
    ServiceA serviceA(ServiceB serviceB) {
        return new ServiceA(serviceB);
    }

    @Bean
    ServiceB serviceB(ServiceA serviceA) {
        return new ServiceB(serviceA);
    }
}

class CircularDependencyExample {
    public static void main(String[] args) {
        // Without @Lazy on one side of the cycle, this line would fail instead
        // of succeeding.
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);
        System.out.println("Context started successfully despite the circular dependency.");
        // Context started successfully despite the circular dependency.
        context.close();
    }
}
```

`@Lazy` is applied here to the `ServiceA` parameter in `ServiceB`'s constructor -- Spring injects a proxy standing in for the real `ServiceA`; that proxy only resolves the actual `ServiceA` bean

the first time one of its methods is really called. At that point, `ServiceB`'s construction can finish, which lets `ServiceA`'s construction finish too. Without `@Lazy`, this code would fail with a `BeanCurrentlyInCreationException` (wrapped in a `BeanCreationException`) -- the container detects the cycle and fails immediately instead of looping forever.

> `@Lazy` isn't the only fix -- switching one side from constructor injection to setter injection also breaks the cycle, because Spring can build a bean "half-ready" (constructor finished, but setters not yet called) and hand that half-ready reference to the other bean in the cycle. But for the reasons covered in "Why Is Constructor Injection Recommended?" in the Dependency Injection lesson, most teams prefer `@Lazy` instead, or (even better) redesigning the classes to remove the cycle entirely -- a circular dependency, much like an overly long constructor parameter list, is usually an early sign that two classes are too tightly coupled to each other.

ApplicationContext in Spring Boot (A Quick Look)

Every example in this lesson created the `ApplicationContext` by hand (`new AnnotationConfigApplicationContext(...)`). If you look at this project's own `LearningPlatformApplication` class, you won't see any of that:

```
@SpringBootApplication
public class LearningPlatformApplication {
    public static void main(String[] args) {
        SpringApplication.run(LearningPlatformApplication.class, args);
    }
}
```

`SpringApplication.run(...)` does exactly what we did by hand in this lesson, behind the scenes -- it creates an `ApplicationContext`, registers beans, refreshes the context -- and on top of that, starts an embedded web server (Tomcat) and keeps the application running. We'll cover how this container finds its beans (component scanning) and which beans Spring Boot defines "on your behalf" (auto-configuration) in Component Scanning & Configuration and Spring Boot Auto-Configuration & Properties, respectively.

Best Practices

- **Prefer `ApplicationContext` wherever you can, and avoid using `BeanFactory` directly** -- real applications (Spring Boot included) already work this way by default (see "ApplicationContext: The Layer Built on Top of BeanFactory").
- **Prefer `@PostConstruct` / `@PreDestroy` over `InitializingBean` / `DisposableBean`** -- it gives you the same guarantee without making your class dependent on Spring (see "The InitializingBean and DisposableBean Interfaces").

- **Keep singleton beans stateless, or make them thread-safe** -- since a single instance is shared across the whole application, mutable state easily turns into a concurrency bug (see "Bean Scope: Singleton (the Default)").
- **Reserve prototype scope for cases that genuinely need "fresh every time"** -- a prototype bean's `@PreDestroy` is never called by the container; cleanup responsibility passes to you.
- **Instead of "hiding" a circular dependency with `@Lazy`, consider removing it by redesigning where possible** -- it's usually a sign that two classes are too tightly coupled (see "Circular Dependency: Why It Happens, How to Resolve It").
- **Reserve `@Lazy` for beans that are genuinely expensive or rarely used** -- making everything lazy just means errors (like a missing configuration) surface much later, in an unrelated moment, instead of at application startup.

Common Mistakes

1. Assuming `BeanFactory` and `ApplicationContext` do the same thing. `ApplicationContext` creates singletons eagerly, `BeanFactory` is lazy -- this difference changes when a startup (or conversely, a never-called) error actually surfaces (see "BeanFactory: The Root Interface" and "ApplicationContext: The Layer Built on Top of BeanFactory").

2. Defining two beans of the same interface and expecting `getBean(Type.class)` to just "pick one." The container never guesses -- it throws `NoUniqueBeanDefinitionException` (see "Bean Naming and Multiple Beans").

3. Assuming `@PostConstruct` runs at the same time as the constructor. `@PostConstruct` runs **after** all dependencies are set -- that's exactly why work that relies on something not yet ready inside the constructor belongs in `@PostConstruct` instead (see "The Bean Lifecycle: How the Container Builds a Bean").

4. Expecting a prototype-scoped bean's `@PreDestroy` to run automatically when the container shuts down. The container has no way of knowing when a prototype bean is no longer needed -- cleanup responsibility passes to the code that received it (see "Bean Scope: Prototype").

5. "Silencing" a circular dependency error with `@Lazy` without rethinking the classes. While often a quick fix, it doesn't address the underlying design problem (two classes too tightly coupled to each other) (see "Circular Dependency: Why It Happens, How to Resolve It").

6. Trying to use web scopes (`@RequestScope` / `@SessionScope`) outside a web request context. These three only make sense inside a real HTTP request/session -- calling `getBean(...)` for one from a standalone `main` method fails (see "Web Scopes: Request, Session, Application (A Quick Look)").

Summary, Cheat Sheet, and Glossary

The Spring IoC container is the mechanism that automates the composition root we built by hand in the Dependency Injection lesson -- it defines beans, resolves the dependencies between them, manages their lifecycles, and decides how many copies to keep based on their scope. Key points:

- `BeanFactory`: the root interface, lazy (a bean definition ≠ a bean instance); `ApplicationContext`: built on top of `BeanFactory`, creates singletons eagerly, the layer used in real applications
- Bean lifecycle order: constructor → dependencies set → `BeanPostProcessor` (before) → `@PostConstruct` → `BeanPostProcessor` (after) → ready for use → (on `context.close()`) `@PreDestroy`
- `@PostConstruct` / `@PreDestroy` (annotation-based) is always preferred over `InitializingBean` / `DisposableBean` (interface-based, makes a class dependent on Spring)
- Scope: **singleton** (default, one instance per container), **prototype** (a new instance per `getBean()`), **request/session/application** (only meaningful in a web context)
- `@Lazy` reverses singletons' default eager creation on a per-bean basis; it can also be used to resolve a circular dependency
- A circular dependency creates a deadlock the container can't resolve under constructor injection (`BeanCurrentlyInCreationException`) -- switching to `@Lazy` or setter injection fixes it, but the root cause is usually a design problem

Quick reference:

```
// Creating an ApplicationContext
ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

@Configuration
class AppConfig {
    @Bean
    MyService myService() { return new MyService(); }

    @Bean
    @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    MyPrototype myPrototype() { return new MyPrototype(); }

    @Bean
    @Lazy
    ExpensiveService expensiveService() { return new ExpensiveService(); }
}

// Bean lifecycle
class ManagedBean {
    @PostConstruct
    void init() { /* dependencies ready, set up here */ }

    @PreDestroy
    void cleanup() { /* release resources when the container shuts down */ }
}

// Resolving a circular dependency
class ServiceB {
    ServiceB(@Lazy ServiceA serviceA) { /* a proxy is injected, deadlock avoided */ }
}

context.close(); // triggers @PreDestroy on every singleton bean
```

Glossary

BeanFactory — The root interface of Spring's container; holds bean definitions and produces objects on demand (lazily).

ApplicationContext — The container interface used in real applications; extends `BeanFactory`, creates singletons eagerly, and offers extra features like event publishing.

Bean — Any object created, configured, and managed throughout its lifecycle by the container.

Bean definition — The information given to the container describing how to create a bean (which class, which dependencies, which scope); a bean definition existing doesn't by itself mean the object has been created.

Bean lifecycle — The fixed sequence of steps, managed by the container, that a bean goes through from creation (constructor) to shutdown (`@PreDestroy`).

`@PostConstruct` / **`@PreDestroy`** — Standard Java (`jakarta.annotation`) annotations marking a bean's setup step (right after its dependencies are set) and cleanup step (when the container shuts down).

`BeanPostProcessor` — An extension point wrapping every bean's initialization, used by the container for its own infrastructure too.

Bean scope — The setting that determines how many copies of a bean the container keeps: singleton, prototype, request, session, application.

`@Lazy` — An annotation that makes a singleton bean get created only the first time it's actually requested, instead of when the context refreshes; also used to resolve circular dependencies.

Circular dependency — A situation where two (or more) beans need each other in a loop, so the container can't finish building either one first.

Appendix: Mini Project — A Container-Managed Reservation System

Let's combine this lesson's ideas: a **singleton** `ReservationRegistry` that prepares data at startup via `@PostConstruct` and prints a summary at shutdown via `@PreDestroy`, handing out **prototype** `ReservationTicket`s -- a fresh copy every time one is requested:

```
import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;

import java.util.ArrayList;
import java.util.List;

// Combines everything from this lesson into one small system: a SINGLETON
// registry (shared state, seeded on startup via @PostConstruct, summarized on
// shutdown via @PreDestroy) handing out PROTOTYPE tickets -- a fresh,
// independent instance every time one is requested.
class ReservationRegistry {
    private final List<String> confirmedTickets = new ArrayList<>();
    private int nextTicketNumber;

    @PostConstruct
    void seed() {
        nextTicketNumber = 1000;
        System.out.println("ReservationRegistry ready, starting from ticket #" +
nextTicketNumber);
    }

    synchronized int nextTicketNumber() {
        return nextTicketNumber++;
    }

    synchronized void confirm(String ticketId) {
        confirmedTickets.add(ticketId);
    }
```

```
    @PreDestroy
    void summarize() {
        System.out.println("Shutting down -- " + confirmedTickets.size() + " ticket(s)
confirmed: " + confirmedTickets);
    }
}

class ReservationTicket {
    private final int ticketNumber;
    private final ReservationRegistry registry;

    ReservationTicket(ReservationRegistry registry) {
        this.registry = registry;
        this.ticketNumber = registry.nextTicketNumber();
    }

    void confirm(String customerName) {
        String ticketId = "T-" + ticketNumber;
        registry.confirm(ticketId);
        System.out.println(ticketId + " confirmed for " + customerName);
    }
}

@Configuration
class AppConfig {
    @Bean
    ReservationRegistry reservationRegistry() {
        return new ReservationRegistry();
    }

    @Bean
    @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    ReservationTicket reservationTicket(ReservationRegistry reservationRegistry) {
        return new ReservationTicket(reservationRegistry);
    }
}
```

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

class ReservationSystemDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);
        // ReservationRegistry ready, starting from ticket #1000

        ReservationTicket first = context.getBean(ReservationTicket.class);
        first.confirm("Ayse");
        // T-1000 confirmed for Ayse

        // A brand new ticket instance every time -- prototype scope in action.
        ReservationTicket second = context.getBean(ReservationTicket.class);
        second.confirm("Mehmet");
        // T-1001 confirmed for Mehmet

        System.out.println(first == second); // false

        context.close();
        // Shutting down -- 2 ticket(s) confirmed: [T-1000, T-1001]
    }
}
```

`ReservationTicket`'s constructor depends on `ReservationRegistry` (a singleton) to get its own ticket number -- as we saw in "Bean Scope: Prototype", every call to `getBean(ReservationTicket.class)` returns a new `ReservationTicket`, but all of them use the same, shared `ReservationRegistry`. When `context.close()` is called, `ReservationRegistry.summarize()` (`@PreDestroy`) runs and summarizes every ticket confirmed up to that point.

> Notice that `ReservationRegistry.confirm(...)` and `nextTicketNumber()` are `synchronized` -- since this is a singleton bean (see the warning in "Bean Scope: Singleton (the Default)"), multiple HTTP requests in a real web application could access the same `ReservationRegistry` at the same time; without `synchronized`, this could result in a race condition (recall the Threads lesson) where two requests get the same ticket number.

Appendix: Mini Project — An Audited Order System (Circular Dependency)

The final mini project shows the idea from "Circular Dependency: Why It Happens, How to Resolve It" in a realistic scenario: `OrderService` needs `AuditLogger` to record every order; `AuditLogger` needs `OrderService` back, to write down how many orders have been placed so far in its log line -- not an artificial relationship, a genuinely two-way one:

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

import java.util.ArrayList;
import java.util.List;

// OrderService needs AuditLogger to record every order; AuditLogger needs
// OrderService back, to look up how many orders have been placed so far when
// it writes a log line -- a genuine two-way relationship, not just an
// accidental one (compare with "Circular Dependency").
class OrderService {
    private final AuditLogger auditLogger;
    private final List<String> orders = new ArrayList<>();

    OrderService(AuditLogger auditLogger) {
        this.auditLogger = auditLogger;
    }

    void placeOrder(String item) {
        orders.add(item);
        auditLogger.log("Order placed: " + item);
    }

    int orderCount() {
        return orders.size();
    }
}

class AuditLogger {
    private final OrderService orderService;

    AuditLogger(@Lazy OrderService orderService) {
        this.orderService = orderService;
    }

    void log(String message) {
        // orderService.orderCount() is safe to call here: by the time log(...)
        // actually runs, OrderService has long finished being constructed --
        // @Lazy only delayed resolving the REFERENCE, not this later method call.
        System.out.println("[audit #" + orderService.orderCount() + "] " + message);
    }
}

@Configuration
class AppConfig {
    @Bean
    OrderService orderService(AuditLogger auditLogger) {
        return new OrderService(auditLogger);
    }

    @Bean
    AuditLogger auditLogger(OrderService orderService) {
        return new AuditLogger(orderService);
    }
}
```

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

class AuditedOrderSystemDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AppConfig.class);
        System.out.println("Context started -- circular dependency resolved via @Lazy.");
        // Context started -- circular dependency resolved via @Lazy.

        OrderService orderService = context.getBean(OrderService.class);
        orderService.placeOrder("Java 21 Book");
        // [audit #1] Order placed: Java 21 Book
        orderService.placeOrder("Spring Boot Book");
        // [audit #2] Order placed: Spring Boot Book

        context.close();
    }
}
```

`@Lazy` is applied only to the `OrderService` parameter in `AuditLogger`'s constructor -- having `@Lazy` on both sides would be unnecessary, since breaking the cycle only takes one side "waiting." Notice that the call to `orderService.orderCount()` inside `AuditLogger.log(...)` is safe: this method runs much later, once the context is fully built and a real order is actually placed -- by that point, the real `OrderService` behind the proxy is long since ready.

> If you tried this scenario without `@Lazy` (using plain `OrderService` / `AuditLogger` parameters on both constructors), the context would fail to start at all, with a `BeanCurrentlyInCreationException` -- you've now seen the warning from "Circular Dependency: Why It Happens, How to Resolve It" play out in a concrete example.