Spring IoC Container ve Bean Yaşam Döngüsü

BeanFactory/ApplicationContext, bean lifecycle, @PostConstruct/@PreDestroy, bean scope'ları, @Lazy ve circular dependency çözümü.

İleri 55 dk

Son güncelleme: 11 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/spring-ioc-container

---

Spring IoC Container ve Bean Yaşam Döngüsü

Dependency Injection ve IoC dersinde "Spring Olmadan Elle Bağımlılık Enjeksiyonu (Composition Root)" bölümünde kendi elimizle yaptığımız işi -- somut sınıfları bilip `new` ile nesneleri doğru sırayla kurmak -- bu derste gerçek bir Spring container'ının nasıl otomatikleştirdiğini işliyoruz. İlk kez burada gerçek bir `ApplicationContext` ayağa kaldırıp kapatacağız; bean'lerin ne zaman yaratıldığını, hangi sırayla başlatıldığını, ne zaman kapatıldığını ve kaç kopyasının var olduğunu (scope) elle gözlemleyeceğiz.

Spring IoC Container Nedir?

Spring IoC container, Dependency Injection ve IoC dersindeki composition root'un otomatikleştirilmiş hâlidir -- sınıfları/tanımları okur, aralarındaki bağımlılık grafiğini çıkarır, nesneleri doğru sırayla kurar ve onların tüm yaşam döngüsünü (yaratma, başlatma, kullanılabilir olma, kapatma) yönetir:

```
// "Spring Olmadan Elle Bağımlılık Enjeksiyonu (Composition Root)" bölümünde
// elle yaptığımız iş:
static OrderService buildOrderService() {
    NotificationSender sender = new EmailNotificationSender();
    return new OrderService(sender);
}

// Container'ın aynı işi otomatik yapması:
ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
OrderService orderService = context.getBean(OrderService.class);
```

İkinci versiyonda `new OrderService(...)` satırını sen değil, container yazıyor -- `AppConfig` içindeki `@Bean` metotlarına bakarak hangi nesnenin hangi bağımlılığa ihtiyaç duyduğunu kendisi çıkarıyor.

Neden Var?

"Spring Olmadan Elle Bağımlılık Enjeksiyonu (Composition Root)" bölümündeki `buildOrderService()` gibi elle yazılmış bir kurulum metodu, birkaç nesne için gayet yönetilebilir. Ama bir uygulama büyüdükçe -- yüzlerce sınıf, aralarında karmaşık bağımlılıklar, bazılarının uygulama boyunca tek bir kopyası olması gerekirken bazılarının her seferinde yeniden yaratılması gerektiği, bazılarının başlarken bir kaynağı (veritabanı bağlantısı gibi) açıp kapanırken serbest bırakması gerektiği bir sistem -- bu elle yazılan kurulum kodu hızla kendi başına bakım gerektiren, hataya açık bir katmana dönüşür.

Container bunu üç şeyi merkezi olarak yöneterek çözer: **bağımlılık grafiğini çözme** (hangi nesne hangisine ihtiyaç duyuyor, hangi sırayla kurulmalı), **yaşam döngüsü** (bir nesne ne zaman "hazır" sayılır, kapanışta ne çalışmalı) ve **scope** (bir nesnenin uygulama boyunca tek mi yoksa her istekte yeni bir kopya mı olacağı). Bu derste üçünü de sırasıyla işleyeceğiz.

Tarihçe

Spring'in container'ı, Spring Framework 1.0 (2004) ile birlikte `BeanFactory` arayüzüyle başladı -- minimal, yalnızca bean tanımlarını tutup istendiğinde nesne üreten temel bir mekanizma. Kısa süre sonra `ApplicationContext` geldi: `BeanFactory`'yi kapsayan (aslında onu extend eden), üzerine olay yayınlama (event publishing), uluslararasılaştırma (mesaj kaynakları) ve AOP-dostu proxy oluşturma gibi "kurumsal" özellikler ekleyen daha zengin bir arayüz.

Başlangıçta bean tanımları XML dosyalarında yapılıyordu (`ClassPathXmlApplicationContext` ile okunurdu); Spring 3.0 (2009) `@Configuration` / `@Bean` ile Java tabanlı konfigürasyonu (`AnnotationConfigApplicationContext`) getirdi -- bugün bu projede de kullandığımız yöntem bu. Spring Boot (2014), bir sonraki konuda (Spring Boot Auto-Configuration & Properties) derinlemesine işleyeceğimiz gibi, bu container'ı `SpringApplication.run(...)` arkasında otomatik olarak kurup yapılandırarak, elle `ApplicationContext` yaratma ihtiyacını neredeyse tamamen ortadan kaldırdı.

BeanFactory: Kök Arayüz

Container hiyerarşisinin en altında `BeanFactory` var -- bean tanımlarını tutan, istenildiğinde nesne üreten, ama başka hiçbir "kurumsal" özelliği olmayan minimal arayüz. Önemli bir özelliği: **tembeldir (lazy)** -- bir bean tanımı kaydetmek, o nesneyi yaratmaz:

```
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.
    }
}
```

`registerBeanDefinition(...)` çağrısından sonra `EmailNotificationSender`'ın constructor'ı hâlâ çalışmamış -- `main`'deki çıktı sırası bunu net gösteriyor. Nesne, yalnızca `getBean(...)` ile gerçekten istendiği an yaratılıyor. Bir sonraki bölümde göreceğimiz gibi, `ApplicationContext` bu varsayılanı değiştiriyor.

ApplicationContext: BeanFactory'nin Üzerine İnşa Edilen Katman

`ApplicationContext`, `BeanFactory`'yi genişletir ama davranışça önemli bir farkı vardır: **singleton bean'leri tembel değil, context oluşturulur oluşturulmaz hemen (eager) yaratır**:

```
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();
    }
}
```

`BeanFactoryExample`'daki sıranın tam tersine dikkat et: burada `EmailNotificationSender constructed` satırı, `getBean(...)` çağrılmadan, `AnnotationConfigApplicationContext`'in

constructor'ı çalışırken (context "refresh" edilirken) yazdırılıyor. Bu yüzden gerçek Spring uygulamalarında (Spring Boot dahil) neredeyse her zaman `ApplicationContext` kullanılır -- `BeanFactory`, container'ın kavramsal temelini anlamak için değerli, ama günlük kullanımda doğrudan karşına nadiren çıkar.

Spring Bean Nedir?

"Bean", container tarafından yaratılan, yapılandırılan ve yaşam döngüsü boyunca yönetilen herhangi bir nesnedir -- Java'daki sıradan bir sınıftan hiçbir farkı yok, farkı yaratılış ve yönetilme şeklinde:

```
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()` çıktısında yalnızca senin `@Bean` metotlarınla tanımladığın `receiptPrinter` değil, Spring'in kendi altyapısı için kaydettiği bean'ler de görünür -- container, kendi iç işleyişini de aynı mekanizmayla yönetir. `byType == byName` karşılaştırması `true` çıkıyor çünkü ("ApplicationContext: BeanFactory'nin Üzerine İnşa Edilen Katman" bölümünde gördüğümüz gibi) varsayılan olarak her bean tek bir örnek -- bunu "Bean Scope: Singleton (Varsayılan)" bölümünde derinleştireceğiz.

Bean Tanımlama: @Bean ile Java Config

`@Configuration` sınıfları içindeki `@Bean` metotları, hangi nesnenin nasıl yaratılacağını tanımlar -- bir `@Bean` metodu başka bir `@Bean`'e parametre olarak ihtiyaç duyduğunda, Spring bunu tıpkı bir constructor parametresi gibi çözer:

```
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();
    }
}
```

`orderService(NotificationSender notificationSender)` metodunun parametresi, Spring Boot'a değinmeden önce Dependency Injection dersinde elle yazdığımız `OrderService` constructor'ının birebir aynısı -- fark, `new OrderService(notificationSender)` satırını artık senin değil, container'ın çağırmasında. Component Scanning & Configuration dersinde, bu Java-config yaklaşımını `@Component` taramasıyla (bean tanımlamanın ikinci yolu) karşılaştıracağız.

Bean Adlandırma ve Birden Fazla Bean

Aynı arayüzün birden fazla implementasyonu bean olarak tanımlandığında, `getBean(Type)` artık hangisini kastettiğini bilemez -- bunun için bean'lerin isimleri (varsayılan olarak `@Bean` metodunun adı) devreye girer:

```
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();
    }
}
```

İki `NotificationSender` bean'i varken `context.getBean(NotificationSender.class)` çağırmak `NoUniqueBeanDefinitionException` fırlatıyor -- container hangisini istediğini tahmin etmeye çalışmıyor, açıkça bir isim ister. Bu belirsizliği `@Qualifier` ve `@Primary` ile *enjekte edilen bir constructor parametresi* seviyesinde nasıl çözeceğimizi Component Scanning & Configuration dersinde işleyeceğiz -- burada gördüğün isimle `getBean(...)` çağrısı, o annotation'ların altında yatan aynı mekanizmadır.

Bean Lifecycle: Container'ın Bir Bean'i İnşa Etme Adımları

Bir bean'in "hazır" hâle gelmesi tek bir adım değil -- container, her bean için sabit bir sıra izler. Bunu, her bean'in başlatılmasını saran özel bir bileşenle (`BeanPostProcessor`) gözlemleyelim:

```
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
    }
}
```

Çıktı sırası tam olarak şu adımları izliyor: (1) constructor çalışır, (2) bağımlılıklar zaten constructor'da set edilmiş olur, (3) `BeanPostProcessor.postProcessBeforeInitialization` her bean için çalışır, (4) `@PostConstruct` metodu çalışır, (5) `postProcessAfterInitialization`

çalışır -- ve bean artık kullanıma hazırdır. Kapanışta (`context.close()`) bu sıranın tersine yakın bir şekilde `@PreDestroy` çalışır. Sıradaki iki bölüm, adım (4) ve kapanıştaki adıma iki farklı açıdan (annotation ve interface) daha yakından bakıyor.

@PostConstruct ve @PreDestroy

Bir bean'in, constructor'ı bittikten (tüm bağımlılıkları set edildikten) sonra çalışması gereken bir kurulum adımı varsa (`@PostConstruct`), ya da container kapanırken serbest bırakması gereken bir kaynağı varsa (`@PreDestroy`), bu iki annotation tam olarak bunun için var:

```
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`'un kendisi hiçbir Spring arayüzü implement etmiyor -- yalnızca iki metodunu annotation'la işaretliyor. `context.close()` çağrıldığında her yönetilen bean'in `@PreDestroy` metodu otomatik çalışır; bu, "Spring Olmadan Elle Bağımlılık Enjeksiyonu (Composition Root)" bölümünde elle `new` ile yaratılmış nesnelerde asla bedava gelmeyen bir garanti -- kimin ne zaman `close()` / `cleanup()` çağıracağını sen takip etmek zorunda kalırdın.

InitializingBean ve DisposableBean Arayüzleri

`@PostConstruct` / `@PreDestroy`'dan önce, aynı işi yapmanın tek yolu iki Spring arayüzünü implement etmekti -- hâlâ çalışır, ama bu yaklaşımın bir bedeli var:

```
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)
    }
}
```

`LegacyStyleConnectionPool implements InitializingBean, DisposableBean` yazdığın an, bu sınıf artık Spring'e bağımlı hâle geliyor -- container olmadan derlenemez bile. `@PostConstruct` / `@PreDestroy` ise yalnızca standart Java annotation'ları (`jakarta.annotation` paketinden), sınıfın kendisi Spring'i hiç import etmeden de anlamlı kalır. Bu yüzden günümüzde neredeyse her zaman annotation tabanlı yaklaşım tercih edilir; arayüz tabanlı yaklaşımı büyük ölçüde eski kod tabanlarında görürsün.

Bean Scope: Singleton (Varsayılan)

Bir bean'in **scope**'u, container'ın kaç kopyasını tutacağını belirler. Varsayılan (hiçbir şey belirtmesen bile geçerli olan) scope singleton'dır -- container başına tek bir örnek:

```
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` ve `second`, aynı nesneyi işaret ediyor (`==` karşılaştırması `true`) -- `first.increment()` ile yapılan değişiklik, `second` üzerinden de görünüyor, çünkü ikisi de aynı `Counter`. Bu, "Spring Bean Nedir?" bölümünde `byType == byName` karşılaştırmasında gördüğümüz davranışın nedeni.

> Singleton bir bean'in **mutable (değişebilir) durum** tutması, çok kolayca beklenmedik paylaşılan durum hatalarına yol açar -- `Counter` burada kasıtlı olarak basit tutuldu, ama gerçek bir uygulamada birden fazla thread aynı singleton bean'e aynı anda erişebileceği için (Threads dersindeki race condition'ları hatırla), singleton bean'lerin ya thread-safe olması ya da mutable durumdan tamamen kaçınması gerekir.

Bean Scope: Prototype

`@Scope("prototype")` ile işaretlenen bir bean'de bu varsayılan tersine döner -- her `getBean(...)` çağrısı, container'ın **yeni bir örnek** yarattığı anlamına gelir:

```
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();
    }
}
```

"Bean Scope: Singleton (Varsayılan)" bölümündeki örnekle birebir aynı `Counter` sınıfı, yalnızca `@Scope` annotation'ı eklenince tamamen farklı davranıyor -- `first` ve `second` artık birbirinden bağımsız, `first`'ü artırmak `second`'ı hiç etkilemiyor. Bu, uygulama boyunca paylaşılmaması gereken (örneğin her kullanıcı işlemi için ayrı tutulması gereken) durum için tercih edilir.

Web Scope'ları: Request, Session, Application (Kısa Bakış)

Singleton ve prototype dışında, yalnızca bir web uygulaması bağlamında (bu proje gibi bir Spring MVC uygulamasında) anlamlı olan üç scope daha vardır -- bunlar standalone bir `AnnotationConfigApplicationContext` ile test edilemez, çünkü varlıkları bir HTTP isteğine bağlıdır:

```
@Bean
@RequestScope   // tek bir HTTP isteği boyunca tek örnek
ShoppingCart requestScopedCart() { return new ShoppingCart(); }

@Bean
@SessionScope   // tek bir kullanıcı oturumu boyunca tek örnek
ShoppingCart sessionScopedCart() { return new ShoppingCart(); }

@Bean
@ApplicationScope   // tüm ServletContext boyunca tek örnek (singleton'a çok yakın)
ShoppingCart applicationScopedCart() { return new ShoppingCart(); }
```

`@RequestScope`, her HTTP isteği için farklı bir örnek verir (bir sonraki istekte eski örnek yok olur); `@SessionScope`, aynı kullanıcının farklı istekleri arasında aynı örneği korur (örneğin bir alışveriş sepeti); `@ApplicationScope` ise pratikte singleton'a çok benzer, ama `ServletContext` 'e bağlıdır. Bu proje şu an bu üç scope'u hiç kullanmıyor (`HomeController` / `TopicController` stateless çalışıyor), ama gerçek bir Spring MVC uygulamasında sıkça karşına çıkarlar.

Lazy Initialization: @Lazy

"ApplicationContext: BeanFactory'nin Üzerine İnşa Edilen Katman" bölümünde gördüğümüz gibi, `ApplicationContext` singleton bean'leri varsayılan olarak hemen (eager) yaratır. `@Lazy` bu varsayılanı tek tek bean bazında geri alır:

```
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`'in constructor'ı context oluşturulurken hemen çalışıyor, ama `@Lazy` işaretli `LazyService`'inki, yalnızca `getBean(LazyService.class)` gerçekten çağrıldığında çalışıyor -- tıpkı ham `BeanFactory`'nin varsayılan davranışı gibi. Bu, yaratılması pahalı ama her çalıştırmada mutlaka kullanılmayan bean'ler için başlangıç süresini kısaltmak amacıyla kullanılır.

Circular Dependency: Neden Olur, Nasıl Çözülür

`A`, `B`'ye ihtiyaç duyuyor; `B` de `A`'ya ihtiyaç duyuyor -- ikisi de constructor injection kullanıyorsa, container'ın hiçbirini önce bitiremeyeceği bir çıkmaz oluşur:

```
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` burada `ServiceB`'nin constructor'ındaki `ServiceA` parametresine uygulanıyor -- Spring, gerçek `ServiceA` yerine onun yerine geçen bir proxy enjekte ediyor; bu proxy,

yalnızca ilk gerçek metot çağrısında asıl `ServiceA` bean'ini çözüyor. Bu noktada `ServiceB` 'nin kurulumu tamamlanabiliyor, dolayısıyla `ServiceA` da kurulumunu bitirebiliyor. `@Lazy` olmasaydı bu kod `BeanCurrentlyInCreationException` ile (bir `BeanCreationException` 'a sarılmış olarak) patlardı -- container sonsuz döngüye girmek yerine çemberi tespit edip hemen hata verir.

> `@Lazy` tek çözüm değil -- constructor injection yerine bir tarafta setter injection'a geçmek de çemberi kırar, çünkü Spring bir bean'i "yarı hazır" (constructor tamamlanmış ama setter'lar henüz çağrılmamış) hâlde önce oluşturup, döngüdeki diğer bean'i bu yarı hazır referansla besleyebilir. Ama Dependency Injection dersindeki "Neden Constructor Injection Öneriliyor?" bölümünde işlediğimiz gerekçelerle, çoğu ekip bunun yerine `@Lazy` 'yi ya da (daha da iyisi) sınıfları yeniden tasarlayıp çemberi tamamen ortadan kaldırmayı tercih eder -- bir circular dependency genelde, tıpkı çok parametreli bir constructor gibi, iki sınıfın birbirine fazla bağımlı tasarlandığının erken bir işaretidir.

Spring Boot'ta ApplicationContext (Kısa Bakış)

Bu dersteki her örnekte `ApplicationContext` 'i elle yarattık (`new AnnotationConfigApplicationContext(...)`). Bu projenin kendi `LearningPlatformApplication` sınıfına bakarsan, bunu hiç görmezsin:

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

`SpringApplication.run(...)` , arka planda tam olarak bu derste elle yaptığımız işi yapıyor -- bir `ApplicationContext` yaratıyor, bean'leri kaydediyor, context'i "refresh" ediyor -- üstüne bir de embedded web sunucusu (Tomcat) başlatıp uygulamayı ayakta tutuyor. Bu container'ın bean'leri nereden bulduğunu (component scanning) ve Spring Boot'un hangi bean'leri "senin yerine" otomatik tanımladığını (auto-configuration) sırasıyla Component Scanning & Configuration ve Spring Boot Auto-Configuration & Properties derslerinde işleyeceğiz.

Best Practices

- **Elinden geldiğince `ApplicationContext` kullan, `BeanFactory` 'yi doğrudan kullanmaktan kaçın** -- gerçek uygulamalarda (Spring Boot dahil) zaten hep bu şekilde çalışırsın (bkz. "ApplicationContext: BeanFactory'nin Üzerine İnşa Edilen Katman").

- `@PostConstruct` / `@PreDestroy`**'ı** `InitializingBean` / `DisposableBean`**'e tercih et** -- sınıfını Spring'e bağımlı kılmadan aynı garantiyi verir (bkz. "InitializingBean ve DisposableBean Arayüzleri").
- **Singleton bean'leri stateless tut ya da thread-safe yap** -- tüm uygulama boyunca tek bir örnek paylaşıldığı için, mutable durum kolayca eşzamanlılık hatasına dönüşür (bkz. "Bean Scope: Singleton (Varsayılan)").
- **Prototype scope'u yalnızca gerçekten "her seferinde yeni" gereken durumlar için kullan** -- prototype bean'lerin `@PreDestroy`'u container tarafından çağrılmaz, temizlik sorumluluğu sana geçer.
- **Bir circular dependency'yi** `@Lazy` **ile "gizlemek" yerine, mümkünse tasarımı değiştirerek ortadan kaldırmayı düşün** -- genelde iki sınıfın birbirine fazla bağımlı olduğunun işaretidir (bkz. "Circular Dependency: Neden Olur, Nasıl Çözülür").
- `@Lazy`**'yi yalnızca gerçekten pahalı ya da nadiren kullanılan bean'ler için kullan** -- her şeyi lazy yapmak, hataların (örn. eksik bir konfigürasyon) uygulama başlangıcında değil, çok daha sonra, ilgisiz bir anda ortaya çıkmasına yol açar.

Yaygın Hatalar

1. `BeanFactory` **ile** `ApplicationContext`**'in aynı şeyi yaptığını sanmak.** `ApplicationContext` singleton'ları eager yaratır, `BeanFactory` lazy'dir -- bu fark, başlangıçtaki (ya da tam tersi, hiç çağrılmayan) bir hatanın ne zaman ortaya çıkacağını değiştirir (bkz. "BeanFactory: Kök Arayüz" ve "ApplicationContext: BeanFactory'nin Üzerine İnşa Edilen Katman").

2. Aynı arayüzden iki bean tanımlayıp `getBean(Type.class)`**'in "birini" seçeceğini ummak.** Container asla tahmin etmez -- `NoUniqueBeanDefinitionException` fırlatır (bkz. "Bean Adlandırma ve Birden Fazla Bean").

3. `@PostConstruct` **metodunun constructor'la aynı anda çalıştığını sanmak.** `@PostConstruct`, tüm bağımlılıklar set edildikten **sonra** çalışır -- constructor içinde henüz hazır olmayan bir şeye güvenip iş yapmak yerine, o işi `@PostConstruct`'a taşımak bu yüzden vardır (bkz. "Bean Lifecycle: Container'ın Bir Bean'i İnşa Etme Adımları").

4. Prototype scope'lu bir bean'in `@PreDestroy`**'unun container kapanırken otomatik çalışacağını beklemek.** Container, prototype bean'in ne zaman artık kullanılmadığını bilemez -- temizlik sorumluluğu bean'i alan koda geçer (bkz. "Bean Scope: Prototype").

5. Circular dependency hatasını, sınıfları yeniden düşünmeden doğrudan `@Lazy` **ile "susturmak".** Çoğu zaman hızlı bir düzeltme olsa da, altında yatan tasarım sorununu (iki sınıfın birbirine fazla bağımlı olması) çözmez (bkz. "Circular Dependency: Neden Olur, Nasıl Çözülür").

6. Web scope'larını (`@RequestScope` / `@SessionScope`**) bir web isteği bağlamı olmadan kullanmaya çalışmak.** Bu üçü yalnızca gerçek bir HTTP isteği/oturumu içindeyken anlamlıdır -- standalone bir `main` metodunda `getBean(...)` ile çağırmak hataya yol açar (bkz. "Web Scope'ları: Request, Session, Application (Kısa Bakış)").

Özet, Cheat Sheet ve Terimler Sözlüğü

Spring IoC container, Dependency Injection dersinde elle yaptığımız composition root'u otomatikleştiren mekanizmadır -- bean'leri tanımlar, aralarındaki bağımlılığı çözer, yaşam döngülerini yönetir ve scope'larına göre kaç kopya tutacağına karar verir. Öne çıkan noktalar:

- `BeanFactory`: kök arayüz, lazy (bean tanımı ≠ bean nesnesi); `ApplicationContext`: `BeanFactory`'nin üzerine kurulu, singleton'ları eager yaratan, gerçek uygulamalarda kullanılan katman
- Bean yaşam döngüsü sırası: constructor → bağımlılıklar set edilir → `BeanPostProcessor` (before) → `@PostConstruct` → `BeanPostProcessor` (after) → kullanıma hazır → (`context.close()`'da) `@PreDestroy`
- `@PostConstruct` / `@PreDestroy` (annotation tabanlı) her zaman `InitializingBean` / `DisposableBean`'e (arayüz tabanlı, Spring'e bağımlı kılar) tercih edilir
- Scope: **singleton** (varsayılan, container başına tek örnek), **prototype** (her `getBean()` yeni örnek), **request/session/application** (yalnızca web bağlamında anlamlı)
- `@Lazy`, singleton'ların varsayılan eager yaratılışını tek tek bean bazında geri alır; circular dependency'yi çözmek için de kullanılabilir
- Circular dependency, constructor injection'da container'ın çözemediği bir çıkmaz yaratır (`BeanCurrentlyInCreationException`) -- `@Lazy` ya da setter injection'a geçmek çözer, ama kök neden genelde bir tasarım sorunudur

Hızlı referans:

```
// ApplicationContext yaratmak
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() { /* bağımlılıklar hazır, kurulum burada */ }

    @PreDestroy
    void cleanup() { /* container kapanırken kaynak serbest bırak */ }
}

// Circular dependency çözümü
class ServiceB {
    ServiceB(@Lazy ServiceA serviceA) { /* proxy enjekte edilir, deadlock önlenir */ }
}

context.close(); // tüm singleton bean'lerin @PreDestroy'unu tetikler
```

Terimler Sözlüğü

BeanFactory — Spring container'ının kök arayüzü; bean tanımlarını tutar, istenildiğinde (lazy) nesne üretir.

ApplicationContext — `BeanFactory`'yi genişleten, singleton'ları eager yaratan ve olay yayınlama gibi ek özellikler sunan, gerçek uygulamalarda kullanılan container arayüzü.

Bean — Container tarafından yaratılan, yapılandırılan ve yaşam döngüsü boyunca yönetilen herhangi bir nesne.

Bean definition (bean tanımı) — Bir bean'in nasıl yaratılacağına dair container'a verilen bilgi (hangi sınıf, hangi bağımlılıklar, hangi scope); bean tanımının kendisi henüz o nesnenin yaratıldığı anlamına gelmez.

Bean lifecycle (bean yaşam döngüsü) — Bir bean'in yaratılmasından (constructor) kapatılmasına (`@PreDestroy`) kadar geçtiği, container tarafından yönetilen sabit adım sırası.

`@PostConstruct` / **`@PreDestroy`** — Bir bean'in kurulum (bağımlılıklar set edildikten hemen sonra) ve temizlik (container kapanırken) adımlarını işaretleyen, standart Java (`jakarta.annotation`) annotation'ları.

`BeanPostProcessor` — Her bean'in başlatılmasını (initialization) saran, container'ın kendi altyapısı için kullandığı bir uzantı noktası.

Bean scope — Bir bean'in container tarafından kaç kopyasının tutulacağını belirleyen ayar: singleton, prototype, request, session, application.

`@Lazy` — Bir singleton bean'in, container refresh edilirken değil, yalnızca ilk gerçekten istendiğinde yaratılmasını sağlayan annotation; circular dependency çözümünde de kullanılır.

Circular dependency — İki (ya da daha fazla) bean'in birbirine dönüşümlü olarak ihtiyaç duyması yüzünden container'ın hiçbirini önce bitiremediği durum.

Ek: Mini Proje — Container Yönetimli Bir Rezervasyon Sistemi

Bu dersteki fikirleri birleştirelim: `@PostConstruct` ile açılışta veri hazırlayan, `@PreDestroy` ile kapanışta özet yazdıran **singleton** bir `ReservationRegistry`, her istemde yeni bir kopyası verilen **prototype** `ReservationTicket`'lar üretiyor:

```
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`'ın constructor'ı, kendi bilet numarasını almak için `ReservationRegistry`'ye (bir singleton'a) bağımlı -- "Bean Scope: Prototype" bölümünde gördüğümüz gibi her `getBean(ReservationTicket.class)` çağrısı yeni bir `ReservationTicket` döndürüyor, ama hepsi aynı, paylaşılan `ReservationRegistry`'yi kullanıyor. `context.close()` çağrıldığında `ReservationRegistry.summarize()` (`@PreDestroy`) çalışıp o ana kadar onaylanan tüm biletleri özetliyor.

> `ReservationRegistry.confirm(...)` ve `nextTicketNumber()` metotlarının `synchronized` olmasına dikkat et -- bu bir singleton bean olduğu için (bkz. "Bean Scope: Singleton (Varsayılan)" bölümündeki uyarı), gerçek bir web uygulamasında birden fazla HTTP isteği aynı `ReservationRegistry`'ye eşzamanlı erişebilir; `synchronized` olmasaydı iki isteğin aynı bilet numarasını almasıyla sonuçlanabilecek bir race condition (Threads dersini hatırla) oluşurdu.

Ek: Mini Proje — Denetimli Sipariş Sistemi (Circular Dependency)

Son mini proje, "Circular Dependency: Neden Olur, Nasıl Çözülür" bölümündeki fikri gerçekçi bir senaryoda gösteriyor: `OrderService`, her siparişi kaydetmek için `AuditLogger`'a ihtiyaç duyuyor; `AuditLogger` da, log satırına kaçıncı sipariş olduğunu yazabilmek için `OrderService`'e geri ihtiyaç duyuyor -- yapay değil, gerçek bir çift yönlü ilişki:

```
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`, yalnızca `AuditLogger`'ın constructor'ındaki `OrderService` parametresine uygulanıyor -- iki taraf da `@Lazy` olsaydı bu gereksiz olurdu, çünkü çemberi kırmak için tek bir tarafın "beklemesi" yeterli. `AuditLogger.log(...)` içinde `orderService.orderCount()` çağrısının güvenle çalıştığına dikkat et: bu metot, context tamamen kurulduktan çok sonra, gerçek bir sipariş verildiğinde çalışıyor -- o noktada proxy'nin arkasındaki gerçek `OrderService` çoktan hazır.

> Bu senaryoyu `@Lazy` olmadan denesen (her iki constructor'da da düz `OrderService` / `AuditLogger` parametreleri kullansan), context hiç ayağa kalkmadan `BeanCurrentlyInCreationException` ile başarısız olurdu -- "Circular Dependency: Neden Olur, Nasıl Çözülür" bölümündeki uyarıyı burada somut bir örnekte görmüş oldun.