SPRING NEDİR?

Java’yı daha basit yapar.

Java’yı daha üretken kılar.

Java’yı reactive yapar.

Java’yı cloud-ready yapar.

Java’yı daha hızlı, kolay ve güvenli yapıyor.

Spring -> Spring Framework

Enterprice edition: java programlama dilini kullanarak kurumsal düzeyde uygulamalar oluşturmak için bir dizi özellik ve kılavuzdur. Büyük ölçekli, çok katmanlı, ölçeklenebilir ve güvenli ağ uygulamaları geliştirmek ve dağıtmak için kapsamlı bir platform sağlar.

Java Enterprice edition = Java EE = Jakarta EE

Enterprice JavaBeans (EJBs) main (ana) component’i (bileşeni) Java EE’nin.

EJBs server side componentler distributed component

Java EE’yi EJB’ler olmadan nasıl kullanırız?

Web tier is provided by a MVC Framework -> Web katmanı, bir MVC yapısını takip eden framework tarafından yönetiliyor.

Business objects will be POJO (Plain Old Java Objects) -> İş mantığını temsil eden nesneler (business objects), sıradan java sınıfları (POJO) olacaktır.

Entity beans are not an option, so data access will be through an O/R mapping solution or JDBC -> Entity Bean kullanımı bir seçenek değildir, bu nedenle veri erişimi O/R (Object - Relational) eşleme çözümü veya JDBC (Java Database Connectivity) aracılığıyla yapılacaktır.

Spring Java EE’nin sağladığını daha basit bir programlama modeliyle ve daha ucuz sağlamak için ortaya çıkmış bir girişim.

Spring iki önemli konu üzerinde duruyor -> POJO ve lightweight container

//

**Spring'de "Container" Ne Demek?**

Spring dünyasında **container**, uygulamadaki nesneleri (bean’leri):

* **oluşturan** (instantiation),
* **yöneten** (lifecycle management),
* **bağımlılıklarını çözen** (dependency injection),
* ve gerektiğinde **bunları sağlayan** (injection / retrieval)

**yönetim sistemidir.**

//

Usage Scenarios

Spring enhances Java -> Spring Java’yı geliştirir.

Spring complements Java -> Spring Java’yı tamamlar.

Spring competes Java -> Spring Java’yla rekabet eder(daha iyi çözümlerle rekabet)

(Bazen Spring, Java'da zaten standart olarak var olan şeylere alternatif çözümler sunarak Java ile rekabet eder.)

DEPENDENCY INJECTION NEDİR?

Bağımlılıkları yönetmek. Bağımlılık zerk etmek.

Coupling nesneler arasında birbirleriyle alakalı bağımlılığın en genel ismi.(daha bilimsel)

Dependency daha özel bir Coupling.

İki nesne arasındaki bir bağımlılık öyle ki birisindeki bir değişim muhtemelen diğerinde de değişime bizi zorlar.

Dependency -> Direct Coupling

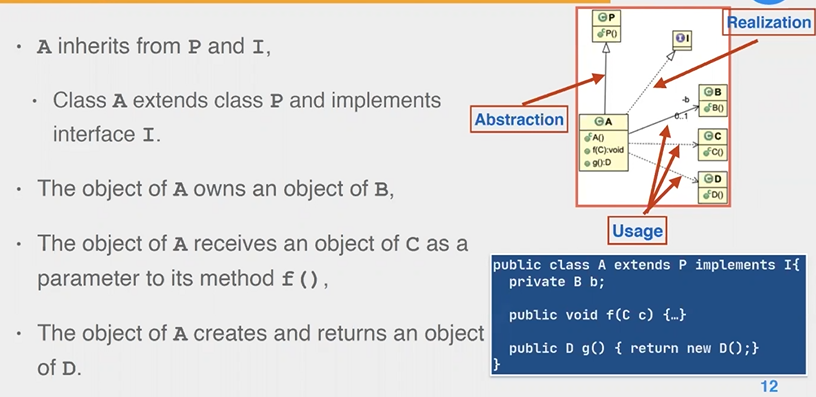
Coupling -> degree of dependency (bağlanmanın derecesi) (low Coupling – high coupling)

Dependency Types

Abstraction: Client ve supplier, aynı kavramı farklı soyutlama seviyelerinde veya farklı bakış açılarından temsil eder.

Realization: Gerçekleştirme, iki nesne kümesi arasında özelleşmiş bir Soyutlama(Abstraction) bağımlılığıdır bunlardan biri bir spesifikasyonu(supplier) temsil ederken, diğeri o spesifiksyonun bir uygulamasını(istemciyi) temsil eder.

Usage: İstemci, tam bir uygulama veya çalıştırma için bu hizmete ihtiyaç duyar.



Abstraction ve Realization -> is-a relationship

Client parent’ın daha spesifik halini sunar. Parent ise client’ın daha genel halidir.(Abstraction)

Client, üst sınıfta (parent) soyut olarak tanımlanmış davranışın somut bir uygulamasını sağlar.(Realization)

Abstraction -> implementation inheritance

Realization -> interface inheritance

Usage Dependency

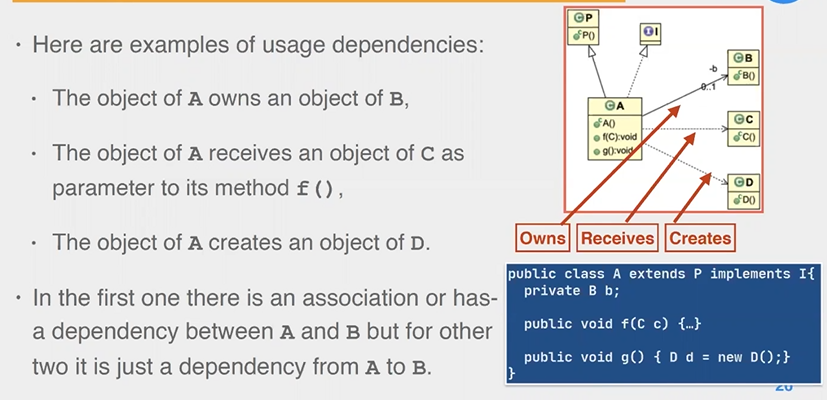
Client service olmazsa çalışamaz.

Service a ihtiyaç olma durumu iki şekilde olur: Definition veya implementation

Association: Definition için yapılan bağımlılığa denir.

Bir nesnenin başka bir nesneye sahip olma ilişkisidir.

Association -> has-a relationship



Aggregation: Diğer objeleri bir araya getirme.

Composition: Bir service objesi bir tane client objesinin parçasıdır. Client’ı memory den silersek service objesi de silinir.

Dependency birlikte anlama ve birlikte değişme problemi ortaya koyar.

Client’ı anlamak için service’ı anlamak zorundayız. Client’ı service den bağımsız anlayamayız.

Dependency Injection

Client’ın service’a ulaşması için farklı senaryolar var.

Managing Dependency: Client create service object, bu durum güçlü bir bağımlılığa işaret eder.

Bu tür bağlılık, yüksek bağlılık (high - coupling) yaratır. Çünkü service nesnesinin varlığı tamamen client nesnesinin varlığına bağlıdır.

Service nesnesi yaşam döngüsü açısından yalnızca client nesnesine bağlıdır; yani bir service nesnesi, sahibi olan client yaşadığı sürece yaşar.

Service nesnesini oluşturmak client için yük olur.

Çoğu zaman bir nesneyi oluşturmak kullanmaktan daha zordur.

Çözüm: bir service nesnesi oluşturup client nesnesine vermek olabilir.

Eğer client nesnesi service nesnesini oluşturmaz da bunun yerine service nesnesini dışarıdan alırsa, bu durum daha gevşek bir bağlılığa (loose coupling) işaret eder.

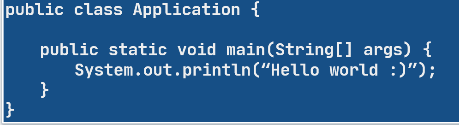
Dependency’yi daha iyi yönetmek adına bağımlı olunan objeyi bir başka yerde yaratıp onu kullanacak objenin emrine vermeye dependency injection denir.

Program to an interface not an implementation.

Dependency injection ile biz bir nesnenin kendisini kullandığı diğer nesneleri oluşturma görevini onun üzerinden alarak aslında client’ın o kullanan nesneyi daha odaklı coupling’i daha düşük yapmış oluyoruz. (Daha odaklıdan kasıt, bir şeye odaklanıyor o nesneyi kullanmak.)

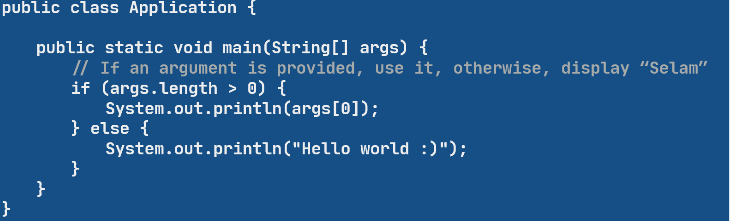
Dependency injection allows the client only know the interface of the service objects it uses not their implementation.

Examples:



Mesajı değiştirirsek ne olur?

Better Hello World

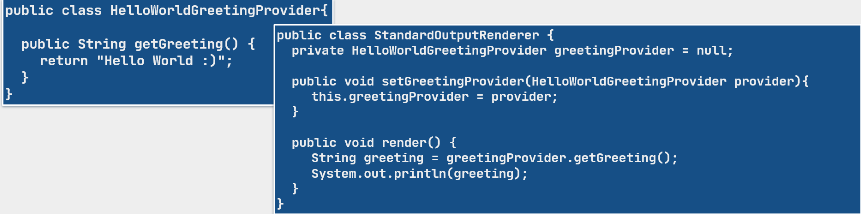


System.out.println hem mesajı alıyor hem de yazdırıyor.

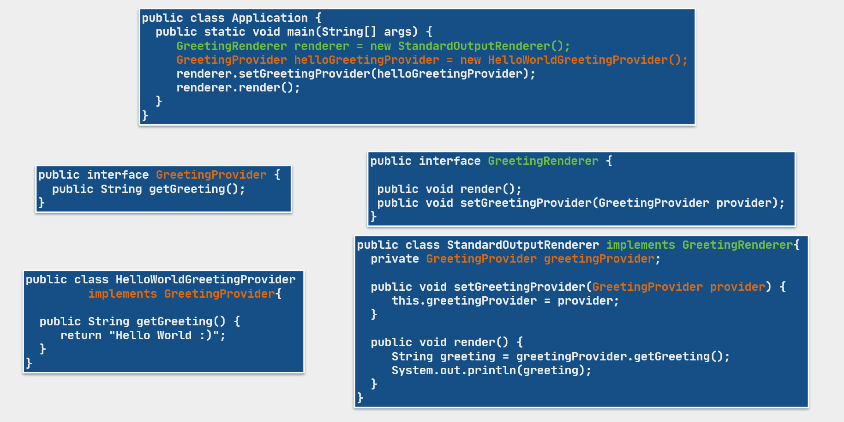
Mesajı almak ile mesajı yazmak farklı sorumluluklar.

Mesajı almak ile yazdırmak sorumluluklarını ayıralım.





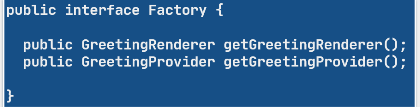
Gerçek nesnelere değil soyutlamalara bağlı olun.

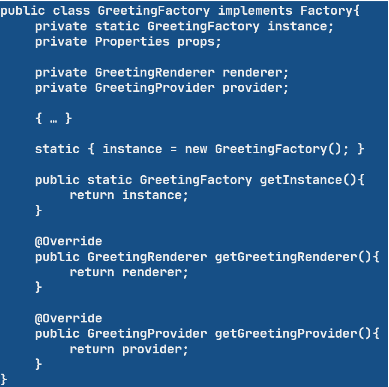


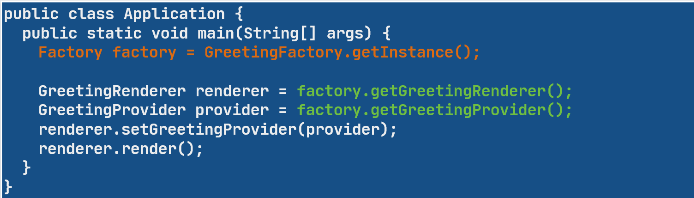
Hala objeleri oluşturma sorumluluğu Application üzerinde.

Ne yapılmalı?

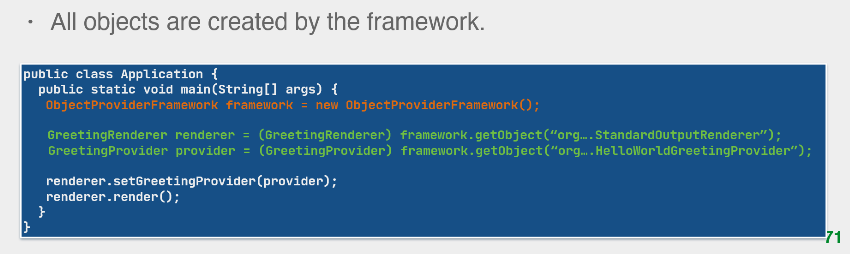
Use of Factories





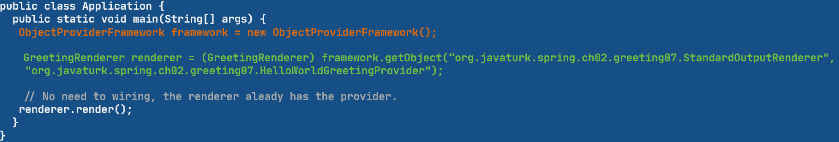


Framework of Factories



Wiring’den kurtulmak lazım

Framework of Objects



Spring Framework for Objects



INVERSİON OF CONTROL

Akışın kontrolü

Bir uygulamadaki akışın adımlarını ne zaman neyi nerede çağırıcaz. Bunları kontrol etmektir, akışın kontrolü

Inversion of control -> kontrolün ters çevrilmesi

Her thread kendi akışını kontrol eder. Dolayısıyla uygulama kendi akışlarını kontrol eder.

Bu noktada application kendi akışlarının sahibidir.

FRAMEWORK

Component bitmiş bir yapıdır ve onla onun API’sini kullanarak sadece interface’ini kullanarak iç yapısını hiçbir şekilde görmeden bilmeye gerek kalmadan ondan hizmet alınır.

Component’ler son ürünken Framework’ler son ürün değildir. Semi-completed application dır.

Ciddi soyutlamalar sağlar. Tekrar kullanılmak üzere.

Framework’lerin application’lar tarafından kullanılması için extends edilmesi gerek.

Framework’ler son ürün olmadığından kullanılabilmeleri için tamamlanmaları gerek.

Tamamlanmaları için de extend edilmeleri gerek. Dolayısıyla extension dediğimiz yerler hook point dediğimiz yerler.

Framework’lerin farklı türleri vardır.

Application development frameworks

Web frameworks -> JSF, ADF, Spring MVC

Security frameworks -> JAAS, Spring Security

Business domain frameworks

Insurance frameworks -> Acord

Akış kontrolü application’da. Ama;

Application’da framework’ü ayağa kaldırdığımız zaman framework o kontrolü kendisi ele alır.

Framework akışın kontrolüne sahip olduğunda, kontrolün tersine çevrilmesi (inversion of control) gerçekleşir.

Framework, ne zaman neyi çağıracağına karar verir.

Application, framework tarafından kullanılan bir component haline gelir.

Ancak bir Application bir component’i veya library’ı kullandığında, akışın kontrolünü vermez; kendi nesneleri ya da kütüphanelerin nesneleri üzerinde ne zaman çağrı yapılacağını kendisi kontrol eder.

Framework Kullanmanın Faydaları

Daha hızlı develop yapmayı sağlar.

Tekrarlayan ve standart kodları yazma işini framework'e bırakarak geliştiricinin asıl iş mantığına (yani uygulamanın değer katan kısmına) odaklanmasını sağlamak.

Daha güvenilir kod

Spring framework’ün kendine ait bir IoC container’ı ve DI mekanizması var. Objeleri yaratmak ve yaşam döngülerini ve bağımlılıklarını yönetmek için.

Spring IoC konteyneri, nesneleri oluşturmak ve bağımlılıkları yönetmek için, uygulamanın nesneleri çektiği fabrika metotları gibi başka yollar da sağlar.

Ancak IoC (Inversion of Control) ve DI (Dependency Injection), Spring çerçevesinin merkezindedir ve Spring, uygulanabilir olduğu her yerde bunu kullanır.

Bean Nedir?

Spring’in IoC container’ı tarafından yönetilen objelere beans denir.

Bean, Spring IoC konteyneri tarafından oluşturulan (örneklenen), bir araya getirilen ve diğer yönleriyle yönetilen bir nesnedir.

Bean'ler ve aralarındaki bağımlılıklar, konteyner tarafından kullanılan yapılandırma (konfigürasyon) metadata'sında yansıtılır (tanımlanır).

Uygulamadaki Spring tarafından oluşturulmayan tüm nesneler veya bean'ler, Spring tarafından yönetilmez.

BeanFactory

Spring’in Bağımlılık Enjeksiyonu (Dependency Injection) işlevselliği, **BeanFactory** ve onun alt türleri kullanılarak gerçekleştirilir.

**BeanFactory**, Spring içinde bean’leri yönetmek için temel işlevselliği sağlar.

Yönetim işlemi, öncelikle nesnelerin oluşturulmasını ve ardından bağımlılıklarını karşılamak üzere bu nesnelerin birbirlerine enjekte edilmesini içerir.

APPLICATION CONTEX (bir container yapısı)

Container’ı BeanFactor’den daha yetkin bir şekilde temsil eden inteface’in ismidir.

Farklı application’larda farklı implementation’ları vardır.

Resource yükleme var. ApplicationContext, uygulama içindeki **kaynak (resource) dosyalarını** (örneğin .properties, .xml, vb.) **yükleyebilir ve yönetebilir.**

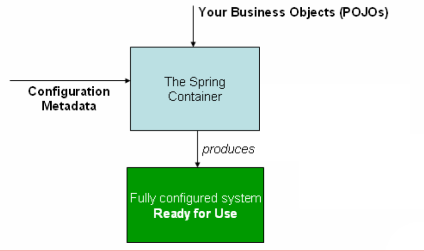
PROPERTY BEAN DEFINATION FILES

Bean’ler, XML dosyalarında veya properties (özellik) dosyalarında tanımlanabilir.

CONFIGURATION METADATA

Spring IoC konteyneri, bean’leri oluşturmak, yapılandırmak ve birleştirmek için talimatlarını yapılandırma (konfigürasyon) metadata’sını okuyarak alır.

Spring yapılandırması, konteynerin yönettiği en az bir ve genellikle birden fazla bean tanımından oluşur.



Configuration metadata’yı göstermenin Spring’de 3 farklı yolu var. 1-XML, 2-Java annocations, 3-Java code.

En klasiği XML.

XML CONFİGURATİON METADATA

Yapılandırma metadata’sını tanımlamak için XML dosyalarını kullanmak, uygulama bağlamını (application context) tanımlamanın klasik yoludur.  
XML tabanlı yapılandırma metadata’sı, bean’leri üst düzey bir <beans/> öğesi içinde yer alan <bean/> öğeleri olarak tanımlar.  
Bean’ler, bağımlılıkları için birbirlerine referans verirler.  
Yapılandırma için birden fazla XML dosyası kullanılabilir.  
XML yapılandırma dosyaları, ApplicationContext implementasyonlarının kurucularına (constructors) parametre olarak geçirilir.

id niteliği (attribute), bir bean tanımını tanımlamak (belirlemek) için kullanılan bir karakter dizisidir (string).

class niteliği (attribute), bean’in türünü, sınıfının tam nitelikli adı (fully qualified name) ile tanımlar.

Ref -> referans (diğer beanlara referans vermek öyle ki birbirlerine inject edilebilsin)

Bir bean id’si, aynı <beans/> öğesi içinde birden fazla kez kullanılamaz; aksi takdirde Spring,  
org.springframework.beans.factory.parsing.BeanDefinitionParsingException istisnasını (exception) fırlatır.

class niteliğinde belirtilen bean türü, bir sınıf (class) olmalıdır ki ondan bir örnek (instance) oluşturulabilsin.

Aksi takdirde, yani belirtilen tür soyut bir sınıf (abstract class) veya bir arayüz (interface) ise, Spring  
org.springframework.beans.BeanInstantiationException istisnasını (exception) fırlatır.

<bean/> türleri;

Parent

Scope

Abstract

Lazy-init

Autowire

Bean yapılandırmalarını düzgün bir şekilde düzenlemek için birden fazla XML yapılandırma dosyası kullanılabilir.  
DAO’lar, backing bean’ler ve işlemler (transactions) gibi bileşenler, farklı dosyalarda tanımlanabilir veya tanımlanmalıdır.  
Bu amaçla, <beans/> öğesi içinde bir veya daha fazla <import/> öğesi kullanılır.

Bean’lerin bir veya birden fazla benzersiz tanımlayıcısı (identifier) olabilir.  
XML yapılandırmasında id ve name tanımlayıcı olarak kullanılır.

* id yalnızca tek bir değer alabilir.
* name içinse, virgül (,), noktalı virgül (;) veya boşluk ( ) ile ayrılmış alfanümerik değerler kullanılabilir.

Bu name değerlerinin tümü, bean’e ait takma adlar (alias) olarak kabul edilir ve büyük sistemlerde farklı bağlamlarda aynı bean’e referans vermek için kullanılabilir.

id ve name zorunlu değildir; bu durumda bean’ler bu tanımlayıcılarla sorgulanamaz, yalnızca türlerine (type) göre erişilebilirler.  
Eğer id veya name belirtilmemişse, Spring bu bean’ler için otomatik olarak isim üretir.

Bean’ler için anlamlı ve **lower camel case** (küçük harfle başlayıp sonraki kelimelerin büyük harfle devam ettiği) adlandırma kullanılması önerilir.

**Classpath taraması (classpath scanning)** durumunda, bileşenler (component) için name belirtilmezse, Spring bileşenin sınıf adını kullanarak, ilk harfini küçük harfe çevirerek bir ad üretir.  
Örneğin: CustomerService sınıfı için otomatik ad customerService olur.

XML’i değiştirdiğimiz zaman source kodu değiştirmeden davranışı değiştirebiliriz.

Allocation’ları kullandığımız zaman source kodunu değiştirmek gerekir.

XML’ler daima allocation’ları ezer.

Daima son sözü söyleyen XML’dir.

XML yapılandırma dosyasında kök (root) öğe mutlaka <beans/> olmalıdır.  
Bir XML yapılandırma dosyasında yalnızca bir tane en üst düzey <beans/> bildirimi bulunabilir; bu XML'in kuralıdır.

Bir XML yapılandırma dosyasında iç içe <beans/> bildirimleri (yani bir <beans/> içinde başka bir <beans/>) yapılabilir; bu, alt küme bean tanımları yapmak için kullanılır.

Eğer iç içe <beans/> bildirimi varsa, içteki <beans/> bildirimi mutlaka dıştaki <beans/> bildiriminin *son* öğesi olmalıdır, aksi halde ayrıştırma (parsing) hatası oluşur.

Yani, bir <beans/> bildiriminden sonra hiçbir şey bulunamaz.

Alias attribute değil bir xml elementidir.

Her **alias**, bean'e yeni bir takma ad kazandırır ve yalnızca bir değer alabilir.  
Alias, farklı bir XML kaynağında da tanımlanabilir.

Spring önce **id**'ye bakar, eğer bulunamazsa **name** öğelerinden ilkine bakar.  
Aslında **id** ve **name** tanımlanmak zorunda değildir; bu durumda bean'ler bu isimlerle sorgulanamazlar, yalnızca **türlerine (type)** göre erişilebilirler.  
Eğer **id** veya **name** belirtilmezse, Spring bean için **sınıf adının tamamını kullanarak** otomatik olarak bir ad üretir.

class niteliğinde belirtilen bean türü mutlaka bir **sınıf** olmalıdır, **ancak** <bean/> etiketi içinde **abstract** olarak tanımlanmadıysa.  
Eğer <bean/> etiketinde **abstract** niteliğine **true** değeri verilirse, Spring bu bean’in bir örneğini (instance) oluşturmaz.

Bean'ler arasındaki **kalıtım ilişkisi**, <bean/> etiketindeki **parent** niteliği kullanılarak tanımlanabilir.  
**parent** niteliği, **ebeveyn bean’in tanımlayıcılarından (id veya name)** birini değer olarak alır.  
Bunun amacı, **alt bean’in ebeveyn bean’in tanımını miras almasını** sağlamaktır.

Eğer bir **alt bean** (child bean) herhangi bir **özellik (property)** tanımlamazsa ve bir **parent** niteliği varsa, **ebeveyn bean’de tanımlı olan tüm özellikleri miras alır**.

**ApplicationContext**, varsayılan olarak Spring IoC container’ı başlatılırken **bean’leri önceden (eagerly) oluşturur**.  
Bu davranışı değiştirmek için <bean/> etiketinin **lazy-init** niteliği kullanılır.  
**lazy-init** niteliğinin örtük (varsayılan) değeri **false**’dur ve bu değer **true** ya da **false** olarak ayarlanabilir.  
**Tembel başlatma (lazy initialization)** davranışı, **<beans/>** öğesindeki **default-lazy-init** niteliği ile tüm container için yapılandırılabilir.  
Ancak belirli bir bean için **<bean>** etiketinde **lazy-init** belirtilirse, bu ayar **<beans/>** seviyesindeki genel ayarı **geçersiz kılar (override eder)**.

Öte yandan, bir bean **ister önceden (eagerly)** ister **tembel (lazily)** oluşturulsun, **bağımlı olduğu diğer tüm bean’ler (işbirlikçileri - collaborators) de oluşturulur**.

Yani bir bean başlatıldığında, ona bağımlı olan diğer bean'ler de otomatik olarak yaratılır. Bu durum, bağımlılık zincirinin tamamının hazır olması için önemlidir.

Bu nedenle, **tembel (lazy) başlatılan bir bean**, **tembel olmayan (eager) bir bean’in bağımlılığı** ise, **ApplicationContext**, bağımlılıkları karşılamak için **lazy bean’i de uygulama başlatılırken oluşturur**.

Yani bağımlılık zincirinde bir bean lazy olsa bile, eğer eager bir bean ona ihtiyaç duyuyorsa, Spring onu da hemen yaratır. Bu, bağımlılıkların eksiksiz çalışmasını sağlar.

**ApplicationContext** bir bean oluşturduğunda, onu **tekil (singleton)** olarak yapılandırır; yani yalnızca **bir örneği** (instance) olur.

Bu, **kapsam (scope)** ile ilgili bir konudur ve <bean/> etiketinin **scope** niteliğiyle belirtilebilir.  
**scope** niteliğinin örtük (varsayılan) değeri **singleton**’dır ve bu değer, **prototype** gibi farklı kabul edilebilir String değerlerle değiştirilebilir.

Eğer bir bean **prototype** olarak tanımlanırsa, **ApplicationContext nesnesinden her istek yapıldığında yeni bir bean örneği oluşturulur**.

Bir bean **singleton** olarak tanımlandığında, **ApplicationContext**, eğer tembel başlatma (lazy initialization) belirtilmemişse, onun örneğini **önceden (eagerly)** oluşturur.

Ancak bir bean **singleton olmayan** (örneğin **prototype**) olarak tanımlanmışsa, **ApplicationContext** onun örneğini **yalnızca uygulama onu çağırmaya çalıştığında** oluşturur.

Eğer bir bean **singleton** olarak tanımlanmışsa, **en fazla bir örneği** (instance) olur ve bu örnek, **içine enjekte edildiği diğer tüm bean’lerle paylaşılır**.

Singleton olan işbirlikçi (collaborator) bean’i alan tüm bean’ler **aynı örneği kullanır** ve bu durum, bean’ler arasında **paylaşılan bir durum (shared state)** oluşturur.

Öte yandan, bir bean **prototype** olarak tanımlanmışsa, **enjekte edildiği her bean, o işbirlikçi bean’in kendine ait bir örneğini alır**.

Bu durumda, işbirlikçi bean’in örnekleri **paylaşılmaz**; her biri, enjekte edildiği bean’in **özel durumu (private state)** olur.

Singleton – Prototype

Elbette **çalışma zamanı (run-time) verimliliği** açısından daha az nesne oluşturmak daha iyidir; bu nedenle genellikle **singleton** tercih edilir.  
Ancak çoğu zaman **çalışma zamanı verimliliği**, seçimdeki **tek etken değildir**.

Bu konudaki kilit kavram **durum (state)**’tur.  
Yani, **işbirlikçi (collaborator) bean’in bir durumu (state) var mı**, eğer varsa, **bu durum enjekte edildiği bean’ler tarafından değiştirilebiliyor mu** ve **uygulama çok iş parçacıklı (multi-threaded) bir ortamda mı çalışıyor**, bunlar dikkate alınması gereken önemli faktörlerdir.

Yani bir bean durum tutuyorsa ve bu durum değiştirilebilirse, özellikle çok iş parçacıklı ortamlarda prototype tercih edilmesi gerekebilir.

Eğer işbirlikçi bean’in bir durumu yoksa, yani **durumsuz (stateless)** ise, bu bean **singleton** olarak tanımlanmalıdır.  
Bu durumda, işbirlikçi bean'e yapılan tüm metod çağrıları yalnızca **yerel (local) değişkenler** kullanır ve bu değişkenler **iş parçacığı güvenli (thread-safe)** olur.

Eğer işbirlikçi bean’in bir durumu varsa, yani **durumlu (statefull)** ise, bu durumda soru şudur:  
**Enjekte edildiği bean’ler bu durumu değiştirmek istiyor mu?**

Eğer işbirlikçi bean’in durumu **değiştirilemez (immutable)**, yani sadece **okunabilir**, **yazılamaz** ise, bu durumda da **singleton** olarak tanımlanmalıdır.

Kısacası, durum yoksa veya durum değiştirilemiyorsa → singleton;  
Durum varsa ve değiştiriliyorsa → dikkatli olmak gerekir, genellikle prototype tercih edilir.

Eğer işbirlikçi bean **değiştirilebilir (mutable) bir duruma** sahipse, o zaman soru şuna dönüşür: **Ortam çok iş parçacıklı (multi-threaded) mı, değil mi?**  
Spring tarafından oluşturulan bean’ler **iş parçacığı güvenli (thread-safe)** değildir.

Eğer işbirlikçi bean’in durumu değiştirilebiliyorsa, onu **çok iş parçacıklı bir ortamda paylaşmak**, **yarış durumlarını (race conditions)** ve **bean’in durumunun bozulmasını (state corruption)** önlemek için **eşzamanlılık (synchronization)** gerektirir.

Bu durumda, **prototype bean’ler kullanmak daha uygun olabilir**, çünkü bu yaklaşım hem **çalışma zamanı verimliliği**, hem de **kod karmaşıklığı** açısından daha etkili olabilir.

BEAN INSTANTIATION

Spring, **tanımlanmış bean’lerin örneklerini yansıtma (reflection)** yoluyla oluşturur.  
Bağımlılıkların nasıl enjekte edildiğine bağlı olarak, ya **varsayılan (parametresiz) bir kurucu (constructor)** ya da **uygun parametrelere sahip bir kurucu** sağlanmış olmalıdır.

Spring, bean’in örneğini oluşturmak için ya **kurucuyu (constructor) çağırır**,  
ya da eğer belirtilmişse, **aynı sınıf üzerinde tanımlı bir statik fabrika metodunu** kullanır.

Spring, örneği oluşturmak için **statik bir fabrika metodunu (static factory method)** çağırır.  
Bunun için, statik fabrika metodunun adı <bean/> etiketindeki **factory-method** niteliğinde belirtilmelidir.  
Spring, bu durumda da bean’e tanımlanan **tüm bağımlılıkları enjekte etmeye devam eder**.

Spring, örneği oluşturmak için **başka bir bean üzerindeki örnek (instance) fabrika metodunu** çağırabilir.  
Bunun için, **statik olmayan (non-static) fabrika metodunu içeren sınıfın bean adı** ve **fabrika metodunun adı**, <bean/> etiketindeki **factory-bean** ve **factory-method** niteliklerinde belirtilmelidir.  
Bu durumda, **class** niteliği atlanabilir.  
Spring, yine de nesneye tanımlanan **tüm bağımlılıkları enjekte etmeye devam eder**.

SPECIFYING DEPENDENCIES: CONSTRUCTOR & PROPERTY INJECTION

**Dependency Injection** (Bağımlılık Enjeksiyonu) sürecinde, bir bean’in **bağımlı olduğu tüm işbirlikçileri (collaborators)**, o bean’e **enjekte edilir**.  
**XML tabanlı yapılandırmada**, başlıca iki tür enjeksiyon yöntemi vardır:

1. **Constructor Injection (Yapıcı metod ile enjeksiyon)**
2. **Property veya Setter Injection (Özellik ya da setter metodu ile enjeksiyon)**

Her iki yöntemde de, bağımlı olunan başka bir bean’e başvurmak için **ref** niteliği,  
primitive bir değer veya String iletmek için ise **value** niteliği kullanılır.

Constructor Injection

**Constructor Injection (Yapıcı metod ile enjeksiyon)** için <constructor-arg/> etiketi kullanılır.  
Bağımlı olunan bean, <constructor-arg/> etiketi içinde yer alan **ref** öğesi ile belirtilir ve bu öğe ilgili bean’in **id, name** ya da **alias** değerini alır.

Yapıcı metot parametrelerinin çözümlemesi (resolution) esas olarak **parametrenin tipi** kullanılarak yapılır.  
Bu çözümlemeye yardımcı olmak için **type**, **index** ve **name** nitelikleri kullanılabilir.

Primitive türler ve String nesneleri enjekte etmek için **value** niteliği kullanılır.

Property Injection

**Property Injection (özellik enjeksiyonu)** için <property/> etiketi kullanılır.  
Bağımlı olunan bean, <property/> etiketi içindeki **ref** öğesiyle belirtilir ve bu öğe ilgili bean’in **id**, **name** veya **alias** değerini alır.

Bağımlılığın çözümü (resolution), temelde **setter metodun adlandırma kurallarına (naming convention)** göre yapılır.  
Primitive türler ve String nesnelerini enjekte etmek için **value** niteliği kullanılır.

İmmutable -> constructor

Zorunlu olan field’lar için yani NullPointerException fırlatması söz konusu olan dependency’ler varsa onların muhakkak inject edildiğinden emin olmak istiyorsak her halükarda constructor injection’ı koyarız.

Inner Bean

Bazen bir bağımlılık doğrudan <bean/> etiketi kullanılarak, **id** veya **name** belirtilmeden tanımlanır; bu durumda bu bean'e **inner bean (iç bean)** denir.  
Bu durumda, **<property/>** veya **<constructor-arg/>** etiketlerinin içinde yer alan bir <bean/> elementi, doğrudan bir **inner bean sınıfına** başvurur.

SPECIFYING DEPENDENCIES: AUTOWIRE

<bean/> etiketindeki **autowire** niteliği, Spring IoC container’ının **bağımlılıkları otomatik olarak bulup karşılamasını** sağlar.  
Bağımlı olunan bean’ler yine de **XML dosyasında tanımlanmış olmalıdır**.

Bu yöntem, **daha sade XML yapılandırma dosyaları** sağlar çünkü <constructor-arg> ve <property> etiketlerine gerek kalmaz.  
Tüm bağımlılıklar, **autowire niteliğine verilen değerin belirlediği kurallara** göre çözülür.

Autrowire attribute 3 farklı değeri var:

Constructor

byName

byType

default value is no

**autowire** niteliğini kullanmanın bazı **sınırlamaları** vardır:

* **Primitive türler** ve **String nesneleri** otomatik olarak enjekte edilemez.
* **Autowiring**, bazen **belirsizliklere (ambiguities)** yol açabilir (örneğin, aynı türde birden fazla bean tanımı varsa).

Bu nedenle, **bağımlılıkların XML dosyasında açıkça belirtilmesi**, bu tür problemleri **önlemek** açısından daha güvenli bir yaklaşımdır.

VALUE INJECTION

**Bean dışındaki değerler** (örneğin primitive türler veya String'ler), **constructor** ve **setter metodlarına** enjekte edilebilir.  
Bu amaçla, bir bean’e başvurmak için kullanılan **ref** niteliği yerine, <constructor-arg/> veya <property/> etiketlerinde **value** niteliği kullanılır.  
Ayrıca, **iç içe yerleştirilmiş <value>...</value> etiketi** de kullanılabilir.

**value** niteliği, **primitive türleri**, **String sabitlerini** ve bunların **koleksiyonlarını** (array, list, set ve map) iletmek için kullanılır.  
Tüm **tip dönüşümleri**, **Spring tarafından otomatik olarak** gerçekleştirilir.

COLLECTIONS INJECTIONS

**Bean’lerin veya değerlerin koleksiyonları**, **constructor** ve **setter metodlarına** enjekte edilebilir.  
Koleksiyon olarak **array**, **List**, **Set** ve **Map** türlerinin implementasyonları kullanılabilir.

XML dosyasında bu koleksiyonları tanımlamak için sırasıyla:  
<array>, <list>, <set> ve <map> etiketleri kullanılır.

Bu etiketlerin içinde:

* Değerler için **iç içe <value>...</value> etiketleri**,
* Bean'ler için ise **iç içe <ref bean='...'/>** kullanılır.

Değerler söz konusu olduğunda, **Spring tüm tip dönüşümlerini otomatik olarak** gerçekleştirir.

**Array**, **List** ve **Set**, doğrudan **değerleri** veya **bean referanslarını** alırken,  
**Map**, **anahtar-değer (key-value) çiftlerini** alır; bu çiftlere **entry (girdi)** denir.

Bu nedenle <map> etiketi içinde **<entry>** etiketleri kullanılır ve bu etiketin üç temel niteliği vardır:

* **key**: Anahtar bir değer ise kullanılır.
* **value**: Değer bir değer (literal) ise kullanılır.
* **value-ref**: Anahtar veya değer bir bean ise, referans belirtmek için kullanılır.

Ayrıca, **bean’leri belirtmek için iç içe <ref bean='...'/> kullanımı da mümkündür**.

ANNOTATION-BASED CONFIGURATION

@Autowired

<beans/> öğesi içinde yer alan <context:annotation-config/> etiketi **gerekli**dir.  
Bu etiket sayesinde Spring, **@Autowired** gibi anotasyonlar aracılığıyla **bağımlılıkları otomatik olarak algılar**.

Ancak, **bean’lerin XML yapılandırmasında tanımlanmış olması gerekir** — yani **bağımlılıkları belirtilmese bile**, bean tanımlarının XML dosyasında bulunması zorunludur.

**@Autowired** anotasyonu aşağıdaki yerlere uygulanabilir ve her zaman **önüne** yazılır:

* **Nesne değişkeni (instance variable)**
* **Yapıcı metod (constructor)**
* **Setter metodu**
* **Herhangi bir başka metot**

Tabii ki, **@Autowired** ile işaretlenen her öğe, XML dosyasında <bean/> etiketiyle tanımlanmış bir bean tarafından **enjekte edilebilir (injectable)** olmalıdır.

Spring, **@Autowired** ile işaretlenmiş **her bir noktaya** bağımlılıkları enjekte eder.

Eğer bir bean sınıfında **yalnızca bir tane constructor** varsa ve bu constructor **enjekte edilebilir bir bağımlılık alıyorsa**,  
**o constructor için @Autowired anotasyonunu kullanmaya gerek yoktur.**

Spring, bu durumda **bağımlılığı otomatik olarak algılar** ve bean’in constructor’ına geçirerek **otomatik enjeksiyon (auto-wiring)** işlemini gerçekleştirir.

@Required

**@Required** yalnızca **setter metodlarla** birlikte kullanılabilir.

**@Required** ile işaretlenen bağımlılık, mutlaka karşılanmalıdır; aksi takdirde Spring,  
**org.springframework.beans.factory.BeanInitializationException** istisnasını fırlatır ve “**property is required**” (özellik zorunludur) şeklinde bir hata mesajı verir.

Bu bağımlılık, ya **XML dosyasındaki <bean> etiketinin property niteliği** ile  
ya da **@Autowired** anotasyonu kullanılarak sağlanmalıdır.

DEFINING BEANS OUTSIDE XML

<beans/> öğesi içinde kullanılan <context:annotation-config/> etiketi ve **@Autowired** anotasyonu,  
Spring’in **sadece bağımlılıkları otomatik olarak algılamasını** sağlar,  
ancak **bean’lerin XML dosyasında tanımlı olması hâlâ gereklidir**.

Spring, bean’leri **XML yapılandırma dosyası dışında tanımlamaya** da izin verir. Bu amaçla iki mekanizma kullanılır:

1. **@Component anotasyonu**
2. **Bean factory metodlar**

Component

org.springframework.stereotype.Component bir sınıfın bileşen (component) olduğunu belirten bir notasyondur.  
@Component sınıfların başına yazılır ve bu sınıfların örneklerinin (instance) Spring bean'leri olmasını sağlar.  
Bu tür sınıflar, anotasyon tabanlı yapılandırma ve classpath taraması (classpath scanning) kullanıldığında otomatik olarak algılanacak adaylar olarak kabul edilir.  
Sınıfların bileşen olarak taranabilmesi <beans/> XML dosyasında <context:component-scan/> etiketi kullanılarak mümkündür.

<context:component-scan/> öğesi, bean taramasının nereden başlayacağını belirtmek için base-package adlı bir öznitelik içerir.  
base-package özniteliğinin kullanılması zorunludur ve bu öznitelik, taranacak anotasyonlu bileşenlerin bulunduğu paketlerin virgül, noktalı virgül, boşluk veya satır sonuyla ayrılmış bir listesini alır.  
Spring, base-package içinde belirtilen tüm paketleri ve onların alt paketlerini tarayarak anotasyonla işaretlenmiş bileşenleri otomatik olarak bean olarak kaydeder.

component scan ve @Component anotasyonunun kullanımı, XML yapılandırma dosyalarını tüm bean tanımlarından kurtarır ve bu dosyaları oldukça kısa hale getirir.  
<context:component-scan/> öğesi, dolaylı olarak <context:annotation-config> öğesinin işlevini de etkinleştirir; bu da @Autowired anotasyonunun kullanılmasını sağlar.  
Bu nedenle, <context:component-scan/> kullanıldığında <context:annotation-config> öğesini ayrıca eklemeye gerek yoktur.

@Component anotasyonunun yalnızca value adında bir özniteliği vardır ve bu öznitelik, bean’e verilecek ismi temsil eden bir String alır.  
Eğer value özniteliği kullanılmazsa, Spring varsayılan olarak bean’in sınıf adının sade halini (simple name) alır ve bunu küçük harfle başlayan camel case biçimine (lower camel case) çevirerek bean ismi olarak kullanır.

Spring farklı görevler için birçok farklı bileşen(component) sağlar.

@Component, @Repository, @Service, @Controller, @RestController, @ControllerAduice ve @Configuration

QUALIFYING BEANS

Spring, hangi beani enjekte edeceğine karar veremediğinde **UnsatisfiedDependencyException** hatasını fırlatır.

Bu durum genellikle aşağıdaki sebeplerden kaynaklanır:

* Aynı türden birden fazla bean tanımlanmışsa ve hangisinin enjekte edileceği belirtilmemişse,
* Gerekli olan bean hiç tanımlanmamışsa,
* @Autowired anotasyonu ile bir bağımlılık istenmiş ancak Spring bunu çözümleyememişse.

Yani özetle: Spring, bağımlılığı karşılayacak uygun bir bean bulamazsa veya birden fazla uygun bean varsa ve hangisinin seçileceği belli değilse bu istisnayı fırlatır.

Problemin çözümü için iki farklı yol var.

Naming convention: Enjeksiyon noktasında, yani bir değişkenin (field) veya parametrenin adını, bean ismiyle **eşleşecek şekilde** kullanmaktır.

Bu kural sayesinde Spring, hangi bean'in enjekte edileceğini daha kolay belirleyebilir. Özellikle aynı türde birden fazla bean varsa, **değişken veya parametre adının**, ilgili bean’in adıyla **aynı olması**, otomatik eşleşme sürecinde fayda sağlar.

Bu çözüm, bileşen adlarının (component names) **kullanıcıları (clients)** tarafından kullanılmasını gerektirir  
ve **adlandırma kuralına (naming convention)** dayanır.

Yani, bağımlılığı alan sınıf (client), hangi beani kullanacağını **bean ismiyle eşleşen bir değişken/parametre adı** kullanarak belirtmelidir.  
Bu yöntem, Spring’in otomatik olarak doğru beani eşleştirebilmesi için isimlerin tutarlı olmasına bağlıdır.

Qualifier

org.springframework.beans.factory.annotation.Qualifier, bu probleme daha iyi bir çözüm sunar.

Bu anotasyon, otomatik bağlama (autowiring) sırasında aday bean'ler arasında seçim yapılabilmesi için **bir alan (field)** veya **parametre** üzerinde bir niteleyici (qualifier) olarak kullanılır.

@Qualifier anotasyonu yalnızca bir adet String türünde argüman alır.  
Bu değer, birden fazla aday bean olduğunda **hangisinin enjekte edileceğini belirtmek** için kullanılır.

Bu value genellikle **enjekte edilmek istenen bileşenin (component) adı** olmalıdır.

Qualifier iki yerde kullanılır:

1. **Bir sınıfı bean olarak tanımlarken** @Component ile birlikte,
2. **Bean’i enjekte ederken** @Autowired ile birlikte.

Eğer bu iki yerde belirtilen @Qualifier değerleri **birbiriyle eşleşirse**, Spring doğru bean’i enjekte eder.

qualifier öğesi, XML yapılandırma dosyasında da kullanılabilir ve </bean> öğesinin içinde **iç içe (nested)** bir öğe olarak yer alır.

<qualifier> etiketi de yalnızca bir adet String niteliğe sahiptir ve bu nitelik, @Autowired ile bağımlılık enjekte edilirken hangi bean'in seçileceğini belirtmek için kullanılır.

@Primary

Primary olarak annotate edilen bean’e öncelik sağlıyor. Birden fazla injection için aday bean olduğu zaman.

Birden fazla bean’de kullanılamaz. Hata alınır.

Yalnızca bir bean **primary (öncelikli)** olarak tanımlanabilir ve diğer bean'leri seçmek için **qualifier** kullanılmalıdır.

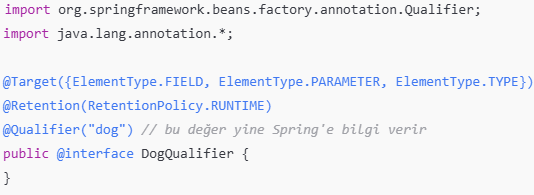
Bunun nedeni, @Primary anotasyonunun kullanılması durumunda **isim temelli eşlemenin (naming-based qualification)** artık geçerli olmamasıdır. Yani @Primary, Spring'e "öncelikli olarak bunu enjekte et" der ve isim eşleşmelerini geçersiz kılar.

Benzer şekilde, XML yapılandırmasında da <bean> öğesinin **primary** niteliği kullanılabilir. Ancak bu yöntemi kullanmak için bean’lerin XML içinde tanımlanmış olması gerekir.

CUSTOM QUALIFIER

@Qualifier ile birlikte String değer kullanmak yerine, **özel (custom) qualifier’lar oluşturmak** daha iyi bir çözüm olabilir.

Bu, @Qualifier türünde **yeni bir anotasyon tanımlanarak** yapılabilir.







### Avantajları:

* **Tip güvenliği sağlar:** Yazım hataları ("dog" yerine "dag" yazmak gibi) ortadan kalkar.
* **Daha okunabilir ve anlamlıdır:** @DogQualifier doğrudan ne yaptığını anlatır.
* **Refactor ve IDE desteği daha iyidir.**

Herhangi bir türde **qualifier (niteleyici) anotasyonunun kullanılması**, **isim temelli eşleştirmeyi (naming-based qualification)** geçersiz kılar.  
Yani, bir @Qualifier kullanıldığında, Spring artık değişken adlarına göre bean seçimi yapmaz.

Ayrıca, @Qualifier ile birlikte @Primary kullanılırsa, **@Qualifier ifadesi geçerlidir**, yani @Primary etkisiz hale gelir.

Bir bileşen (component) **birden fazla qualifier** taşıyabilir; bu qualifier’lar **farklı bağlamlarda (context)** uygulanabilir.

Yani bir bileşen hem **öncelikli (primary) bean** olabilir, hem de **belirli qualifier’lara** sahip olabilir.  
Bu da şu anlama gelir: **Bean**, farklı durumlarda farklı qualifier’lar kullanılarak seçilebilir.

VALUE INJECTION USING @VALUE

@Value

org.springframework.beans.factory.annotation.Value, **harici özellikleri (external properties)** almak için kullanılan bir anotasyondur.  
**Alan (field)** ve **parametre** seviyesinde kullanılır.

Genellikle, **ifade (expression) tabanlı** ya da **özellik (property) tabanlı** bağımlılık enjeksiyonu için kullanılır.

* **İfade tabanlı (expression-driven)** kullanım, **Spring Expression Language (SpEL)** ifadesiyle değer atanması anlamına gelir.
* **Özellik tabanlı (property-driven)** kullanım ise, diğer bean’lerin özelliklerine erişerek değer atanması demektir.

Spring Expression Language (SpEL), çalışma zamanında (runtime) nesne grafiğini sorgulama ve manipüle etme imkanı sağlayan bir ifade dilidir.

Özellikleri şunlardır:

* Sabit (literal) ifadeler
* Boolean ve ilişkisel (relational) operatörler, atama işlemleri
* Düzenli ifadeler (regular expressions)
* Sınıf ifadeleri ve metot çağrıları
* Nesne özelliklerine, dizilere, listelere ve haritalara erişim gibi işlemler.

Every SpEL expression is represented by Expression interface.

@Value anotasyonunun, değeri belirten **value** adında zorunlu bir özniteliği (attribute) vardır.  
Basit string değerler, diğer bean’lerin özellikleri ve daha karmaşık değerler **SpEL** kullanılarak enjekte edilebilir.  
Tüm tür dönüşümleri (type conversions) Spring tarafından otomatik olarak gerçekleştirilir.

Bir **SpEL** öğesi, bir bean’in özelliğinin (property) değerini #{expression} şeklinde tanımlar.  
@Value anotasyonlarında, ifade çözücü (expression resolver) önceden yapılandırılmıştır ve ifade metnini çözerken bean isimlerini arar.  
Bean özelliklerine erişilirken, ilgili **getter metotları** çağrılır.

Ayrıca, bir **SpEL öğesi**, bir özellik dosyasında (properties file) tanımlı bir özelliğin değerini ${property-name} şeklinde belirtir.

Properties File

Properties dosyaları, XML dosyasında <context:property-placeholder location=""> etiketiyle veya kaynak kodda @PropertySource anotasyonu ile belirtilir.  
Her ikisi birden belirtilirse, Spring bu kaynakları birleştirir.  
Eğer aynı isimde birden fazla property varsa, **son belirtilen kaynak** öncelikli olur ve diğerini geçersiz kılar.  
Ayrıca, hem XML hem de anotasyonla belirtildiyse, önce XML’de belirtilen properties dosyası yüklenir, ardından anotasyonla belirtilen dosya yüklenir.

SCOPE

Singleton - Prototype

Spring’in bileşenleri (components) varsayılan olarak **singleton (tekil)** scope’a sahiptir.  
Bu davranış, @Scope anotasyonu kullanılarak değiştirilebilir.  
@Scope, bir **String argüman** alır ve bu argüman scope’un adını belirtir.

**singleton** veya **prototype** scope’ları için, "singleton" veya "prototype" string değerleri ya da org.springframework.beans.factory.config.ConfigurableBeanFactory sabitleri kullanılabilir.  
@Scope anotasyonu için varsayılan değer **singleton**’dır.

@Profile

org.springframework.beans.factory.annotation.Profile, **bir veya daha fazla belirli profil etkin olduğunda** hangi bileşenlerin (component) kayıt için uygun olduğunu seçmek amacıyla kullanılır.

**Profil (Profile)**, bir **adı olan mantıksal bir grup** olarak düşünülebilir.  
Bir profil, belirli bir amaç için hazırlanmış bean yapılandırmalarını içerir.

Örneğin:

* Farklı **veritabanı sunucuları** için,
* Farklı **test ortamları** için ayrı profiller tanımlanabilir.

Bu sayede, uygulama çalışırken yalnızca ilgili profile ait bileşenler aktif hale gelir.

@Profile anotasyonunun yalnızca **value** adında bir özniteliği vardır ve bu öznitelik **String dizisi (String[])** türündedir.  
Bu, XML yapılandırmasında <beans profile="p1, p2"> şeklinde **profile** özniteliği kullanmakla aynıdır.

@Profile anotasyonu:

* **Sınıf seviyesinde**, @Component ile birlikte kullanılabilir,
* Ayrıca **özel profiller oluşturmak için** de kullanılabilir.

Bu sayede, belirli profiller aktif olduğunda sadece o profile ait bileşenler yüklenir.

@Conditional

org.springframework.context.annotation.Conditional anotasyonu, bir bean’in **yalnızca belirli bir koşul geçerli olduğunda** kaydedilmeye uygun olduğunu belirtmek için kullanılır.  
Yani bean’lerin **koşullu olarak (conditional) kaydedilmesini** sağlar.

Bu, bean kayıt işlemi için adeta bir **if-else** yapısı gibidir.

**Koşul (condition)**, bean tanımı kaydedilmeden önce **programatik olarak belirlenebilen herhangi bir durum** olabilir.  
Örneğin: sistem özellikleri, ortam değişkenleri, özel mantıklar vs.

@Conditional anotasyonu, @Component ve @Configuration ile birlikte kullanılabilir.

@Conditional anotasyonunun bir özniteliği vardır ve bu öznitelik,  
org.springframework.context.annotation.Condition arayüzünü uygulayan sınıflardan oluşan bir **dizi (array)** türündedir.  
Bu sayede birden fazla koşul birlikte kontrol edilebilir.

Bir bean’in kaydedilebilmesi için, belirtilen **tüm koşulların geçerli olması** gerekir.

Condition arayüzü, bir bileşenin (component) kaydedilebilmesi için **sağlanması gereken tek bir koşulu** temsil eder.

Koşullar, bean tanımı kaydedilmeden hemen önce kontrol edilir.

Condition arayüzünün, matches() adında bir metodu vardır.  
Bu metod, ConditionContext ve AnnotatedTypeMetadata parametrelerini alır ve **boolean** (true/false) değer döndürür.

Bean’ler için her türlü koşul (condition) düşünülebilir:

* Bir sistem ya da ortam (environment) özelliğinin (property) mevcut olup olmaması,
* Bean’in belirli bir anotasyona sahip olup olmaması,
* Ve benzeri durumlar.

@Condition vs. @Profile

@Condition, @Profile anotasyonunun daha genel (generic) bir versiyonudur.  
Her ikisi de **if-else** mantığıyla çalışır:  
Profil doğruysa veya koşul sağlanmışsa, ilgili bean’ler kaydedilir.

Profil’ler çoğunlukla **çevresel yapılandırmalar (environmental configurations)** arasından seçim yapmak için kullanılırken,  
koşullar (conditions) **her türlü bean seçimi** için kullanılabilir.

Bu yüzden, profiller koşullara göre daha **genel (coarse-grained) bir seçim stratejisi** olarak düşünülebilir.

XML vs. ANNOTATIONS for CONFIGURATION

XML vs. Annotation Metadata

Yapılandırma (configuration) metadatası için XML dosyaları veya anotasyonlar kullanmanın kendine göre avantajları ve dezavantajları vardır.

* XML dosyaları kolayca büyüyebilir ve bu da karmaşıklığa yol açar; ancak yapılandırma bilgilerini kaynak koddan ayırarak, tüm bean’lerin saf POJO (Plain Old Java Object) olarak kalmasını sağlar.
* Anotasyonlar ise bean’ler hakkında küçük ve bağlama (context) uygun bilgiler sunar, bu da geliştirici dostudur; fakat yapılandırma değişikliklerinde kaynak kodun değiştirilmesini zorunlu kılar.

JAVA-BASED CONFIGURATION

Spring, konteynerin yapılandırılmasını Java kodu ile yapmaya olanak sağlar.  
Java tabanlı yapılandırmanın temel öğeleri **@Bean** ve **@Configuration** anotasyonlarıdır.

Ayrıca Spring, Java’nın standart bağımlılık enjeksiyon mekanizmalarını da destekler:

* @Resource
* @Inject

@Bean

org.springframework.context.annotation.Bean anotasyonu, **metotlar için** kullanılır.  
@Bean anotasyonu, bir metodu Spring konteyneri tarafından yönetilecek bir bean üreten **fabrika metodu** haline getirir.

@Bean ile oluşturulan bean örneklerinin sınıflarını @Component ile işaretlemeye gerek yoktur.

@Bean, XML dosyasındaki <bean> etiketi ile aynı işlevselliği sağlar.

@Bean anotasyonu ile işaretlenmiş metotlar, herhangi bir Spring @Component içinde tanımlanabilir; bu durumda bir bean nesnesi başka bir bean nesnesi üretmiş olur.

Ancak, @Bean metotları en sık **@Configuration anotasyonlu** sınıflarda kullanılır.  
Çünkü, fabrika metotlarını ayrı yapılandırma sınıflarında toplamak ve bu sınıfları @Configuration ile işaretlemek en iyi uygulamadır.

@Configuration

org.springframework.context.annotation.Configuration anotasyonu, **sınıflar için** kullanılır.  
@Configuration, bir sınıfın **@Bean metotları tanımladığını** belirtir.

Spring konteyneri, bu anotasyonlu sınıfı işleyerek çalışma zamanında (runtime) bean tanımlarını oluşturur ve bu bean’lere yönelik servis isteklerini yönetir.

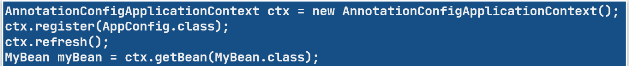
@Configuration sınıfları genellikle,  
AnnotationConfigApplicationContext veya web destekli versiyonu olan  
AnnotationConfigWebApplicationContext ile başlatılır (bootstrap edilir).

@Configuration anotasyonunun iki özniteliği vardır:

* value: Bir **String** türündedir ve yapılandırmanın (configuration) adını temsil eder.
* proxyBeanMethods: Bir **boolean** türündedir ve varsayılan değeri **true**’dur.

Bootstraping @Configuration Classes

Yapılandırma sınıflarını kaydetmek için birkaç yöntem vardır:  
AnnotationConfigApplicationContext, anotasyonları kullanan bağımsız (stand-alone) bir konteyneri başlatmak (bootstrap etmek) için kullanılır.



XML yapılandırma dosyası, yapılandırma sınıflarını kaydetmek için kullanılabilir.



Bileşen taraması (component scan) yoluyla da yapılandırma sınıfları kaydedilebilir.

AnnotationConfigApplicationContext

org.springframework.context.annotation.AnnotationConfigApplicationContext, hem **BeanFactory** hem de **ApplicationContext** arayüzlerini (interface) uygulayan (implements) bir sınıftır.  
Bu sınıf, @Component ile işaretlenmiş bileşen sınıfları ve @Configuration anotasyonu ile işaretlenmiş sınıflar tarafından üretilen diğer bean’leri kabul eder.

AnnotationConfigApplicationContext, **bileşen (component) sınıflarını ve bean’leri kaydedebilir**.  
Ayrıca, **paketleri tarayabilir (scan edebilir)**.  
Bu işlemleri gerçekleştirmek için çeşitli **yapıcı metodlara (constructors)** ve **metotlara (methods)** sahiptir.

@ComponentScan

org.springframework.context.annotation.ComponentScan, tüm @Component sınıflarıyla birlikte kullanılmak üzere **bileşen tarama yönergesi (component scanning directive)** sağlayan bir anotasyondur.

@Configuration, bir tür @Component olduğu için **sınıf yolunda (classpath) taramaya** dahildir.

@ComponentScan anotasyonunun, **taranacak paketleri** belirtmek için kullanılan öznitelikleri vardır.  
Varsayılan olarak, **@ComponentScan anotasyonunun bulunduğu paketten itibaren** taramaya başlar.

@Import

org.springframework.context.annotation.Import, bir veya daha fazla bileşen sınıfını (**component class**) içe aktarmak (**import etmek**) için kullanılan bir anotasyondur.

@Bean and @Configuration

@Bean anotasyonunun **profile**, **scope**, **lazy** veya **depends-on**, **primary** gibi herhangi bir özniteliği (attribute) yoktur.  
Gerekli etkileri elde etmek için @Scope, @Lazy, @DependsOn ve @Primary anotasyonları, @Bean ile birlikte kullanılmalıdır.

Eğer @Lazy anotasyonu @Configuration üzerinde kullanılırsa, @Bean metotlarıyla oluşturulan tüm bean’ler **tembel (lazy) olarak başlatılır**.

@Profile

@Configuration ile işaretlenmiş bir sınıf aynı zamanda @Profile anotasyonuna da sahip olabilir.  
Bu durumda, sınıftaki tüm @Bean metotları ve @Import edilen bileşenler, belirtilen profil(ler) ile ilişkilendirilir.

LITE @BEAN

Fabrika (@Bean) metotları, @Component anotasyonlu bir sınıfta veya normal (düz) bir sınıfta tanımlanabilir; bu durumda bunlara **lite @Bean metotları** denir.

@Configuration anotasyonlu sınıflardaki @Bean metotları, bean’ler arası bağımlılıkları yönetmek için bean üretebilirken, **lite @Bean metotları bean’ler arası bağımlılıkları bildiremezler**; fonksiyonellikleri yalnızca kendi sınıflarıyla sınırlıdır ve sadece o sınıfın durumuna (state) gerekli bean ve değerleri üretirler.

Bu durum, **lite mod** ve **full mod** olarak adlandırılır.

Lite mod, bir bileşenin (component) kendi bağımlılıklarının fabrikası (factory) olması amacıyla kullanılabilir.

Eğer @Component anotasyonuna sahip bir sınıfın kendi @Bean metotları varsa, yani **lite mod** kullanılıyorsa, enjeksiyonlar için kendi @Bean metotları çağrılır.

Eğer enjeksiyon **yapıcı (constructor) içine** yapılacaksa, bu @Bean metotlarının **static** olması gerekir; çünkü nesnenin kendisi henüz oluşturulmamıştır.

Alan (field) ve özellik (property) enjeksiyonları için, @Bean ile işaretlenmiş **örnek (instance) metotlar** kullanılabilir.

JAVA’S DI MECHANISMS

@Resource

javax.annotation.Resource anotasyonu, **alanlar (fields)** ve **özellik setter metotları** üzerinde bağımlılık enjeksiyonu için kullanılır.  
@Resource anotasyonu birkaç özniteliğe sahiptir; bunlardan biri de **name** özniteliğidir.

Spring, name özniteliğinin değerini, enjekte edilecek bean’in adı olarak alır.  
Eğer name belirtilmemişse, varsayılan isim, alan adı veya setter metotla işaretlenmişse özellik adı baz alınarak türetilir.

Spring’in tüm niteleme (qualification) mekanizmaları, @Resource anotasyonu ile sorunsuz bir şekilde çalışır.

Spring ile birlikte @Resource kullanmanın başlıca kullanım durumu,  
**daha önce @Resource anotasyonunu kullanmış olan bir Java kod parçasını**,  
**Spring kullanılan yeni bir projede yeniden kullanma** ihtiyacıdır.

Bu durum dışında, Spring kullanan bir projede @Resource kullanmaya genellikle gerek yoktur.

@Inject

(type lar arasında)

@Inject, @Autowired yerine kullanılabilir.  
Herhangi bir özniteliği yoktur ve **alan (field)**, **yapıcı (constructor)** ve **metot** seviyesinde kullanılabilir.

@Inject, **POJO** (Plain Old Java Object) olan herhangi bir Java nesnesini enjekte eder.  
Enjekte edilecek POJO’ların ayrıca işaretlenmesine gerek yoktur.

Bağımlılık enjeksiyonunun (@Inject) çalışabilmesi için gereken tek yapılandırma, **paketlerin kök dizininde yer alan META-INF klasörü içinde bir beans.xml dosyası** bulundurulmasıdır.

@Named

@Named, @Component ile **aynı işlevselliği sağlar**.  
**value** adında bir özniteliğe sahiptir ve bu, **string tabanlı bir qualifier (niteleyici)** belirtmek için kullanılır.

Aynı zamanda, enjeksiyon için bean’leri nitelemek amacıyla da kullanılır.  
Bu kullanımda, Spring’in @Qualifier anotasyonuyla **aynı işlevselliğe** sahiptir.

Hatta, özel qualifier’lar oluşturmak için javax.inject.Qualifier anotasyonu da kullanılabilir.

@Named = @Component, ancak @Named daha **genel**, @Component ise **Spring'e özgü ve tercih edilen** çözümdür.

BEAN LIFECYCLE

Eager and Lazy Loading

Bildiğimiz gibi, Spring varsayılan olarak **singleton bean’leri erken (eagerly) yükler**.  
**Prototype** bean’ler ise, yalnızca context’ten talep edildiklerinde yüklenir.

Tüm bean’lerin **tembel (lazy) olarak yüklenmesini** belirtmek için, XML dosyasında <beans> etiketine  
default-lazy-init="true" özniteliği eklenebilir.

Ayrıca, sadece belirli bean’ler için bu davranışı sağlamak istenirse, her <bean> etiketi için ayrı ayrı  
lazy-init="true" özniteliği kullanılabilir.

Başlatma (bootstrap) sırasında, Spring konteyneri verilen tüm XML dosyalarını tarar ve **erken (eager) olarak yüklenmesi belirtilen** tüm bean örneklerini oluşturur.

Bir bean **tembel (lazy)** olarak tanımlanmışsa, context’ten istenene kadar oluşturulmaz — **erken yüklenecek şekilde tanımlansa bile** bu geçerlidir.

Bir bean’in başka bir bean içine enjekte edilmesi de, **enjekte edilen bean’in oluşturulmasına** neden olur.

Tüm singleton bean’lerin **erken yüklenmesi** uygulamanın başlatılmasının biraz zaman almasına yol açabilir;  
ancak uygulama başlatıldığında, tüm bean örnekleri **kullanıma hazır** durumda olur.

@Lazy

Spring’in bean’leri **erken (eager) yükleme** varsayılan davranışı, hem **@Component ile yapılan anotasyon tabanlı tanımlamalar** hem de **@Bean ve @Configuration ile yapılan Java tabanlı yapılandırmalar** için geçerlidir.

Bu iki yapılandırma türünde de, bean’lerin **tembel (lazy) yüklenmesini** belirtmek için @Lazy anotasyonu kullanılır.

org.springframework.context.annotation.Lazy, **hem sınıf seviyesinde** hem de **metot seviyesinde** kullanılabilen bir anotasyondur.  
@Lazy anotasyonunun **value** adında bir özniteliği vardır ve bu özniteliğin varsayılan değeri **true**’dur.

@Lazy, @Component ve @Bean ile birlikte kullanıldığında, ilgili bean’in **tembel (lazy) olarak yüklenmesi** gerektiğini belirtir.

@Configuration ile birlikte kullanıldığında ise, **o sınıftaki tüm @Bean metotlarıyla oluşturulacak bean’lerin tembel yüklenmesini** sağlar.

Ancak, bir @Bean metoduna @Lazy ile value = false verilerek, @Configuration seviyesindeki tembel yükleme davranışı **geçersiz kılınabilir**.

Fakat @Lazy(value = false) kullanımı **çok nadirdir**.

@Lazy, **bağımlılık enjeksiyonu noktalarında** @Autowired ve @Inject anotasyonlarıyla birlikte de kullanılabilir.

@Lazy, **bağımlılık enjeksiyonu noktalarında**, @Autowired ve @Inject anotasyonlarıyla birlikte de kullanılabilir.  
Bu durumda, **gerçek nesne yerine bir proxy (vekil) nesnesi** oluşturulur ve enjekte edilir.

Gerçek nesnenin oluşturulması, **yalnızca sarmalayan (enclosing) nesne tarafından erişildiğinde** gerçekleşir.

Bu mekanizma, **GoF (Gang of Four)** tasarım desenlerinden biri olan **proxy (vekil) deseni** olarak bilinir.

ORDERING BEANS

Order for Initialization and Injection

Bean'ler arasında başlıca iki tür sıra (order) vardır:

1. **Başlatma sırası (initialization order)** – genellikle bean’ler arasındaki **bağımlılık ilişkilerine** göre belirlenir.
2. **Enjeksiyon sırası (injection order)** – bağımlılıkların hangi sırayla enjekte edileceğini belirler.

**Başlatma sırası**, @DependsOn anotasyonu ile kontrol edilebilir.  
**Enjeksiyon sırası** ise, Spring içinde farklı yollarla kontrol edilebilir.

ORDERING BEANS: INITIALIZATION ORDER AND @DEPENDSON

@DependsOn

org.springframework.beans.factory.annotation.DependsOn, bir bean’in başka bir bean’e **bağımlı olduğunu** belirtmek için kullanılır.  
@DependsOn, bean’lerin **başlatılma sırasını (initialization order)** etkilemek amacıyla kullanılır.

@DependsOn, **String dizisi (String[])** türünde bir argüman alır ve bu dizide, bağımlı olunan bean’lerin **isimleri** yer alır.

Bir bean, @DependsOn ile başka bir bean’e bağımlı olduğunu belirtirse, Spring’in IoC (Inversion of Control) mekanizması,  
**önce bağımlı olunan bean’i**, ardından @DependsOn ile işaretlenmiş olan bean’i başlatacağını garanti eder.

@DependsOn, **singleton bean’lerin yok edilme (destruction) sırasını** da etkiler.  
Eğer bir bean, başka bir bean’e bağımlı olduğunu @DependsOn ile belirtirse,  
**önce bağımlılığı bildiren bean yok edilir**, ardından bağımlı olunan (depend edilen) bean yok edilir.

Eğer **bileşen taraması (component-scanning)** kullanılmıyorsa, @DependsOn anotasyonu **sınıf seviyesinde herhangi bir etki oluşturmaz**.

**Bağımlılık ilişkisi (depends-on)**, XML yapılandırma dosyasında da tanımlanabilir; bu durumda <bean> etiketinin depends-on özniteliği kullanılır.

Eğer bir bean için **hem XML’de depends-on özniteliği**, hem de **@DependsOn anotasyonu** tanımlanmışsa,  
**@DependsOn anotasyonu yok sayılır**, ve **XML’deki tanım geçerli olur**.

Dikkatli olun: **Döngüsel bağımlılıklar (circular dependencies)** oluşturmamaya özen gösterin.

Örneğin:

* Bean A, Bean B’ye bağımlı
* Bean B, Bean A’ya bağımlı → ❌ **Çıkmaz**

Ordering Beans: Injection Order

Order of Injection

**Spring**’de enjeksiyon sırası (**injection order**) çeşitli mekanizmalarla kontrol edilebilir:

* @Order anotasyonu
* @Priority anotasyonu
* Ordered arayüzü (interface)

Ancak, **enjeksiyon sırası**, **başlatma sırasını (initialization order)** etkilemez.  
Başlatma sırası, esas olarak bean’ler arasındaki **bağımlılık ilişkileri** ve @DependsOn anotasyonu ile belirlenir

@Order.

org.springframework.core.annotation.Order, **bean’lerin enjeksiyon sırasını kontrol etmek** için kullanılan bir anotasyondur.  
@Order, örneğin bir liste içinde bean’leri belirli bir sıraya göre **sıralamak** amacıyla kullanılır.

@Order anotasyonunun yalnızca bir özniteliği vardır:

* value: **int** türündedir.

Varsayılan değeri: org.springframework.core.Ordered.LOWEST\_PRECEDENCE (en düşük öncelik).  
Yani, **@Order değeri küçük olan bean’ler, büyük olanlardan önce enjekte edilir**.

**Aynı @Order değerine sahip bean’ler**, **rastgele bir sırayla (arbitrary order)** enjekte edilir.  
Bu sıralamadan **org.springframework.core.OrderComparator** sınıfı sorumludur.

Ordered Interface

org.springframework.core.Ordered, **bean’lerin enjeksiyon sırasını kontrol etmek** için kullanılan bir arayüzdür (interface).  
Bu arayüzün yalnızca bir metodu vardır:

* getOrder(): Enjeksiyon sırasını belirleyen **tamsayı (int)** değeri döndürür.

Bu arayüzle yapılan sıralama, @Order anotasyonu ile yapılan sıralama ile **aynıdır**.  
**Daha düşük değer döndüren bean’ler**, **daha yüksek değer döndürenlerden önce** enjekte edilir.

Bean Lifecycle

What is Lifecycle?

**Yaşam döngüsü (lifecycle)**, bir nesnenin yaşamı boyunca geçtiği evreleri ifade eder.  
Bean’lerin yaşam döngüsü sırasında meydana gelen birçok önemli dönüm noktası vardır, bunlara **olaylar (events)** denir.

**Yaşam döngüsünün ve olayların ne anlama geldiği**, içinde bulunulan **ortama** (environment) göre değişir.

Örneğin Spring’de;

* Bean örneğinin oluşturulması,
* Tüm bağımlılıklarıyla birlikte başlatılması (initialization),
* IoC konteyneri kapandığında yok edilmesi (destruction)  
  gibi olaylar, bean’in yaşam döngüsündeki adımlardır.

Ancak örneğin JPA’da, **farklı olaylar** ve dolayısıyla **farklı bir yaşam döngüsü modeli** vardır.

**Yönetilmeyen (non-managed) ortamlarda**, bu tür olayların tamamı **programcı tarafından oluşturulur ve yönetilir**.

Ancak **yönetilen (managed) ortamlarda**, yani konteynerlerde, **yaşam döngüsü konteyner tarafından yönetilir** ve ilgili olaylar **otomatik olarak tetiklenir**.

Java EE'nin **servlet** ve **EJB konteynerleri** ile **Spring IoC konteyneri** gibi konteynerler,  
kendi nesneleri (servlet, EJB veya Spring bean’leri gibi) için farklı **yaşam döngüsü modellerine** sahiptir.

Bu konteynerler, nesnelerinin veya bean’lerinin yaşam döngüsünü yönetir ve  
bu olaylara yönelik **bildirimler (notification)** gibi kolaylıklar sağlar.

Events and Callback Methods

**Yönetilen ortamlarda**, yaşam döngüsü konteyner tarafından yönetilen bir nesneyle ilgili **önemli bir olay gerçekleştiğinde**,  
konteyner bir **olay (event)** oluşturur ve **bu olayla ilgilenebilecek nesnelere bildirim (notification)** gönderir.

Bu tür olaylara dair bildirimleri alan metotlara **geri çağırım metotları (callback methods)** denir.  
Olayı temsil eden nesneler (event object), bu callback metotlarına **parametre olarak iletilir**.

Bu metotlar genellikle **belirli bir arayüze (interface)** ait metotlardır;  
**konteyner bu metotları çağırır** ve **olay nesnelerini iletir**.

Ve elbette, **bu arayüzlerin (interface) nesneleri**, bu tür **bildirimleri (yani olayları/event’leri)** alabilmek için **kendi kendilerini bu olaylara kaydetmelidir (register etmelidir)**.

Bu mekanizma, genellikle **GoF (Gang of Four)** tarafından tanımlanan **Observer (Gözlemci) tasarım deseni** gibi **tasarım kalıpları (design patterns)** kullanılarak uygulanır.

Bean Initialization

Hatırlayabileceğiniz gibi, **bir bean’in başlatılması (initialization)**, **bean örneğini oluşturmak için yapıcı (constructor) çağrısıyla** başlar.

Eğer **bağımlılık enjeksiyonu içeren bir yapıcı metod** varsa, **yalnızca bir tane olmalıdır** ve bu metod, bean örneğini oluşturmak için çağrılır.  
Elbette bu durumda, **yapıcıda tanımlı tüm bağımlılıkların karşılanmış (satisfied) olması gerekir**.

Eğer enjeksiyon içeren bir yapıcı yoksa, **varsayılan (parametresiz) yapıcı metod** olmalıdır ve bean bu metodla oluşturulur.  
Bu durumda bağımlılıklar, **alanlar (fields)** veya **özellikler (properties)** üzerinden enjekte edilebilir.

Örnek (instance) oluşturulduktan sonra, **IoC konteyneri**, **örnek değişkenlerinde (instance variables)** ve **özelliklerde (properties)** karşılanması gereken **bağımlılıkları (dependencies)** arar.

Önce, **örnek değişkenlerindeki (alanlardaki / fields)** bağımlılıklar karşılanır.  
Ardından, **metotlardaki** bağımlılıklar karşılanır.

**Field** ve **metot** bağımlılıklarının hangi sırayla karşılanacağı, **fiziksel tanımlanma sırasına (class içindeki yazım sırasına)** göre belirlenir.

**XML tabanlı yapılandırmada**, yalnızca **özellik (property)** ve **yapıcı (constructor)** enjeksiyonları mümkündür.  
**Alanlar (field)** ve **yapılandırma metotları (config methods)** için enjeksiyon tanımı yapılamaz.

Ancak, **anotasyon tabanlı yapılandırmada**, @Autowired anotasyonu ile **alanlarda**, **yapıcılarda**, **özellik setter metotlarında** ve **yapılandırma metotlarında** enjeksiyon yapılabilir.

Lifecycle Events and Notifications

Post-Initialization & Pre-Destruction

Bean’ler için iki ana **bildirim (notification)** türü vardır:

1. **Başlatma sonrası (post-initialization)**
2. **Yok edilmeden önce (pre-destruction veya pre-destroy)**

Bir bean’in **başlatılması (initialization)**, onun örneğinin oluşturulmasını ve **tüm bağımlılıklarının enjekte edilmesini** kapsar.  
Bu işlem tamamlandığında, bean örneği artık **hizmet sağlamaya hazır** hale gelir.

**Yok edilme (destruction)** ise, bean örneğinin **tamamen ortadan kaldırılmasıdır** — artık **hizmet veremez** duruma gelir.

Post-Initialization

**Tüm bağımlılıkları enjekte edildikten hemen sonra** çağrılan **başlatma sonrası (post-initialization)** bildirimi,  
**başlatılmış bean örneği hazır olduğunda özel işlemler (custom actions)** gerçekleştirmek için kullanılabilir.

Bu özel işlemler **yapıcı (constructor)** içinde yapılamaz; çünkü bazı bağımlılıklar **alanlar (instance variables)** ve **özellikler (properties)** olabilir  
ve **tam başlatma (full initialization)** için bunların da enjekte edilmiş olması gerekir.

Bu özel işlemler;

* **diğer bean’lere hazır olduğu bilgisini iletmek**,
* **ileride yapılacak işlemler için ortam hazırlamak** gibi görevler olabilir.

**Pre-Destruction**

**Konteyner tarafından yok edilmeden hemen önce** çağrılan **yok edilme öncesi (pre-destruction)** bildirimi,  
bean’in sahip olabileceği **kaynakları temizlemek (clean up)** için kullanılabilir.

Bean’ler, **JVM dışındaki** diğer nesnelere veya kaynaklara referanslar tutabilir.  
Özellikle **dosya gibi JVM’in doğrudan kontrolü dışında olan kaynaklar** söz konusuysa,  
bean’in elindeki tüm kaynakları serbest bırakması (release etmesi) **faydalı ve gereklidir**.

Post-Initialization & Pre-Destruction

Eğer bean’ler, **erken (eager) yükleniyorsa** — ki bu, **singleton** bean’ler için varsayılan davranıştır —  
bean’ler context’ten **talep edilmese bile başlatılır** ve bu sırada **başlatma sonrası (post-initialization)** olayları tetiklenir.

Ancak, **tembel (lazy) yükleme** durumunda, bean’ler yalnızca **context’ten istendiği zaman başlatılır**  
ve **yalnızca o anda post-initialization olayları gerçekleşir**.

**Post-initialization (başlatma sonrası)** olayı, **bean’in kapsamı (scope)** ne olursa olsun çağrılır.  
Ancak, **pre-destruction (yok edilme öncesi)** olayı **yalnızca singleton bean’ler** için çağrılır.  
**Konteyner, prototype bean’ler için bu olayı tetiklemez.**

**Prototype** bean’lerin **pre-destruction** olaylarını alabilmesi için,  
**DestructionAwareBeanPostProcessor** gibi özel bir **post-processor** kullanılmalıdır.

Lifecycle Events and Notifications: Lifecycle of ApplicationContext

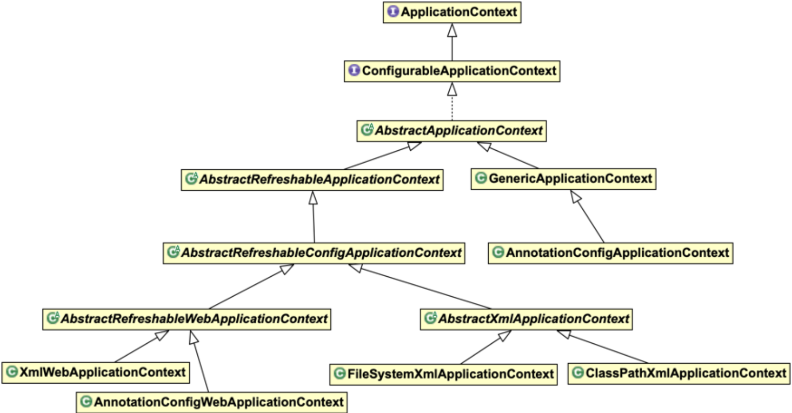
Lifecycle of ApplicationContext

**ApplicationContext**, Spring'de bean yapılandırması için kullanılan **ana nesnedir**.

ConfigurableApplicationContext, ApplicationContext arayüzünün bir **alt arayüzüdür**  
ve buna **yapılandırma ve yaşam döngüsü metodları** ekler.

Başlıca üç yaşam döngüsü metodu şunlardır:

1. **refresh()** – Uygulama bağlamını (context) yeniler, tüm bean’leri yeniden oluşturur ve başlatır.
2. **close()** – Bağlamı kapatır ve tüm singleton bean’leri yok eder.
3. **registerShutdownHook()** – JVM kapatıldığında bağlamın otomatik olarak kapatılmasını sağlar.



AbstractApplicationContext

AbstractApplicationContext, tüm context nesneleri için **soyut (abstract) taban sınıftır**  
ve tüm **yaşam döngüsü metodlarını** uygular.

* **refresh()**: Kalıcı konfigürasyonun temsili olan yapılandırmayı **yükler veya yeniler**.
* **close()**: **Context’i kapatır**, kullanılan tüm **kaynakları** ve **kilitleri (locks)** serbest bırakır.

registerShutdownHook(), JVM çalışma zamanı (java.lang.Runtime) ile birlikte  
**SpringContextShutdownHook adında bir kapanış kancası (shutdown hook)** kaydeder.

Bu hook, **JVM kapanırken context’i kapatır** — **tabii daha önce kapatılmadıysa**.

Runtime sınıfı, addShutdownHook(Thread hook) adında bir metoda sahiptir.  
Bu metod, uygulamaların **bir Thread nesnesini kayıt ettirmesine** olanak tanır;  
böylece JVM kendini kapatmadan hemen önce, **java.lang.Shutdown** sınıfı tarafından bu thread çalıştırılır.

Ayrıca, close() metodu çağrıldığında, **kayıtlı olan herhangi bir shutdown hook silinir**.

AbstractApplicationContext, **Template (Şablon) tasarım deseni**ni kullanır.  
Bean factory’nin yaşam döngüsünü yöneten **ana algoritmayı kendisi belirler**,  
ancak bu sürecin bazı bölümlerinin uygulanmasını **alt sınıflara bırakır**.

Bu amaçla tanımlanan üç soyut (abstract) metot vardır ve bunların uygulanması **somut alt sınıflara aittir**.

Bu metotlar, bean factory’yi yöneten metotlardır:

1. **ConfigurableListableBeanFactory getBeanFactory()**
2. **void closeBeanFactory()**
3. **void refreshBeanFactory()**

AbstractApplicationContext, **içsel bir bean factory’ye sahip olsa da**,  
**bean factory’nin nasıl oluşturulacağı, kapatılacağı, yenileneceği** ve  
**bean tanımlarının nerede saklanacağı** gibi konuları **alt sınıflara bırakır**.

Buna karşılık, **tüm olay (event) ve bildirim (notification) mekanizmalarını kendisi yönetir**.  
Bunu, context içinde tanımlı olan aşağıdaki bean’leri **otomatik olarak kaydederek (register ederek)** yapar:

* BeanFactoryPostProcessor
* BeanPostProcessor
* ApplicationListener

AbstractRefreshableApplicationContext

AbstractRefreshableApplicationContext, **birden fazla refresh() çağrısına izin veren soyut taban sınıftır**.

Her refresh() çağrısında, **yeni bir bean factory örneği oluşturur**.

Genellikle, **yeni yapılandırma kaynaklarından (configuration locations)** yeni bean tanımlarını yüklemek için  
yeni bir refresh() çağrısı gerekir.

Bu sınıfın yalnızca bir adet soyut metodu vardır ve bu metod, **bean tanımlarının yüklenmesinden sorumludur**:

* loadBeanDefinitions(DefaultListableBeanFactory beanFactory)

Lifecycle Events and Notifications: Lifecycle Methods – refresh()

refresh() metodu, **context nesneleri için bir başlatma (startup) metodudur**.

**Yenilenebilir (refreshable) context'ler** için taban sınıf,  
**AbstractRefreshableApplicationContext** sınıfıdır.

Bu sınıfın refresh() metodunun çağrılması, **yeni bir içsel (internal) bean factory örneği oluşturur**.

Ayrıca, bu metod çağrıldığında, yapılandırmanın kalıcı temsili (persistent representation)  
**yüklenir veya yenilenir**.  
Bu yapılandırma:

* Java tabanlı bir konfigürasyon,
* XML dosyası,
* Properties dosyası,
* İlişkisel veritabanı şeması  
  veya
* başka bir formatta olabilir.

Yani, bir **ApplicationContext nesnesi her başlatıldığında (initialize) veya yenilendiğinde (refresh)**:

* **Tüm bean’ler yüklenir**,
* **Post-processor bean’ler tespit edilir ve etkinleştirilir**,
* **Singleton bean’ler önceden oluşturulur (pre-instantiated)**  
  ve
* ApplicationContext nesnesi **kullanıma hazır hale gelir**.

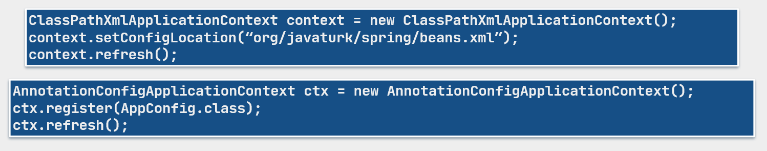
Bir context **yenilendiğinde (refresh)**, onunla ilişkili **callback (geri çağırım) metodları da çağrılır**.  
Ayrıca, başlatılmış tüm bean’ler için tanımlı olan **tüm post-initialization (başlatma sonrası) callback metodları çalıştırılır**.

Eğer konfigürasyon, ApplicationContext nesnelerinin **yapıcı (constructor)** metodlarına geçirilirse,  
bu nesneler, verilen konfigürasyondaki **tüm bean’leri tarar** ve  
**erken (eager) yüklenen tüm singleton bean’leri başlatır**, böylece bunlar hizmet vermeye hazır olur.

ApplicationContext’in alt sınıfları olan örneğin  
FileSystemXmlApplicationContext ve ClassPathXmlApplicationContext,  
bean tanımlarını yüklediklerinde **yapıcılarında (constructor) kendilerini otomatik olarak refresh() ederler**.

Ancak, **tembel (lazy) yüklenen bean’ler yalnızca refresh() işlemi tamamlandıktan sonra**,  
**istek üzerine başlatılır ve hizmet verir**.

Eğer context nesnesi, konfigürasyonu **yapıcısı (constructor)** aracılığıyla almazsa,  
yeni bir konfigürasyon nesnesi verildiğinde,  
**tüm bean’lerin hazır hale gelmesi için refresh() çağrılması gerekir**.



ApplicationContext’in bazı alt sınıfları — aslında AbstractRefreshableApplicationContext’in tüm alt sınıfları,  
örneğin FileSystemXmlApplicationContext ve ClassPathXmlApplicationContext —  
birden fazla refresh() çağrısına izin verir.

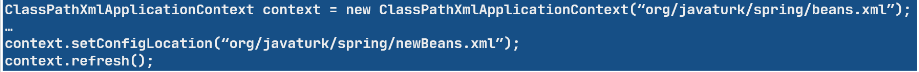
Spring bu duruma **“hot refresh”** (sıcak yenileme) der.

refresh() metodu çağrıldığında, yeni bir içsel bean factory örneği oluşturulur ve  
muhtemelen yeni bir konfigürasyon yüklenir; böylece önceki bean factory’ye kayıtlı tüm bean’ler yok edilir (destroyed).

Ancak, **refresh() metodunun tekrar çağrılma ihtiyacı** genellikle **konfigürasyonun değişmesi ve yeni konfigürasyondaki tüm bean’lerin yeniden yüklenmesi** içindir.  
Aksi takdirde, yeni konfigürasyonun herhangi bir etkisi olmaz.

Bu özelliği destekleyen uygulama sınıfı, **birden fazla refresh() çağrısını desteklemelidir**.

FileSystemXmlApplicationContext ve ClassPathXmlApplicationContext bu **birden çok yenilemeyi (multiple refreshes)** destekler.



AnnotationConfigApplicationContext gibi tüm GenericApplicationContext sınıfları,  
birden fazla refresh() çağrısını desteklemez; yalnızca **tek bir refresh() çağrısına izin verirler**.

AnnotationConfigApplicationContext

AnnotationConfigApplicationContext, **anotasyon tabanlı yapılandırmayı destekleyen bağımsız (standalone) bir application context’dir**.

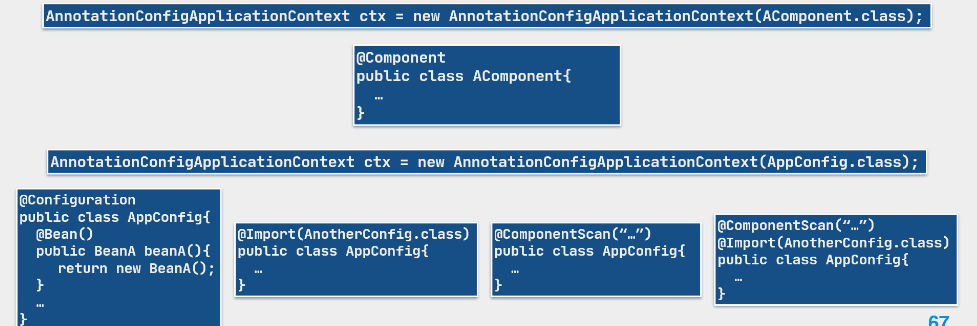
Yapıcı metoduna (constructor) girdi olarak,

* @Component,
* @Configuration anotasyonlarına sahip component sınıflarını ve
* javax.inject anotasyonlarını kullanan JSR-330 uyumlu sınıfları kabul eder.

Çoğu zaman, yapıcı metoduna,

* @ComponentScan anotasyonuna sahip bir sınıf verilir; bu sınıf, @Component ile işaretlenmiş sınıfları taramak için kullanılır.  
  Ya da
* @Import ile @Configuration anotasyonlu sınıflara işaret eder.

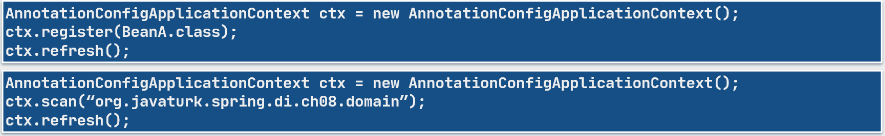
AnnotationConfigApplicationContext, yapıcısında (constructor) ayrıca otomatik olarak bir refresh() çağrısı yapar.



Eğer birden fazla @Configuration sınıfı varsa,  
daha sonraki sınıflardaki @Bean metodları,  
önceki sınıflardakilerin üzerine yazacaktır (override edecektir).

AnnotationConfigApplicationContext, sınıfları kaydetmek için iki yöntem sunar:

* register(Class...) yöntemi ile **tek tek sınıfları kayıt edebilir**,
* scan(String...) yöntemi ile **sınıf yolunu (classpath) tarayabilir**.



**Varsayılan yapıcı metodu (default constructor)** refresh() çağrısı yapmadığı için,  
**yeni yapılandırmanın yüklenip hazır hale gelmesi**,  
**manuel olarak yapılacak bir refresh() çağrısına bağlıdır**.

AnnotationConfigApplicationContext, GenericApplicationContext’in bir alt sınıfı olarak,  
**birden fazla refresh() çağrısını desteklemez; yaşam süresi boyunca sadece bir kez refresh() yapılmasına izin verir**.

Bu yüzden, yapılandırma yükleniyorsa,  
refresh() çağrısı ya **yapıcıda (constructor) yapılmalı** ya da  
yeni yapılandırma **register()** veya **scan()** ile yüklendikten sonra yapılmalıdır.

Bir kez refresh() çağrıldıktan sonra,  
sonraki register() veya scan() çağrılarından sonra **yeni bir refresh() çağrısına gerek yoktur**,  
çünkü yeni yapılandırma otomatik olarak yüklenip hazır hale getirilir.

Bunun bir istisnası vardır:  
Eğer @Configuration ile işaretlenmiş bir sınıf register() veya scan() ile kaydedilirse,  
yapılandırmanın etkili olması için **her zaman yeni bir refresh() çağrısı yapılması gerekir**.

Eğer aynı bean, @Component ile işaretlenmiş ve yeni bir konfigürasyonda tekrar kayıt edilirse,  
**daha sonraki kayıtlar, önceki kayıtların üzerine yazar**.

Aynı durum @Configuration sınıfları için de geçerlidir;  
daha sonraki sınıflardaki @Bean metodları, önceki sınıflardakilerin üzerine yazacaktır.

register() veya scan() metodlarını kullanmak,  
**yapılandırmayı çalışma zamanında (run-time) programlı şekilde oluşturmak için faydalıdır**.

Lifecycle Events and Notifications: Lifecycle Methods – close()

Close()

close() metodu, **application context’i kapatır**.  
Ayrıca, uygulamanın elinde tutabileceği tüm **kaynakları** ve **kilitleri (locks)** serbest bırakır.  
Bu, **önbelleğe alınmış (cached) tüm singleton bean’lerin yok edilmesini (destroy) de kapsar**.

Bir context kapatıldığında,  
**singleton bean’ler üzerinde pre-destruction (yok edilme öncesi) callback metodları da çağrılır**.

Context kapatıldıktan sonra,  
**yaşam döngüsünün sonuna ulaşır** ve artık **yenilenemez (refresh), yeniden başlatılamaz (restart) veya herhangi bir bean’i sunamaz**.

close() metodu **birden fazla kez çağrılsa da yan etkisi olmaz**;  
zaten kapalı olan bir context’e yapılan sonraki close() çağrıları **yoksayılır (ignore edilir)**.

IoC konteyneri, yapılandırma yenilendikten (refresh edildikten) sonra tüm bean’lerde post-construct (başlatma sonrası) bildirimleri çağırır,  
ancak **prototype scope** olan bean’lerde pre-destroy (yok edilme öncesi) bildirimleri otomatik olarak çağrılmaz.

Context kapatıldığında (close edildiğinde),  
**tüm bean’ler (singleton veya prototype fark etmeksizin) yok edilir**,  
böylece context artık bu bean’leri sunamaz.

Farklı olan ise, lifecycle (yaşam döngüsü) yok edilme metodlarının prototype bean’lerde çağrılmasıdır;  
prototype bean’ler yok edilir fakat pre-destroy bildirimlerini **otomatik olarak almazlar**.

Kullanıcı (client), **prototype scope**’lu bean’leri temizlemek (clean up) ve bu bean’lerin tutabileceği pahalı kaynakları serbest bırakmak zorundadır.  
Prototype bean’ler, DestructionAwareBeanPostProcessor gibi özel bir post-processor kullanılarak  
**pre-destruction (yok edilme öncesi) olaylarını alabilirler**.

Spring şöyle der:  
Bazı açılardan, Spring konteynerinin prototype scope’lu bean’ler için rolü, Java’nın new operatörünün yerini almaktır.  
Bundan sonraki tüm yaşam döngüsü yönetimi (lifecycle management) **kullanıcı tarafından yapılmalıdır**.

Removing Beans from Context

Bean’ler, org.springframework.beans.factory.support.BeanDefinitionRegistry üzerindeki removeBeanDefinition() metodu çağrılarak **manuel olarak context’ten kaldırılabilir**.

Bean örneği (instance) context’in BeanDefinitionRegistry’sinden manuel olarak kaldırıldığında,  
bu bean **artık context üzerinden erişilebilir olmaz**.

Destroying Beans

Bir ApplicationContext nesnesi **kapatıldığında**, onun oluşturduğu **tüm bean’ler JVM’in belleğinde başlatılmış (initialized) halde kalmaya devam eder**,  
ancak context artık **hiçbir nesneye hizmet veremez** veya **olay (event) fırlatamaz**.

Bir ApplicationContext nesnesi, **yaşam süresi içinde refresh() edildiğinde**,  
önceden oluşturduğu tüm bean’ler yine JVM belleğinde başlatılmış olarak kalır,  
ancak **context artık bu bean’leri içermez ve yönetmez**.

refresh() işleminden sonra, ApplicationContext nesnesi  
**yeni yapılandırmayla kaydedilen yeni bean’lere hizmet vermeye başlar**.

Lifecycle Events and Notifications: Lifecycle Methods – registerShutdownHook()

registerShutdownHook() metodunun çağrılması, **JVM ile birlikte bir shutdown hook (kapanış kancası)** kaydeder.  
Bu metodun amacı, **JVM kapanmadan hemen önce uygulamanın düzgün (graceful) bir şekilde kapatılmasını sağlamaktır**.

JVM kapandığında, context daha önce kapatılmamışsa, **otomatik olarak kapatılır**.  
Bu sırada, **singleton bean’ler üzerindeki ilgili destroy (yok etme) metodları çağrılır**,  
böylece bu bean’ler sahip oldukları kaynakları serbest bırakabilir.

Lifecycle Events and Notifications: Lifecycle Callback Methods

**Bean’ler için tanımlanan lifecycle (yaşam döngüsü) callback metodları**,  
**IoC container tarafından otomatik olarak çağrılır**.

Bu callback metodları, **post-construct** (oluşturma sonrası) ve **pre-destroy** (yok edilme öncesi) gibi olaylara ilişkin bildirimleri alır.

**Elbette, yalnızca singleton scope’lu bean’ler pre-destroy olaylarını otomatik olarak alır**.

Bean’lerin olaylar hakkında **container tarafından bilgilendirilmesini (bildirim almasını)** sağlamanın başlıca **üç yolu vardır**:

**Yöntemler (Methods):**  
Bildirim alacak metodlar, ya **XML dosyasında** belirtilir ya da  
**JSR-250** anotasyonları olan @PostConstruct ve @PreDestroy kullanılarak tanımlanır.

**Arayüzler (Interfaces):**  
Bean’ler, **Spring’in belirli arayüzlerini (örneğin InitializingBean, DisposableBean)** implemente ederek bildirim alabilir.

**Anotasyonlar (Annotations):**  
@Bean anotasyonu, **bildirim alacak metodların tanımlanabileceği niteliklere (özelliklere)** sahiptir (örneğin initMethod, destroyMethod).

Lifecycle Events and Notifications: Lifecycle Callback Methods – Methods

Life – Cycle Events in XML

Yaşam döngüsü (lifecycle) olayları, **XML’deki <bean> etiketinin öznitelikleriyle (attributes) yapılandırılabilir**:

* init-method: **başlatma (initialization)** olayı için,
* destroy-method: **yok etme (destroy)** olayı için kullanılır.

Tüm bean’ler için geçerli olacak şekilde yaşam döngüsü olayları,  
**<beans> etiketinin öznitelikleriyle** de yapılandırılabilir:

* default-init-method ve
* default-destroy-method.

Bu yöntemler, yalnızca **ilgili <bean> etiketinde açıkça belirtilmediği (override edilmediği)** sürece etkili olur.

Bu öznitelikler (örneğin init-method, destroy-method) için belirtilen metodlar **herhangi bir parametre almaz**,  
ancak **geri dönüş tipine (return type)** sahip olabilirler.

Ayrıca, bu metodlar **erişim belirleyici (access modifier)** içerebilir (örn. public, private vb.).

PostConstruct & PreDestroy

java.annotation paketindeki @PostConstruct ve @PreDestroy anotasyonları,  
**bir bean’in yaşam döngüsü olayları (lifecycle events)** için kullanılan **iki callback (geri çağırım) anotasyonudur**.  
Bu anotasyonlar, **JSR-250** standardının bir parçasıdır.

Bean üzerinde **herhangi bir metoda uygulanabilirler**.  
Bu metodlar **private olabilir**, **parametre almazlar** ve **geri dönüş değeri (return value)** olabilir.

**Bir bean için birden fazla yaşam döngüsü olayı** yapılandırılabilir.

**Container’ın bu anotasyonları (@PostConstruct, @PreDestroy) işlemesi için**,  
XML dosyasında ya <context:annotation-config /> ya da <context:component-scan /> etiketi bulunmalı  
veya **Java tarafında @ComponentScan anotasyonu** kullanılmalıdır.

Lifecycle Events and Notifications: Lifecycle Callback Methods – Interfaces

Spring’s Life-Cycle Events

org.springframework.beans.factory paketinde, **callback (geri çağırım) bildirimlerini almak için kullanılan iki arayüz (interface)** vardır:

* InitializingBean arayüzünün afterPropertiesSet() metodu, **bean’in tüm özellikleri (properties) ayarlandıktan sonra çağrılır**.
* DisposableBean arayüzünün destroy() metodu, **bean yok edilirken çağrılır**.

Ancak **Spring**, bu arayüzleri kullanarak koda Spring bağımlılığı getirmemek adına  
**@PostConstruct ve @PreDestroy anotasyonlarının kullanılmasını önerir**.

Lifecycle – SmartLifecycle Interfaces

org.springframework.context.Lifecycle,  
**başlatma (start) ve durdurma (stop)** bildirimleri için kullanılan **basit bir arayüzdür (contract)**.  
**Otomatik bildirimler için tasarlanmamıştır**.

Bunun alt arayüzü olan org.springframework.context.SmartLifecycle ise,  
**olaylar için otomatik geri çağırım (callback) alır**.

Ayrıca, **SmartLifecycle arayüzü asenkron durdurma (asynchronous stop)** işlevini de sağlar.

Lifecycle Events and Notifications: Lifecycle Callback Methods – Annotations

Life – Cycle Events in @Bean

Yaşam döngüsü (life-cycle) olayları, @Bean anotasyonu ile şu öznitelikler kullanılarak yapılandırılabilir:

* initMethod: başlatma (initialization) olayı için
* destroyMethod: yok etme (destroy) olayı için

Bu öznitelikler için belirtilen metodlar **herhangi bir parametre almaz**,  
ancak **geri dönüş tipine (return type)** sahip olabilirler.  
Ayrıca, bu metodlar **herhangi bir erişim belirleyiciye (access modifier)** sahip olabilir (örn. public, private vb.).

close() and shutdown() Methods

Java konfigürasyonu ile @Bean kullanılarak tanımlanan bean’lerde,  
**public close() veya shutdown() metodlarından biri varsa**,  
**bu metodlardan biri otomatik olarak destruction (yok edilme) callback’i olarak çağrılır**.

Eğer bu metodların **otomatik olarak callback olarak çağrılmasını istemiyorsanız**,  
@Bean anotasyonundaki destroyMethod özniteliği boş bir string olarak verilerek  
**bu özellik devre dışı bırakılabilir**.  
Örneğin: @Bean(destroyMethod = "")

Which One?

**Arayüzleri (interface) kullanmak**, bean’leri bu arayüzlere **bağımlı hale getirir**.  
**JSR-250 anotasyonlarını kullanmak**, kodu **diğer yöntemlere göre daha taşınabilir (portable)** yapar.  
Ancak, **taşınabilirlik öncelik değilse**, arayüzleri kullanmak **daha iyi bir tercih olabilir**.

PROCESSORS

Extensions to ApplicationContext

**IoC (Inversion of Control)**, ApplicationContext nesnesini genişletmek (extend) için kullanılabilecek **birkaç arayüze (interface)** sahiptir.  
Bunlara genel olarak **"processor" (işleyici)** adı verilir.

İki ana processor şunlardır:

* BeanPostProcessor
* BeanFactoryPostProcessor

Processors

BeanFactoryPostProcessor, **BeanPostProcessor'lardan önce çalışır**.

Eğer processor’lar **classpath içinde yer alıyorsa**, ApplicationContext tarafından **otomatik olarak algılanırlar (autodetected)**.

Eğer bu processor’lar bir @Configuration sınıfı tarafından oluşturuluyorsa,  
onları döndüren @Bean metodlarının **static olarak tanımlanması gerekir**.  
Bunun nedeni, aynı sınıftaki diğer @Bean metodlarının tetiklenip,  
**başka bean’lerin erken başlatılmasının (initialization) önlenmek istenmesidir**.

BeanPostProcessor

org.springframework.beans.factory.config.BeanPostProcessor,  
**bean’ler üzerinde özel değişiklikler (custom modification) yapmaya olanak tanıyan** bir arayüzdür.

Bu arayüzde isteğe bağlı olarak uygulanabilecek **iki adet varsayılan (default) metod** bulunur:

* default Object postProcessAfterInitialization(Object bean, String beanName)
* default Object postProcessBeforeInitialization(Object bean, String beanName)

Bu metodlar sırasıyla:

* Bean tamamen başlatıldıktan (initialized) **sonra**,
* Bean başlatılmadan **önce** çağrılır.

postProcessBeforeInitialization() ve postProcessAfterInitialization() metodları,  
**herhangi bir bean başlatma (initialization) geri çağrımından (callback) hemen önce ve sonra çağrılır**.  
Bunlara örnek olarak:

* @PostConstruct anotasyonuna sahip bir metod
* veya InitializingBean arayüzündeki afterPropertiesSet() metodu gösterilebilir.
* Bean örneği oluşturulduktan hemen sonra ve diğer bağımlılıkların başlatılmasından (initialization) **hemen önce**  
  **programatik olarak bazı bağımlılıkları ayarlamak gibi özel işlemler**,  
  postProcessBeforeInitialization() metodunda uygulanabilir.
* Bean'in tüm bağımlılıklarının karşılanıp karşılanmadığını kontrol etmek  
  veya **başlatma işleminden sonra bean ya da bağımlılıkları üzerinde değişiklik yapmak gibi özel işlemler**,  
  postProcessAfterInitialization() metodunda gerçekleştirilebilir.

BeanFactoryPostProcessor

org.springframework.beans.factory.config.BeanFactoryPostProcessor,  
**bean fabrikası (bean factory) üzerinde özel değişiklikler (custom modification)** yapmaya olanak tanır.

Bu arayüzün **tek bir soyut (abstract) metodu** vardır:

void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)

Bu metod, örneğin:

* **kayıtlı tüm bean’ler hakkında bilgi almak**,
* veya **bean fabrikasındaki bazı özellikleri (properties) değiştirmek** için kullanılabilir.

Dikkat edilmelidir ki, postProcessBeanFactory() çağrıldığında **henüz hiçbir kayıtlı bean başlatılmamıştır**.  
Bu nedenle postProcessBeanFactory() metodu, **bean örneklerini (bean instances) değil, bean tanımlarını (bean definitions) değiştirmek için kullanılmalıdır**.  
Çünkü bu metod, **herhangi bir bean başlatılmadan önce çağrılır** ve burada bean başlatmaya çalışmak, erken ve beklenmedik bean örneklerinin oluşmasına yol açabilir.

Awareness Interfaces

Aware Interfaces

Spring, IoC (Inversion of Control) kapsayıcısındaki farklı nesnelerle ilgili **farkındalık (awareness)** sağlamak için çeşitli **"aware" arayüzleri (interfaces)** sunar.  
Bu arayüzleri uygulayan (implement eden) bean’ler, belirli nesneler hakkında bilgi sahibi olabilirler:

* BeanNameAware: Bean’in **kendi adını öğrenmesini sağlar**.
* BeanFactoryAware: Bean’in **kendisini oluşturan BeanFactory hakkında bilgi sahibi olmasını sağlar**.
* ApplicationContextAware: Bean’in **bağlı olduğu ApplicationContext hakkında bilgi sahibi olmasını sağlar**.

Bu arayüzlerin uygulanma sırası ise şöyledir:

**BeanNameAware → BeanFactoryAware → ApplicationContextAware**

org.springframework.beans.factory.Aware, bu arayüzlerin **ebeveyni (parent)** olan bir **işaretleyici (marker) arayüzdür**.  
**Herhangi bir işlevselliği yoktur.**

Bu arayüz, onu uygulayan bean’i bir **gözlemci (observer)** veya **dinleyici (listener)** olarak işaretler  
ve böylece **bildirim (notification) almaya uygun hale getirir**.

**Aware arayüzleri**, bean başlatıldıktan (initialized) **sonra çağrılır**.

BeanNameAware

BeanNameAware, **bean’in kendi adını öğrenmesini sağlar**.

Ancak **Spring, bir bean’in kendi adına bağımlı olmasını (depend on) önermez.**

BeanFactoryAware

BeanFactoryAware, **bean’in kendisini oluşturan BeanFactory hakkında bilgi sahibi olmasını sağlar**.

Bean, bu sayede **aynı BeanFactory tarafından oluşturulan diğer bean’lere erişmek isteyebilir**.

Ancak, eğer bir bean’in başka bir bean’e zaten bir bağımlılığı (örneğin bir alan ya da constructor aracılığıyla bir referansı) varsa,  
**bu arayüzü kullanmaya gerek yoktur**.

Bu arayüz, genellikle **bean’in doğrudan erişimi olmayan diğer bean’lere ulaşması gerektiğinde kullanılır**.

ApplicationContextAware

ApplicationContextAware, **bean’in bağlı olduğu ApplicationContext hakkında bilgi sahibi olmasını sağlar**.

Bean, bu sayede **yalnızca ApplicationContext nesnesi tarafından sağlanabilen sistem özellikleri (system properties), dosya kaynakları (file resources) gibi bilgilere ulaşmak isteyebilir**.

Warning

Spring, **referans dokümantasyonunda bu arayüzlerin kullanımına karşı uyarıda bulunur** ve şöyle der:

**Bu arayüzlerin kullanımı, kodunuzu Spring API’sine bağımlı hâle getirir**  
ve **Inversion of Control (Kontrolün Tersine Çevrilmesi)** ilkesine **uymaz**.

IoC Container Lifecycle for Beans

IoC Lifecycle

Spring IoC kapsayıcısının bean'lerle ilgili olarak geçtiği yaşam döngüsü (lifecycle) adımları şunlardır:

1. Verilen **XML dosyalarını** ve **classpath üzerindeki bileşenleri** tarar.
2. **Bağımlılık ilişkilerine göre bean’leri oluşturur**.
3. Bean’leri **bağımlılıklarıyla doldurur** (dependency injection).
4. **Aware arayüzlerini çağırır** (örn. BeanNameAware, BeanFactoryAware).
5. BeanPostProcessor arayüzünün **postProcessBeforeInitialization()** metodunu çağırır.
6. **Post-initialization (başlatma sonrası) geri çağrım (callback) metodlarını çağırır**.
7. BeanPostProcessor arayüzünün **postProcessAfterInitialization()** metodunu çağırır.
8. **Bean artık kullanıma hazırdır**.
9. Ve **ApplicationContext kapatılmadan hemen önce**,
10. **Pre-destruction (yıkım öncesi) geri çağrım metodlarını çağırır**.