Dependency Injection ve IoC

Spring'e değinmeden, saf Java ile Inversion of Control, Dependency Injection ve constructor/setter/field injection.

Orta 45 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/dependency-injection

---

Dependency Injection ve IoC

Bu ders, sitedeki ilk Spring Boot konusu olsa da, henüz Spring'e hiç dokunmuyor -- Dependency Injection (DI) ve Inversion of Control (IoC), Spring'den çok önce var olan, çerçeveden tamamen bağımsız iki tasarım fikri. Amaç, Spring'in `@Autowired` ile "sihirli" biçimde yaptığı şeyi önce tamamen elle, saf Java ile yapabilmek; bir sonraki konuda ("Spring IoC Container ve Bean Yaşam Döngüsü") bu elle yaptığımız işi otomatikleştiren container'ın kendisini işleyeceğiz. Sıkı bağlılık probleminden başlayıp constructor/setter/field injection'ın üçünü de karşılaştıracak, sonunda Spring'in bu üçünü nasıl otomatikleştirdiğine kısa bir bakış atacağız.

Dependency Injection ve IoC Nedir?

En basit haliyle Dependency Injection (DI), bir nesnenin ihtiyaç duyduğu başka nesneleri (bağımlılıklarını) kendisi `new` ile yaratmak yerine, dışarıdan hazır olarak almasıdır. Inversion of Control (IoC) ise bunun arkasındaki daha genel fikir: "kontrolün tersine çevrilmesi" -- normalde bir sınıf kendi bağımlılıklarını ve akışını kendisi yönetirken, IoC bu kontrolü sınıfın dışına, ayrı bir mekanizmaya (elle yazılmış bir "composition root"a ya da Spring gibi bir container'a) devreder. DI, IoC'nin en yaygın ve en somut gerçekleştirme biçimidir:

```
// DI olmadan: OrderService kendi bağımlılığını kendisi yaratır ve sahiplenir.
class OrderService {
    private final EmailSender sender = new EmailSender();
}

// DI ile: OrderService yalnızca neye ihtiyaç duyduğunu bildirir; hangi
// EmailSender'ın (ya da hangi alternatifin) verileceğine başkası karar verir.
class OrderService {
    private final EmailSender sender;
    OrderService(EmailSender sender) { this.sender = sender; }
}
```

İkinci versiyonda `OrderService`, kendisine verilen `EmailSender`'ın nereden geldiğini hiç bilmiyor -- bu, ilerleyen bölümlerde tek tek işleyeceğimiz constructor/setter/field injection'ın üçünün de ortak paydası.

Neden Var?

Dependency Injection'ın çözdüğü temel problem **sıkı bağlılık (tight coupling)** -- bir sınıfın, ihtiyaç duyduğu başka bir sınıfın somut implementasyonunu (`new SomeClass()` satırıyla) doğrudan içinde barındırması. Bunun üç somut maliyeti var: **test edilemezlik** (gerçek bir e-posta servisiyle test yazmak zorunda kalırsın, ağ çağrısı olmadan test edemezsin), **değişim zorluğu** (yarın SMS'e geçmek istediğinde `OrderService`'in içine girip kodu değiştirmen gerekir) ve **karışık sorumluluk** (bir sınıf hem "ne yapacağını" hem "bağımlılıklarını nasıl kuracağını" aynı anda üstlenir).

DI bu üçünü de, bağımlılığı sınıfın dışına taşıyarak çözer: `OrderService`, hangi `NotificationSender`'ı kullanacağını değil, yalnızca *bir* `NotificationSender`'a ihtiyacı olduğunu bilir. Bu ayrım, sonraki bölümlerde göreceğimiz gibi hem testleri hızlandırır hem de yeni bir kanal eklemeyi, mevcut kodu hiç değiştirmeden mümkün kılar.

Tarihçe

Dependency Injection kavramı Spring'den önce de vardı -- fikrin kökleri 1990'ların "Inversion of Control" tartışmalarına uzanır. Ama ismini ve popülerliğini büyük ölçüde Spring Framework'e borçlu: Rod Johnson, 2002'de yazdığı *Expert One-on-One J2EE Design and Development* kitabında, dönemin ağır ve karmaşık EJB (Enterprise JavaBeans) modeline karşı çok daha hafif bir alternatif önerdi -- bu fikirler 2004'te Spring Framework 1.0 olarak somutlaştı.

Aynı yıl Martin Fowler, "Inversion of Control Containers and the Dependency Injection pattern" başlıklı makalesinde, o zamana kadar belirsiz kullanılan "Inversion of Control" terimini netleştirip "Dependency Injection" ismini önerdi -- bugün kullandığımız terminoloji büyük ölçüde bu makaleden geliyor. 2009'da JSR-330 (`javax.inject`, bugünkü adıyla `jakarta.inject`), `@Inject` gibi ortak annotation'ları standartlaştırarak DI'ı yalnızca Spring'e özgü olmaktan çıkardı -- Spring hâlâ kendi `@Autowired`'ını tercih eder, ama `@Inject`'i de destekler.

Sıkı Bağlılık (Tight Coupling) Problemi

"Neden Var?" bölümünde sözünü ettiğimiz problemi somut kodda görelim -- bir `OrderService`, kendi `EmailSender`'ını kendisi yaratıyor:

```
// The "before" picture: OrderService constructs its own dependency with `new`,
// so it is permanently welded to EmailSender -- no other channel, and no fake
// version for a test, can ever take its place.
class EmailSender {
    void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

class OrderService {
    private final EmailSender emailSender = new EmailSender();

    void placeOrder(String customerEmail, String item) {
        // Business logic and object construction are tangled together here.
        emailSender.send(customerEmail, "Your order for '" + item + "' has been placed.");
    }
}

class TightlyCoupledOrderService {
    public static void main(String[] args) {
        OrderService orderService = new OrderService();
        orderService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.

        // There is no way to send this via SMS instead, and no way to replace
        // EmailSender with a fake for a test -- OrderService leaves us no seam.
    }
}
```

`OrderService`, `new EmailSender()` satırını içinde barındırdığı sürece, onu SMS'e geçirmenin ya da testte sahte bir gönderici kullanmanın tek yolu `OrderService`'in kaynak kodunu açıp değiştirmektir. Sorun, `EmailSender`'ın kendisinde değil -- `OrderService`'in *hangi* göndericiyi kullanacağına dair kararı, göndericiyi *kullanma* mantığıyla aynı yere gömmüş olmasında.

Inversion of Control (IoC) Nedir?

IoC'nin en küçük hali: nesne yaratma kararını, onu kullanan sınıfın dışına taşımak. Aşağıda `OrderNotifier`, `EmailMessageSender`'ı artık kendisi `new` ile yaratmıyor -- bu işi ayrı bir factory'ye devrediyor:

```
// Inversion of Control, in its smallest possible form: OrderNotifier no
// longer decides HOW its dependency gets built -- a separate factory does,
// and OrderNotifier only asks for the finished object.
interface MessageSender {
    void send(String to, String message);
}

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

// The factory is the one place that knows EmailMessageSender exists. Swapping
// the concrete implementation later means editing this one method, not every
// class that used to call `new` directly.
class MessageSenderFactory {
    static MessageSender create() {
        return new EmailMessageSender();
    }
}

class OrderNotifier {
    private final MessageSender messageSender;

    OrderNotifier() {
        // OrderNotifier still decides to CALL the factory itself here -- that
        // is the piece "Dependency Injection: Programming Against a Contract"
        // removes next: even the factory call moves outside this class.
        this.messageSender = MessageSenderFactory.create();
    }

    void notifyCustomer(String email, String item) {
        messageSender.send(email, "Your order for '" + item + "' has been placed.");
    }
}

class ManualFactoryExample {
    public static void main(String[] args) {
        OrderNotifier notifier = new OrderNotifier();
        notifier.notifyCustomer("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.
    }
}
```

`OrderNotifier` hâlâ factory'yi kendisi çağırıyor -- kontrol tam olarak henüz tersine çevrilmiş değil, yalnızca bir adım dışarı taşınmış. Bir sonraki bölümde bu son adımı da kaldırıp, `OrderService`'in hiçbir şeyi kendisi çağırmadan, hazır bir nesneyi doğrudan constructor'ından alacağı hâline geleceğiz.

> Bu fikir bazen "Hollywood Principle" (Hollywood İlkesi) diye anılır: "Don't call us, we'll call you" -- yani bir bileşen, ihtiyaç duyduğu şeyi kendisi çağırıp istemez, ihtiyacı olduğunda dışarıdan kendisine getirilmesini bekler.

Dependency Injection: Sözleşmeye Karşı Programlamak

Şimdi hem factory adımını hem de somut sınıf bağımlılığını tamamen kaldırıyoruz -- `OrderService`, bir `NotificationSender` **interface'ine** bağımlı, hangi implementasyonun kullanılacağına ise dışarıdan, constructor aracılığıyla karar veriliyor:

```
// The "after" picture: OrderService now depends only on an abstraction
// (NotificationSender), never on a concrete class -- the same interface
// pattern from the "Interface" lesson, applied to the dependency problem.
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);
    }
}

class OrderService {
    private final NotificationSender notificationSender;

    // The dependency now arrives from OUTSIDE, through the constructor --
    // OrderService no longer contains the words "new EmailNotificationSender()"
    // anywhere. This is dependency injection: the caller decides, and injects.
    OrderService(NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

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

class NotificationSenderExample {
    public static void main(String[] args) {
        OrderService emailBackedService = new OrderService(new EmailNotificationSender());
        emailBackedService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.

        // Same OrderService class, a completely different channel -- nothing
        // inside OrderService changed to make this possible.
        OrderService smsBackedService = new OrderService(new SmsNotificationSender());
        smsBackedService.placeOrder("+90 555 000 00 00", "Java 21 Book");
        // [sms to +90 555 000 00 00] Your order for 'Java 21 Book' has been placed.
    }
}
```

Bu, Interface dersindeki "implementasyona değil, arayüze göre programla" ilkesinin dependency injection'a uygulanmış hâli. `OrderService`'in kaynak kodunda

`EmailNotificationSender` ya da `SmsNotificationSender` isimleri hiç geçmiyor -- `main`'deki iki çağrının gösterdiği gibi, aynı `OrderService`, hiçbir satırı değişmeden iki farklı kanala bağlanabiliyor.

Constructor Injection

Bağımlılığı vermenin en yaygın yolu, onu bir constructor parametresi olarak almak ve `final` bir alanda saklamaktır:

```
// Constructor Injection: the dependency is a required constructor parameter,
// stored in a `final` field. There is no way to end up with a half-built
// OrderService that is missing its NotificationSender -- the object simply
// cannot exist without one.
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;
    private final String storeName;

    // Multiple dependencies/parameters are injected the same way -- just more
    // constructor arguments. All of them are guaranteed to be set once the
    // constructor returns.
    OrderService(NotificationSender notificationSender, String storeName) {
        this.notificationSender = notificationSender;
        this.storeName = storeName;
    }

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

class ConstructorInjectionExample {
    public static void main(String[] args) {
        OrderService orderService = new OrderService(new EmailNotificationSender(), "Java
Kitabevi");
        orderService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] [Java Kitabevi] Your order for 'Java 21 Book' has
been placed.

        // The line below would not compile if uncommented -- there is no
        // no-argument constructor, so "forgetting" the dependency is not an
        // option the compiler will allow.
        // OrderService broken = new OrderService();
    }
}
```

`notificationSender` ve `storeName`, `OrderService` nesnesi var olduğu sürece **her zaman** doludur -- bunları unutup boş bırakmanın hiçbir yolu yok, çünkü derleyici o parametreleri olmadan bir `OrderService` yaratmana izin vermiyor. "Neden Constructor Injection Öneriliyor?" bölümünde bu garantinin neden önemli olduğuna daha yakından bakacağız.

Setter Injection

İkinci yaklaşım, bağımlılığı nesne yaratıldıktan **sonra** bir setter metoduyla vermek:

```
// Setter Injection: the dependency is assigned through an ordinary setter
// method AFTER the object already exists -- useful for genuinely optional
// dependencies, but it also means the object can exist in a "half-wired"
// state until someone remembers to call the setter.
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 {
    // Not final -- it has to stay reassignable so the setter can populate it
    // after construction.
    private NotificationSender notificationSender;

    void setNotificationSender(NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

    void placeOrder(String customerContact, String item) {
        // If setNotificationSender(...) was never called, this throws a
        // NullPointerException here -- at call time, not at construction time.
        notificationSender.send(customerContact, "Your order for '" + item + "' has been
placed.");
    }
}

class SetterInjectionExample {
    public static void main(String[] args) {
        OrderService orderService = new OrderService();
        orderService.setNotificationSender(new EmailNotificationSender());
        orderService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.

        // A second OrderService, created but never wired -- this compiles fine
        // and only fails much later, when placeOrder() actually runs.
        OrderService forgotten = new OrderService();
        try {
            forgotten.placeOrder("mehmet@example.com", "Spring Boot Book");
        } catch (NullPointerException e) {
            System.out.println("Failed: notificationSender was never set.");
            // Failed: notificationSender was never set.
        }
    }
}
```

Burada `notificationSender` artık `final` değil -- setter'ın onu sonradan doldurabilmesi için değişebilir kalması gerekiyor. Bunun bedelini `main`'deki ikinci `OrderService` gösteriyor: `setNotificationSender(...)` çağrılmadan `placeOrder(...)` çağrıldığında hata, nesne yaratılırken değil, tam da o satırda, çalışma zamanında ortaya çıkıyor.

Field Injection

Üçüncü yaklaşımda bağımlılık ne constructor'dan ne setter'dan geçer -- doğrudan bir alana "enjekte edilir". Spring'de bunu `@Autowired` bir alanla görürsün; burada aynı mekanizmayı, bir framework'ün arka planda ne yaptığını görmek için elle simüle ediyoruz:

```
import java.lang.reflect.Field;

// Field Injection: a framework (Spring's @Autowired on a field is the classic
// example) reaches directly into a private field and sets it via reflection --
// the same Field.setAccessible(true) + Field.set(...) mechanism from the
// Reflection lesson's "Accessing Private Fields and Methods" section, just driven
// by a framework instead of your own code.
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 {
    // A real Spring field would be annotated @Autowired; there is no
    // constructor or setter here at all -- nothing but the bare field.
    private NotificationSender notificationSender;

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

class FieldInjectionExample {
    public static void main(String[] args) throws ReflectiveOperationException {
        OrderService orderService = new OrderService();

        // This is, in miniature, what a dependency injection framework does
        // for every @Autowired field: find it by reflection, force it
        // accessible, and set it -- no constructor call, no setter call.
        Field field = OrderService.class.getDeclaredField("notificationSender");
        field.setAccessible(true);
        field.set(orderService, new EmailNotificationSender());

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

        // Nothing in OrderService's own source code reveals how the field got
        // its value -- that opacity is exactly why "Common Mistakes" warns
        // against relying on field injection.
    }
}
```

`OrderService`'in hiçbir constructor'ı ya da setter'ı yok -- `field.set(...)` çağrısı, Reflection dersindeki "Private Alan ve Metotlara Erişmek" bölümünde gördüğümüz tam mekanizmayla, `private` alana dışarıdan doğrudan yazıyor. Gerçek bir Spring uygulamasında bunu sen değil, container yapar; ama sonuç aynıdır: `OrderService`'in kaynak koduna bakarak bu alanın nasıl dolduğunu anlayamazsın.

> Field injection, `OrderService`'i test etmek için de reflection gerektirir -- normal bir `new OrderService(fakeSender)` çağrısı bağımlılığı hiç ayarlayamaz, çünkü ortada bunu kabul eden bir constructor yok. "Yaygın Hatalar" bölümünde bu ve benzer nedenlerle field injection'ın neden tercih edilmediğine değineceğiz.

Injection Türlerini Karşılaştırma

Üç yaklaşımı yan yana koyduğumuzda:

- **Constructor Injection:** bağımlılık `final`, zorunlu, nesne yaratılır yaratılmaz garantili. Eksik bir bağımlılık **derleme zamanında** (parametre eksikse) ya da en geç nesne yaratılırken yakalanır.
- **Setter Injection:** bağımlılık değişebilir, isteğe bağlı olabilir. Eksik bir bağımlılık, yalnızca o bağımlılığın gerçekten kullanıldığı satırda, çalışma zamanında ortaya çıkar.
- **Field Injection:** en az kod (constructor/setter yazmaya gerek yok), ama en az kontrol -- bağımlılığın nereden geldiği kaynak koddan anlaşılmaz, elle (framework'süz) test etmek reflection gerektirir.

Bu üçü birbirini dışlamaz -- aynı sınıfta bir zorunlu bağımlılık constructor'dan, isteğe bağlı bir tanesi setter'dan gelebilir. Ama pratikte tek bir stil neredeyse her zaman diğerlerine tercih edilir; bir sonraki bölüm nedenini işliyor.

Neden Constructor Injection Öneriliyor?

Constructor injection'ın önerilmesinin sebebi rastgele değil -- garantilerinden geliyor:

```
import java.util.Objects;

// Constructor injection lets every dependency be `final` -- once built, an
// OrderService can never end up pointing at a different (or missing)
// NotificationSender. Combined with an explicit null-check, a broken wiring
// attempt fails immediately and loudly, not with a mysterious NPE three
// method calls later (compare with "Setter Injection").
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) {
        // Fail fast: if the caller passes null, we find out right here, at
        // the exact line that got it wrong -- not somewhere deep inside
        // placeOrder() later.
        this.notificationSender = Objects.requireNonNull(notificationSender,
"notificationSender must not be null");
    }

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

class ImmutableOrderService {
    public static void main(String[] args) {
        OrderService orderService = new OrderService(new EmailNotificationSender());
        orderService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.

        try {
            new OrderService(null);
        } catch (NullPointerException e) {
            System.out.println("Failed immediately: " + e.getMessage());
            // Failed immediately: notificationSender must not be null
        }
    }
}
```

`Objects.requireNonNull(...)` sayesinde, `null` bir bağımlılıkla `OrderService` yaratmaya çalışmak, `main`'in gösterdiği gibi **anında** patlıyor -- hatanın kaynağı, hatanın ortaya çıktığı satırla aynı yerde. Setter injection'da (bkz. "Setter Injection") bu hata, unutulan setter çağrısından çok sonra, ilgisiz görünen bir satırda ortaya çıkabilirdi.

> Constructor injection'ın gizli bir faydası daha var: bir constructor'ın parametre listesi beşe altıya çıkmaya başladığında, bu genelde sınıfın çok fazla sorumluluk yüklendiğinin erken bir işaretidir -- setter/field injection bu "kod kokusunu" gizler, çünkü çok sayıda bağımlılık dağınık satırlara yayılır ve tek bakışta fark edilmez.

Dependency Injection ve Test Edilebilirlik

DI'ın en somut günlük faydası testte ortaya çıkar -- gerçek bir `NotificationSender` yerine, yalnızca ne gönderildiğini hafızada tutan sahte bir tanesini vermek yeterli:

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

// The payoff of depending on an interface: in a test, we can swap the real
// EmailNotificationSender for a tiny in-memory fake that just records what it
// was asked to send -- no real email is sent, and the test can assert on
// exactly what OrderService tried to do.
interface NotificationSender {
    void send(String to, String message);
}

class FakeNotificationSender implements NotificationSender {
    final List<String> sentMessages = new ArrayList<>();

    @Override
    public void send(String to, String message) {
        sentMessages.add(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.");
    }
}

class TestableOrderServiceExample {
    public static void main(String[] args) {
        FakeNotificationSender fake = new FakeNotificationSender();
        OrderService orderService = new OrderService(fake);

        orderService.placeOrder("ayse@example.com", "Java 21 Book");

        // A hand-rolled assertion -- no test framework needed to see the point:
        // this check runs against memory, in milliseconds, without a real
        // email provider or network call anywhere in sight.
        if (fake.sentMessages.size() != 1) {
            throw new AssertionError("Expected exactly one message to be sent");
        }
        System.out.println("Test passed: " + fake.sentMessages.get(0));
        // Test passed: ayse@example.com: Your order for 'Java 21 Book' has been placed.
    }
}
```

`FakeNotificationSender`, gerçek bir e-posta sağlayıcısına hiç bağlanmadan çalışıyor -- test, milisaniyeler içinde bitiyor ve `sentMessages` listesine bakarak `OrderService`'in tam olarak ne

yapmaya çalıştığını doğrulayabiliyoruz. Bu, "Sıkı Bağlılık (Tight Coupling) Problemi" bölümündeki `TightlyCoupledOrderService`'le hiçbir şekilde mümkün değildi -- orada `EmailSender`'ı değiştirmenin tek yolu kaynak kodu düzenlemekti.

Spring Olmadan Elle Bağımlılık Enjeksiyonu (Composition Root)

Buraya kadarki her `main` metodu aslında küçük bir "composition root"tu -- uygulamanın somut sınıfları bildiği tek yer. Bunu daha büyük bir örnekte, birden fazla bağımlılığı aynı anda bağlayarak netleştirelim:

```
// A "composition root": one single place in the whole application where
// `new` is allowed to wire concrete classes together. Every class below this
// point (OrderService) only ever sees interfaces -- exactly what a Spring
// container will automate in the next lesson, done here with nothing but
// plain constructors.
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);
    }
}

interface ReceiptPrinter {
    void print(String item, double price);
}

class ConsoleReceiptPrinter implements ReceiptPrinter {
    @Override
    public void print(String item, double price) {
        System.out.printf("[receipt] %s - %.2f TL%n", item, price);
    }
}

class OrderService {
    private final NotificationSender notificationSender;
    private final ReceiptPrinter receiptPrinter;

    OrderService(NotificationSender notificationSender, ReceiptPrinter receiptPrinter) {
        this.notificationSender = notificationSender;
        this.receiptPrinter = receiptPrinter;
    }

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

```
class CompositionRootExample {
    // This method is the composition root: the one place that knows about
    // EmailNotificationSender and ConsoleReceiptPrinter by name. Nothing else
    // in the application does.
    static OrderService buildOrderService() {
        NotificationSender notificationSender = new EmailNotificationSender();
        ReceiptPrinter receiptPrinter = new ConsoleReceiptPrinter();
        return new OrderService(notificationSender, receiptPrinter);
    }

    public static void main(String[] args) {
        OrderService orderService = buildOrderService();
        orderService.placeOrder("ayse@example.com", "Java 21 Book", 349.90);
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.
        // [receipt] Java 21 Book - 349.90 TL
    }
}
```

`buildOrderService()` dışında, ne `OrderService`'in kendisi ne de onu çağıran kod `EmailNotificationSender` ya da `ConsoleReceiptPrinter`'ın var olduğunu biliyor. Gerçek bir uygulamada bu desen "Pure DI" ya da "Poor Man's DI" olarak anılır -- hiçbir framework'e ihtiyaç duymadan, yalnızca sınıf ve constructor kullanarak IoC'nin tüm faydalarını elde etmeyi sağlar; küçük uygulamalarda veya framework bağımlılığından kaçınmak istediğinde hâlâ gayet geçerli bir seçimdir.

Spring'in DI'ı Nasıl Otomatikleştirdiği (Kısa Bakış)

Az önce elle yazdığımız composition root'u, Spring bir container ile otomatikleştirir -- sınıfları tarayıp (`@Component` / `@Service`), constructor'larını okuyup (`@Autowired`), doğru sırayla nesneleri kendisi kurar:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

// A preview only -- this file will not do anything useful on its own, since
// there is no container here to create these objects. It shows the same
// OrderService design from "Manual Dependency Injection Without Spring",
// now annotated so that a Spring container could build the composition root
// FOR us.
interface NotificationSender {
    void send(String to, String message);
}

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

@Service
class OrderService {
    private final NotificationSender notificationSender;

    // @Autowired on a constructor is optional when there is only one
    // constructor (Spring uses it automatically) -- it is written explicitly
    // here to keep the intent visible, matching "Constructor Injection".
    @Autowired
    OrderService(NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

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

class SpringPreviewExample {
    // No output worth demonstrating here: without an ApplicationContext,
    // nobody scans for @Component/@Service or calls this constructor.
    // "Spring IoC Container & Bean Lifecycle" is where that container itself
    // gets built.
    public static void main(String[] args) {
        System.out.println("This class needs a Spring ApplicationContext to do anything --
see the next lesson.");
        // This class needs a Spring ApplicationContext to do anything -- see the next
lesson.
    }
}
```

Bu dosya, üzerinde çalışan bir `ApplicationContext` olmadığı için tek başına bir şey yapmıyor -- tarama yapan, `@Autowired` constructor'ı bulup çağıran, "Spring IoC Container ve Bean Yaşam Döngüsü" konusunda ele alacağımız container'ın kendisi. Şimdilik önemli olan şu:

burada gördüğün `OrderService` ile "Spring Olmadan Elle Bağımlılık Enjeksiyonu (Composition Root)" bölümündeki `OrderService` **tasarım olarak birebir aynı** -- Spring, `buildOrderService()`'in elle yaptığı işi, annotation'lara bakarak kendisi yapıyor.

> Reflection dersindeki "Basit Dependency Injection Container" mini projesiyle karıştırma: orada yazdığımız `SimpleContainer`, bir tipin constructor'ının hangi parametrelere ihtiyaç duyduğunu **kendi başına, reflection ile keşfediyordu** -- burada ise hangi implementasyonun hangi arayüze bağlanacağını `@Component` / `@Service` ile biz açıkça işaretliyoruz. Spring'in gerçek container'ı ikisinin bir karışımıdır: hem reflection kullanır hem de annotation'larla yönlendirilir.

Best Practices

- **Varsayılan olarak constructor injection kullan** -- zorunlu bağımlılıkları `final` yapar, eksik bir bağımlılığı en erken noktada yakalar (bkz. "Neden Constructor Injection Öneriliyor?").
- **Setter injection'ı yalnızca gerçekten isteğe bağlı bağımlılıklar için sakla** -- bir bağımlılık olmadan sınıf anlamlı çalışamıyorsa, onu setter'a değil constructor'a koy.
- **Field injection'dan kaçın** -- ne test edilebilirliği ne de bağımlılıkların açıkça görünür olmasını sağlar (bkz. "Field Injection" ve "Yaygın Hatalar").
- **Somut sınıflar yerine arayüzlere bağımlı ol** ("Dependency Injection: Sözleşmeye Karşı Programlamak") -- bu, hem gerçek implementasyonlar arasında geçişi hem testte sahte implementasyon kullanmayı hiçbir çağıran kodu değiştirmeden mümkün kılar.
- **Bir constructor'ın parametre sayısı arttıkça bunu bir uyarı olarak oku** -- genelde sınıfın çok fazla sorumluluk yüklendiğinin işaretidir, ekstra bir parametre eklemek yerine sınıfı bölmeyi değerlendir.
- `Objects.requireNonNull(...)` **ile fail-fast davran** ("Neden Constructor Injection Öneriliyor?") -- bir bağımlılık eksikse bunu hemen, nesne yaratılırken öğrenmek, çok sonra ilgisiz bir hatayla karşılaşmaktan her zaman daha iyidir.

Yaygın Hatalar

1. Field injection'ı, "daha az kod yazdığı için" varsayılan tercih yapmak. Daha az kod, daha az kontrol demektir -- bağımlılık nereden geliyor, kaynak koddan anlaşılmaz ve elle test etmek reflection gerektirir (bkz. "Field Injection").

2. Setter injection'daki eksik bir bağımlılığı, hatanın ortaya çıktığı satırda aramak. Hata genelde `setX(...)` çağrısının unutulduğu, çok daha önceki bir satırdan kaynaklanır (bkz. "Setter Injection").

3. Bir constructor'ın beş altı parametreye çıkmasını normal karşılamak. Bu, sınıfın tek bir sorumluluktan fazlasını yüklendiğinin erken bir işaretidir (bkz. "Neden Constructor Injection Öneriliyor?").

4. `Objects.requireNonNull(...)` gibi bir kontrol olmadan, `null` bir bağımlılığın sessizce kabul edilmesine izin vermek. Böyle bir nesne başarıyla yaratılır ama ilk gerçek kullanımda, ilgisiz görünen bir yerde patlar (bkz. "Neden Constructor Injection Öneriliyor?").

5. DI'ı yalnızca Spring'e özgü bir kavram sanmak. DI, "Spring Olmadan Elle Bağımlılık Enjeksiyonu (Composition Root)" bölümünde gördüğümüz gibi hiçbir framework olmadan da uygulanabilir bir tasarım fikridir -- Spring bunu yalnızca otomatikleştirir.

6. Somut sınıflara bağımlı kalıp arayüz tanımlamayı atlamak. Bu, "Neden Var?" bölümündeki sıkı bağlılık problemini geri getirir ve testte sahte bir implementasyon kullanmayı imkânsızlaştırır.

Özet, Cheat Sheet ve Terimler Sözlüğü

Dependency Injection, bir nesnenin bağımlılıklarını kendisi yaratmak yerine dışarıdan alması; Inversion of Control ise bunun arkasındaki daha genel "kontrolü dışarı devret" fikridir. Öne çıkan noktalar:

- Sıkı bağlılık (`new` ile doğrudan somut sınıf yaratmak), test edilemezlik, değişim zorluğu ve karışık sorumluluğa yol açar
- Üç injection stili: **constructor** (zorunlu, `final`, en erken hata yakalama), **setter** (isteğe bağlı, sonradan değişebilir), **field** (en az kod, en az kontrol)
- Constructor injection varsayılan tercih olmalı -- garantili doluluk, fail-fast doğrulama, kalabalık parametre listesi erken bir tasarım uyarısı olarak işlev görür
- "Composition root": uygulamanın somut sınıfları bildiği, `new` çağrılarının toplandığı tek bir yer -- Spring olmadan da IoC'nin faydalarını sağlar
- Spring, aynı fikri `@Component` / `@Service` taraması ve `@Autowired` ile otomatikleştirir -- container'ın kendisi bir sonraki konunun (Spring IoC Container ve Bean Yaşam Döngüsü) konusu

Hızlı referans:

```
// Sıkı bağlılık (kaçınılması gereken)
class OrderService {
    private final EmailSender sender = new EmailSender();
}

// Constructor injection (önerilen varsayılan)
class OrderService {
    private final NotificationSender sender;
    OrderService(NotificationSender sender) {
        this.sender = Objects.requireNonNull(sender);
    }
}

// Setter injection (yalnızca gerçekten isteğe bağlı bağımlılıklar için)
class OrderService {
    private NotificationSender sender;
    void setSender(NotificationSender sender) { this.sender = sender; }
}

// Field injection (Spring: @Autowired; elle karşılığı yok, framework/reflection gerekir)
class OrderService {
    private NotificationSender sender; // framework tarafından reflection ile set edilir
}

// Composition root: somut sınıfları bilen tek yer
class AppComposition {
    static OrderService buildOrderService() {
        return new OrderService(new EmailNotificationSender());
    }
}
```

Terimler Sözlüğü

Dependency Injection (DI) — Bir nesnenin ihtiyaç duyduğu bağımlılıkları kendisi yaratmak yerine dışarıdan alması.

Inversion of Control (IoC) — Bir bileşenin akışını/bağımlılıklarını kendisinin değil, dışarıdaki bir mekanizmanın (composition root ya da container) yönetmesi; DI, IoC'nin en yaygın somut gerçekleştirme biçimidir.

Sıkı bağlılık (tight coupling) — Bir sınıfın, ihtiyaç duyduğu başka bir sınıfın somut implementasyonuna doğrudan (`new` ile) bağımlı olması.

Constructor Injection — Bağımlılığın zorunlu bir constructor parametresi olarak alınıp `final` bir alanda saklanması.

Setter Injection — Bağımlılığın, nesne yaratıldıktan sonra bir setter metoduyla, isteğe bağlı olarak verilmesi.

Field Injection — Bağımlılığın, ne constructor ne setter kullanılmadan doğrudan bir alana (genelde reflection ile, bir framework tarafından) atanması.

Composition root — Bir uygulamada somut sınıfların bilindiği, `new` çağrılarının toplandığı tek bir yer; Pure DI/Poor Man's DI olarak da anılır.

Fail-fast — Bir hatanın (örn. eksik bir bağımlılık), ortaya çıktığı en erken noktada (genelde nesne yaratılırken) fırlatılması; hatanın kaynağını bulmayı kolaylaştırır.

Test double / fake — Testte gerçek bir implementasyonun yerine geçen, davranışı basitleştirilmiş ya da gözlemlenebilir yapılmış bir nesne.

Ek: Mini Proje — Çok Kanallı Bildirim Dağıtıcısı

Şimdiye kadar öğrendiklerimizi ("Constructor Injection", "Dependency Injection: Sözleşmeye Karşı Programlamak") birleştirip bir adım ileri götürelim: bağımlılık tek bir `NotificationSender` değil, **birden fazla implementasyonun tamamı** olsun. Fikir basit -- `NotificationDispatcher`, kaç kanal olduğunu ya da bunların ne olduğunu hiç bilmeden, kendisine verilen listedeki her kanala aynı mesajı iletiyor:

```
import java.util.List;

// Combines "Constructor Injection" (a required, final dependency) with a
// twist Spring uses constantly: injecting a WHOLE LIST of implementations at
// once, so every registered channel gets used without NotificationDispatcher
// ever naming a single concrete class.
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);
    }
}

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

class NotificationDispatcher {
    private final List<NotificationSender> senders;

    // Every NotificationSender the caller decides to pass in gets used -- the
    // dispatcher itself has zero knowledge of how many channels exist or what
    // they are called.
    NotificationDispatcher(List<NotificationSender> senders) {
        this.senders = senders;
    }

    void dispatch(String to, String message) {
        for (NotificationSender sender : senders) {
            sender.send(to, message);
        }
    }
}
```

```
import java.util.List;

class NotificationDispatcherDemo {
    public static void main(String[] args) {
        // The composition root: this is the only place that lists the
        // concrete channels by name.
        NotificationDispatcher allChannels = new NotificationDispatcher(
                List.of(new EmailNotificationSender(), new SmsNotificationSender(), new
PushNotificationSender()));

        allChannels.dispatch("ayse@example.com", "Your order has shipped.");
        // [email to ayse@example.com] Your order has shipped.
        // [sms to ayse@example.com] Your order has shipped.
        // [push to ayse@example.com] Your order has shipped.

        // A second dispatcher, wired with a different (smaller) list -- same
        // NotificationDispatcher class, no code changes required.
        NotificationDispatcher emailOnly = new NotificationDispatcher(List.of(new
EmailNotificationSender()));
        emailOnly.dispatch("mehmet@example.com", "Your order has shipped.");
        // [email to mehmet@example.com] Your order has shipped.
    }
}
```

`NotificationDispatcher`'ın constructor'ı, tek bir `NotificationSender` yerine bir `List<NotificationSender>` alıyor -- composition root, listeye kaç eleman koyacağına (`allChannels` üç kanallı, `emailOnly` tek kanallı) kendisi karar veriyor, `dispatch(...)` metodunun tek satırı bile değişmiyor.

> Gerçek bir Spring container'ında `List<NotificationSender>` enjekte ettiğinde, listenin sırası varsayılan olarak **tanımsızdır** (bean tanım sırasına, hatta sınıf tarama sırasına bağlı olabilir) -- belirli bir sıraya ihtiyacın varsa `@Order` annotation'ı ya da `Ordered` arayüzü kullanılır. Burada elle yazdığımız `List.of(...)` çağrısında sıra bizim kontrolümüzde, ama bu güvence Spring'in otomatik taramasında kendiliğinden gelmez.

Ek: Mini Proje — Ödeme İşlemcisi

Son mini proje, aynı fikirleri farklı bir alanda (ödeme işleme) ve bir isteğe bağlı bağımlılıkla ("Injection Türlerini Karşılaştırma" bölümünde bahsettiğimiz gibi her bağımlılığın zorunlu olması gerekmez) tekrar gösteriyor. `PaymentProcessor`, zorunlu bir `PaymentGateway`'e (`Objects.requireNonNull`, bkz. "Neden Constructor Injection Öneriliyor?") ve isteğe bağlı, `null` olabilen bir `FraudChecker`'a bağımlı:

```
import java.util.Objects;

// A second, different domain to show the same ideas hold generally -- not
// just for notifications. PaymentProcessor requires a PaymentGateway
// (constructor injection, see "Why Is Constructor Injection Recommended?"),
// and accepts an OPTIONAL FraudChecker that may legitimately be null -- a
// reminder that not every collaborator needs Objects.requireNonNull.
interface PaymentGateway {
    boolean charge(String cardNumber, double amount);
}

class CreditCardGateway implements PaymentGateway {
    @Override
    public boolean charge(String cardNumber, double amount) {
        System.out.printf("[credit card] Charged %.2f TL to card ending in %s%n",
                amount, cardNumber.substring(cardNumber.length() - 4));
        return true;
    }
}

interface FraudChecker {
    boolean looksSuspicious(double amount);
}

class ThresholdFraudChecker implements FraudChecker {
    private final double threshold;

    ThresholdFraudChecker(double threshold) {
        this.threshold = threshold;
    }

    @Override
    public boolean looksSuspicious(double amount) {
        return amount > threshold;
    }
}

class PaymentProcessor {
    private final PaymentGateway gateway;
    private final FraudChecker fraudChecker; // may be null -- genuinely optional

    PaymentProcessor(PaymentGateway gateway, FraudChecker fraudChecker) {
        this.gateway = Objects.requireNonNull(gateway, "gateway must not be null");
        this.fraudChecker = fraudChecker;
    }

    boolean process(String cardNumber, double amount) {
        if (fraudChecker != null && fraudChecker.looksSuspicious(amount)) {
            System.out.println("[fraud-check] Blocked a suspicious payment of " + amount +
" TL");
            return false;
        }
        return gateway.charge(cardNumber, amount);
    }
}
```

```
class PaymentProcessorDemo {
    public static void main(String[] args) {
        // With fraud checking enabled.
        PaymentProcessor guarded = new PaymentProcessor(new CreditCardGateway(), new
ThresholdFraudChecker(5000));
        guarded.process("4242424242424242", 250.00);
        // [credit card] Charged 250.00 TL to card ending in 4242
        guarded.process("4242424242424242", 8000.00);
        // [fraud-check] Blocked a suspicious payment of 8000.0 TL

        // Without a fraud checker at all -- perfectly legal, since it is optional.
        PaymentProcessor unguarded = new PaymentProcessor(new CreditCardGateway(), null);
        unguarded.process("4242424242424242", 8000.00);
        // [credit card] Charged 8000.00 TL to card ending in 4242

        // A fake gateway swapped in, exactly like "Dependency Injection ve Test
        // Edilebilirlik" -- no real payment provider involved.
        PaymentGateway fakeGateway = (cardNumber, amount) -> {
            System.out.println("[fake] Pretending to charge " + amount + " TL, no real
network call made.");
            return true;
        };
        PaymentProcessor testable = new PaymentProcessor(fakeGateway, null);
        testable.process("4242424242424242", 100.00);
        // [fake] Pretending to charge 100.0 TL, no real network call made.
    }
}
```

`PaymentProcessorDemo`'daki üç senaryoya dikkat et: dolandırıcılık kontrolüyle, kontrol olmadan (`null` geçirilerek) ve son olarak "Dependency Injection ve Test Edilebilirlik" bölümündeki gibi bir lambda ile anında yazılmış sahte bir `PaymentGateway` ile. Üçünde de `PaymentProcessor`'ın kendi kodu tek bir satır bile değişmiyor.

> `FraudChecker fraudChecker`'ın `null` olabilmesine izin vermek yerine, Spring dünyasında bu genelde `Optional<FraudChecker>` ya da `@Autowired(required = false)` ile daha açık hâle getirilir -- "Spring IoC Container ve Bean Yaşam Döngüsü" konusunda, isteğe bağlı bağımlılıkları Spring'in nasıl ifade ettiğine bakacağız.