Reflection

Java'da çalışma zamanında sınıf, alan, metot ve constructor bilgisine erişme ve bunları dinamik olarak kullanma.

İleri 50 dk

Son güncelleme: 1 Kasım 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/reflection

---

Reflection

Java'da **Reflection**, bir programın çalışma zamanında (runtime) kendi sınıflarını, alanlarını, metotlarını ve constructor'larını inceleyip, hatta bunları dinamik olarak çağırıp /oluşturabilmesini sağlayan bir API'dir. Normalde derleme zamanında (compile-time) bilinmesi gereken tip bilgisine, programın kendisi çalışırken erişmeyi sağlar — "kod, kendi kendini inceleyebilir" fikrinin Java'daki karşılığıdır.

Reflection Nedir?

Elindeki bir nesnenin tipini, çalışma zamanında, hakkında hiçbir şey bilmeden (sadece bir `Object` referansı olarak) öğrenebildiğini düşün:

```
Object obj = "merhaba";
System.out.println(obj.getClass().getName()); // java.lang.String
```

Bu, aslında hepimizin defalarca kullandığı en basit reflection örneği — `getClass()`. Reflection API bunu çok daha ileri götürür: bir sınıfın hangi alanlara, metotlara ve constructor'lara sahip olduğunu listeleyebilir, `private` üyelere bile erişebilir, hatta parametrelerini bilmediğin bir constructor'ı çağırıp yeni bir nesne oluşturabilirsin — tüm bunları, ilgili sınıfı derleme zamanında hiç `import` etmeden, sadece ismini (örneğin bir string olarak) bilerek yapabilirsin.

Neden Var?

Java, statik tipli (statically typed) bir dildir — derleyici, her değişkenin tipini derleme zamanında bilir ve buna göre kontrol eder. Bu güçlü bir güvenlik ağı sağlar, ama bazı gerçek dünya problemleri tam tersini gerektirir: bir framework'ün, senin hiç bilmediği bir sınıfı (kullanıcının yazdığı bir DTO, bir controller, bir entity) çalışma zamanında keşfedip onun alanlarını doldurması, metotlarını çağırması, annotation'larına göre davranış değiştirmesi

gerekir. Spring'in `@Autowired` ile bağımlılık enjekte etmesi, Jackson'ın bir JSON'u senin sınıfına dönüştürmesi, JUnit'in `@Test` işaretli metotları bulup çalıştırması — bunların hiçbiri, o kütüphanenin senin sınıflarını derleme zamanında "bilmesiyle" mümkün değil. Reflection tam olarak bu boşluğu dolduruyor: kütüphane, tipleri isim ve annotation üzerinden çalışma zamanında keşfedip, generic bir mekanizmayla onlarla çalışabiliyor. Bu framework'lerin her birine, ilerleyen "Gerçek Dünya Kullanım Alanları" bölümünde tek tek değineceğiz.

Tarihçe

Reflection, Java'ya çok erken bir dönemde, JDK 1.1 (1997) ile `java.lang.reflect` paketi altında eklendi — dilin en eski API'lerinden biri. O zamandan beri altyapısı büyük ölçüde aynı kalsa da, etrafına önemli eklemeler yapıldı: Java 5, generic tip bilgisini reflection'a taşıdı; Java 7, `java.lang.invoke` paketiyle `MethodHandle` mekanizmasını getirdi — klasik reflection'a göre çok daha performanslı, modern bir alternatif ("Performans Değerlendirmeleri" bölümünde detaylandıracağız); Java 9'daki modül sistemi (JPMS), reflection'ın *her şeye* erişebilmesini kısıtladı — bir modül kendi iç paketlerini açıkça `opens` etmediği sürece, dışarıdan `setAccessible(true)` ile bile erişilemez hale geldi ("Güvenlik Değerlendirmeleri" bölümünde işleyeceğiz). Bu projede kullandığımız Java 21, tüm bu katmanları (klasik reflection + `MethodHandle` + modül kısıtlamaları) bir arada barındırıyor.

Class Nesnesi Elde Etmek

Reflection'daki her şey bir `java.lang.Class` nesnesinden başlar — bu nesne, çalışma zamanında bir tipin "künyesini" temsil eder: adı, alanları, metotları, constructor'ları, hepsi buradan erişilir. Bir `Class` nesnesi elde etmenin üç yolu var:

```
class Book {
    private String title;
    private String author;
    private int pages;

    Book() {
        this("Untitled", "Unknown", 0);
    }

    Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }

    @Override
    public String toString() {
        return "Book{title='" + title + "', author='" + author + "', pages=" + pages + "}";
    }
}

class ThreeWaysToGetClass {
    public static void main(String[] args) throws ClassNotFoundException {
        Book book = new Book("Effective Java", "Joshua Bloch", 412);

        Class<?> fromInstance = book.getClass();
        Class<?> fromLiteral = Book.class;
        Class<?> fromName = Class.forName("Book");

        System.out.println(fromInstance.getName());       // Book
        System.out.println(fromInstance == fromLiteral);  // true
        System.out.println(fromInstance == fromName);     // true
    }
}
```

Beklenen Çıktı

```
Book
true
true
```

- `nesne.getClass()` — elinde zaten bir örnek varsa, en doğal yol.
- `TipAdı.class` — bir örneğe ihtiyaç duymadan, derleme zamanında sabit bir referans; tip adını doğrudan biliyorsan (ki genelde biliyorsundur) en temiz yol budur.
- `Class.forName("TipAdı")` — tip adını yalnızca bir **string** olarak bildiğin durumlar için (örneğin bir config dosyasından okunan sınıf adı); framework'lerin en çok kullandığı yöntem tam olarak budur.

Üçü de aynı `Class` nesnesini döner — bunun nedeni, her sınıf için JVM'in kendi classloader'ı başına yalnızca **tek bir** `Class` örneği tutması: `fromInstance == fromLiteral` karşılaştırması bu yüzden `true` döner, `equals()`'a bile gerek yok.

Primitive tiplerin de kendi `Class` nesneleri vardır — `int.class`, `boolean.class`, `void.class` gibi. Bunlar, karşılık gelen kutulanmış (boxed) tipin `Class`'ından **farklıdır**: `int.class != Integer.class`. Reflection API'sinde parametre tipi eşleştirirken bu ayrımı unutmak, sık karşılaşılan bir `NoSuchMethodException` sebebidir.

`Class.forName(String)`, ilgili sınıfı yalnızca "bulmakla" kalmaz — varsayılan olarak sınıfı **yükler ve initialize eder** (static bloklar, static alan initializer'ları çalışır). Bir sınıfı sadece incelemek istiyorsan ama onu initialize etmek istemiyorsan (örneğin static bloğunun yan etkilerinden kaçınmak için), üç parametreli aşırı yüklenmiş halini kullan: `Class.forName(name, false, classLoader)`.

Sınıf Bilgisini İncelemek

Bir `Class` nesnesi elde ettikten sonra, o tip hakkında zengin bir bilgi kümesine erişebilirsin:

```
import java.lang.reflect.Modifier;

class Book {
    private String title;
    private String author;
    private int pages;

    Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }
}

class ClassInfoExample {
    public static void main(String[] args) {
        Class<Book> type = Book.class;

        System.out.println(type.getName());        // Book
        System.out.println(type.getSimpleName());   // Book
        System.out.println(type.getPackageName());  // (empty string — default package)
        System.out.println(type.isInterface());     // false
        System.out.println(type.isRecord());        // false
        System.out.println(Modifier.isPublic(type.getModifiers())); // false — package-
private
    }
}
```

Beklenen Çıktı

```
Book
Book

false
false
false
```

`getName()` ve `getSimpleName()` arasındaki fark, iç içe (nested) sınıflarda ve dizilerde belirginleşir — `getName()` her zaman JVM'in iç temsilini (nested sınıflar için `$` ayracıyla, diziler için `[` öneki ile) döner, `getSimpleName()` ise kaynak kodunda yazdığın gibi okunaklı ismi verir. `getPackageName()`'in, varsayılan pakette (bu projedeki örneklerde olduğu gibi) boş string döndüğüne dikkat et.

Sınıf hiyerarşisini ve implement edilen arayüzleri de `Class` üzerinden sorgulayabilirsin — herkesin tanıdığı `ArrayList` üzerinde:

```
Class<?> listType = java.util.ArrayList.class;
System.out.println(listType.getSuperclass());
// class java.util.AbstractList
System.out.println(java.util.Arrays.toString(listType.getInterfaces()));
// [interface java.util.List, interface java.util.RandomAccess, interface java.lang.
Cloneable, interface java.io.Serializable]
```

`Modifier.isPublic(type.getModifiers())`, `Modifier.isFinal(...)`, `Modifier.isAbstract(...)` gibi statik yardımcı metotlar, `getModifiers()`'ın döndürdüğü ham `int` bit maskesini okunaklı boolean sorgulara çevirir — bit işlemlerini elle yapmana hiç gerek yok.

`isInstance(Object)` ve `isAssignableFrom(Class)` metotlarını `instanceof` operatörüyle karıştırma: `type.isInstance(obj)`, `obj instanceof Type` ile aynı anlama gelir (bir **nesnenin** bir tipe uyup uymadığını sorar); `typeA.isAssignableFrom(typeB)` ise iki **tip**'in birbirine atanabilir olup olmadığını sorar (`typeB`'nin bir örneği `typeA`'ya atanabilir mi?). Yönü ters çevirmek, sessizce yanlış sonuç veren çok yaygın bir hatadır.

Alanları (Fields) Okumak

Bir sınıfın alanlarını listelemek için iki metottan biri kullanılır, ve aralarındaki fark sık karıştırılır:

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

class Book {
    public static final String CATEGORY = "Literature";
    private String title;
    private String author;
    private int pages;

    Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }
}

class FieldsExample {
    public static void main(String[] args) throws Exception {
        Class<Book> type = Book.class;

        System.out.println("getFields():");
        for (Field f : type.getFields()) {
            System.out.println("  " + f.getName());
        }
        // getFields(): CATEGORY

        System.out.println("getDeclaredFields():");
        for (Field f : type.getDeclaredFields()) {
            System.out.println("  " + f.getName() + " : " + f.getType().getSimpleName());
        }
        // getDeclaredFields(): CATEGORY : String, title : String, author : String, pages :
int

        Field categoryField = type.getField("CATEGORY");
        System.out.println(categoryField.get(null)); // Literature — static field, no
instance needed
    }
}
```

`getFields()`, yalnızca **public** alanları döner — ama bunlara üst sınıflardan/ arayüzlerden miras alınanlar da dahildir. `getDeclaredFields()` ise tam tersi: yalnızca **o sınıfın kendisinde tanımlanan** alanları döner (erişim belirleyicisi ne olursa olsun — `private` dahil), ama üst sınıftan miras alınanları **içermez**. Bu iki metodun "public mi değil mi" ve "miras mı değil mi" eksenlerinde birbirinin tam tersi davrandığını hatırlamak, hangisini seçeceğine karar verirken en pratik yoldur.

`Field.get(Object)`, bir alanın değerini okur; alan `static` ise parametre olarak `null` geçilebilir (yukarıdaki örnekte `CATEGORY` için yaptığımız gibi) — `static` olmayan bir alan için gerçek bir örnek vermek zorundasın.

> Yukarıdaki örnekte yalnızca **public** bir alanı (`CATEGORY`) okuduk — `private` bir alanı `get()` ile okumaya çalışsaydın, `IllegalAccessException` alırdın. Bunun çözümünü ("Private Alan ve Metotlara Erişmek" bölümünde işleyeceğiz) `setAccessible(true)` çağırmak oluşturuyor, ama bunu şimdilik erteliyoruz — önce "sadece keşfetmeyi" ayırt etmek önemli.

Metotları Okumak

Metotlar için de aynı ikili örüntü geçerli — `getMethods()` / `getDeclaredMethods()` :

```
import java.lang.reflect.Method;

class Book {
    private String title;

    Book(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }

    private boolean isLong() {
        return title.length() > 50;
    }
}

class MethodsExample {
    public static void main(String[] args) {
        Class<Book> type = Book.class;

        System.out.println("getMethods() count: " + type.getMethods().length);
        // includes getTitle() plus everything inherited from Object (toString, equals,
hashCode, ...)

        System.out.println("getDeclaredMethods():");
        for (Method m : type.getDeclaredMethods()) {
            System.out.println("  " + m.getName() + " -> " + m.getReturnType().
getSimpleName());
        }
        // getTitle -> String, isLong -> boolean (declared here, private included,
inherited excluded)
    }
}
```

`getMethods()` , sınıfın **public** metotlarını döner — kendi tanımladıkların **ve** `Object` 'ten miras aldığın `toString()` , `equals()` , `hashCode()` gibi metotlar dahil (bu yüzden basit bir sınıf için bile listenin boyutu göründüğünden büyük çıkar). `getDeclaredMethods()` , yalnızca o sınıfın **kendi gövdesinde** tanımladığı metotları döner — erişim belirleyicisi fark etmez (`private` dahil), ama miras alınanlar hariçtir.

Her bir `Method` nesnesi üzerinden `getReturnType()`, `getParameterTypes()`, `getParameterCount()` gibi metotlarla imzayı tamamen keşfedebilirsin — bir sonraki iki bölümde bu bilgiyi, önce nesne oluşturmak (constructor'lar için), sonra da bu metotları gerçekten **çağırmak** için kullanacağız.

> Aşırı yüklenmiş (overloaded) bir metodu ada göre değil, tam imzasına göre almak istiyorsan `getMethod(String, Class<?>...)` kullan — örneğin `getMethod("process", String.class)` ile `getMethod("process", int.class)`, aynı isimli iki farklı metodu ayırt eder. Parametre tipi belirtmeden yalnızca isimle arama yapan bir metot yoktur, çünkü Java'da overload çözümü tipe göre yapılır.

Constructor'ları Okumak

Constructor'lar, metotlara çok benzer şekilde keşfedilir — `getConstructors()` (public olanlar) ve `getDeclaredConstructors()` (erişim belirleyicisi fark etmeksizin hepsi):

```
import java.lang.reflect.Constructor;
import java.util.Arrays;

class Book {
    private String title;
    private String author;
    private int pages;

    Book() {
        this("Untitled", "Unknown", 0);
    }

    Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }
}

class ConstructorsExample {
    public static void main(String[] args) {
        Class<Book> type = Book.class;

        for (Constructor<?> c : type.getDeclaredConstructors()) {
            System.out.println(c.getParameterCount() + " parameter(s): "
                    + Arrays.toString(c.getParameterTypes()));
        }
        // 0 parameter(s): []
        // 3 parameter(s): [class java.lang.String, class java.lang.String, int]
    }
}
```

Her `Constructor` nesnesi, `getParameterCount()` ve `getParameterTypes()` ile kendi imzasını açığa çıkarır — aynı sınıfın birden fazla (overload edilmiş) constructor'ı varsa, doğru olanı

parametre tiplerine göre ayırt edip seçebilirsin. Bir sonraki bölümde tam olarak bunu yapacağız: seçtiğimiz constructor'ı kullanarak gerçek bir nesne üreteceğiz.

Nesneleri Dinamik Olarak Oluşturmak

Bir constructor'ı bulduktan sonra, `newInstance(Object...)` ile gerçekten çağırıp yeni bir nesne üretebilirsin:

```
import java.lang.reflect.Constructor;

class Book {
    private String title;

    Book(String title) {
        this.title = title;
    }

    @Override
    public String toString() {
        return "Book{title='" + title + "'}";
    }
}

class DynamicObjectCreation {
    public static void main(String[] args) throws Exception {
        Class<Book> type = Book.class;

        Constructor<Book> constructor = type.getDeclaredConstructor(String.class);
        Book book = constructor.newInstance("Domain-Driven Design");

        System.out.println(book); // Book{title='Domain-Driven Design'}
    }
}
```

Beklenen Çıktı

```
Book{title='Domain-Driven Design'}
```

`getDeclaredConstructor(Class<?>...)`, parametre tiplerine göre doğru constructor'ı bulur; `newInstance(...)` da o constructor'ı, verdiğin argümanlarla çağırıp yeni bir nesne döner — tıpkı `new Book(...)` yazmış gibi, ama tip adını (`Book`) derleme zamanında hiç bilmeden.

> `Constructor.newInstance(...)` ile karıştırılmaması gereken, artık **deprecated** olan eski bir API var: `Class.newInstance()` (parametresiz). Java 9'dan beri kullanımdan kaldırıldı (deprecated), çünkü iki önemli sorunu var: (1) constructor'ın fırlattığı **checked** exception'ları sarmalamadan doğrudan fırlatıyor — bu da derleyicinin checked exception kontrolünü etkin bir şekilde by-pass ediyor; (2) `private` / `protected` constructor'lara erişim kontrolünü, `Constructor.newInstance()` kadar tutarlı yapmıyor. Yeni kodda her zaman `getDeclaredConstructor(...)` + `Constructor.newInstance(...)` kullan.

Metotları Dinamik Olarak Çağırmak

`Method.invoke(Object, Object...)`, bulduğun bir metodu, verdiğin nesne üzerinde, verdiğin argümanlarla çalıştırır:

```
import java.lang.reflect.Method;

class Book {
    private String title;

    Book(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }

    public static String describe() {
        return "A Book represents a published work.";
    }
}

class DynamicMethodInvocation {
    public static void main(String[] args) throws Exception {
        Book book = new Book("Refactoring");
        Class<Book> type = Book.class;

        Method getTitle = type.getMethod("getTitle");
        System.out.println(getTitle.invoke(book)); // Refactoring

        Method describe = type.getMethod("describe");
        System.out.println(describe.invoke(null)); // static method -> no target instance
needed
    }
}
```

> **Beklenen Çıktı**
>
> ```
> Refactoring
> A Book represents a published work.
> ```

İlk çağrıda, `getTitle` instance metodunu `book` nesnesi üzerinde çalıştırdık — ilk parametre (`invoke`'un hedef nesnesi) budur. İkinci çağrıda ise `describe()` bir **static** metot olduğu için hedef nesne olarak `null` geçtik — static metotların çağrılması için bir örneğe ihtiyaç yoktur, tıpkı static alan okumada olduğu gibi ("Alanları Okumak" bölümünde gördüğümüz `CATEGORY` örneğiyle aynı mantık).

> Çağırdığın metot bir exception fırlatırsa, `invoke()` bunu doğrudan fırlatmaz — orijinal exception'ı **`InvocationTargetException`** içine sarmalayıp onu fırlatır; gerçek hatayı görmek için `getCause()` ile sarmalanmış exception'ı açman gerekir. Bu sarmalamayı unutup doğrudan beklediğin exception tipini yakalamaya çalışmak, sık karşılaşılan bir reflection hatasıdır:

```
try {
    method.invoke(target, args);
} catch (java.lang.reflect.InvocationTargetException e) {
    Throwable realCause = e.getCause();
    // asıl hata burada, e'nin kendisinde değil
}
```

Private Alan ve Metotlara Erişmek

Buraya kadar yalnızca **public** üyelerle çalıştık. Ama reflection'ın belki de en çok bilinen (ve en çok kötüye kullanılan) özelliği, `private` alan ve metotlara da erişebilmesidir — `AccessibleObject.setAccessible(true)` ile:

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

class Book {
    private String title = "Untitled";

    private boolean isLong() {
        return title.length() > 50;
    }
}

class PrivateAccessExample {
    public static void main(String[] args) throws Exception {
        Book book = new Book();
        Class<Book> type = Book.class;

        Field titleField = type.getDeclaredField("title");
        titleField.setAccessible(true);
        titleField.set(book, "Effective Java");
        System.out.println(titleField.get(book)); // Effective Java

        Method isLong = type.getDeclaredMethod("isLong");
        isLong.setAccessible(true);
        System.out.println(isLong.invoke(book)); // false
    }
}
```

Beklenen Çıktı

```
Effective Java
false
```

`setAccessible(true)`, Java'nın normalde derleme zamanında uyguladığı erişim kontrolünü (private/protected/package-private) reflection çağrıları için devre dışı bırakır — hem `Field.get()` / `Field.set()` hem de `Method.invoke()` için geçerlidir. Bu satır olmadan yukarıdaki örnek `IllegalAccessException` fırlatırdı.

> `setAccessible(true)`, Java 9'daki modül sistemi (JPMS) ile artık **her zaman** çalışmıyor — "Tarihçe" bölümünde değindiğimiz gibi, hedef sınıf adlandırılmış bir modülün içindeyse ve o modül ilgili paketi `opens` ile açıkça açmamışsa, `setAccessible(true)` çağrısı `InaccessibleObjectException` fırlatır. Bu kısıtlamayı ve nasıl yönetileceğini "Güvenlik Değerlendirmeleri" bölümünde derinlemesine işleyeceğiz — bu proje modülsüz (classpath tabanlı) çalıştığı için burada sorunsuz çalışıyor, ama gerçek bir modüler uygulamada dikkat etmen gereken bir noktadır.

> `setAccessible(true)`'ı yalnızca gerçekten ihtiyacın olan tek bir `Field` / `Method` nesnesi üzerinde, mümkün olduğunca dar bir kapsamda çağır — bir sınıfın tüm private üyelerini toplu olarak "erişilebilir" yapıp geniş çaplı kullanmak, hem okunabilirliği hem de kodun test edilebilirliğini zayıflatır. Reflection'la private üyelere erişmek genelde bir framework /test altyapısı ihtiyacıdır, iş mantığında (business logic) tercih edilecek bir yol değildir.

Annotation'larla Çalışmak

Bir annotation'ı reflection ile okuyabilmen için, o annotation'ın **çalışma zamanına kadar hayatta kalması** gerekir — bu, annotation tanımındaki `@Retention` ile belirlenir:

```
@Retention(RetentionPolicy.SOURCE)   // yalnızca kaynak kodda, derleyici sonrası kaybolur — örn. @Override
@Retention(RetentionPolicy.CLASS)    // .class dosyasında kalır ama JVM çalışırken erişilemez (varsayılan)
@Retention(RetentionPolicy.RUNTIME)  // çalışma zamanında reflection ile okunabilir — bize gereken bu
```

`@Override` iyi bilinen bir örnek: `RetentionPolicy.SOURCE` ile işaretlidir, yalnızca derleyicinin "bu metot gerçekten bir üst sınıf metodunu override ediyor mu?" kontrolü için var olur — derlenmiş `.class` dosyasında hiçbir izi kalmaz, reflection ile hiçbir zaman göremezsin.

`RetentionPolicy.RUNTIME` ile işaretlenmiş bir annotation'ı, `Class`, `Field`, `Method` ve `Constructor` üzerinde (hepsi `AnnotatedElement` arayüzünü implement eder) aynı metotlarla okuyabilirsin:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.reflect.Method;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Auditable {
    String value() default "";
}

class AccountService {
    @Auditable("balance-check")
    void checkBalance() {
        // ...
    }

    void internalHelper() {
        // not annotated
    }
}

class AnnotationExample {
    public static void main(String[] args) throws Exception {
        Method checkBalance = AccountService.class.getDeclaredMethod("checkBalance");
        Method internalHelper = AccountService.class.getDeclaredMethod("internalHelper");

        System.out.println(checkBalance.isAnnotationPresent(Auditable.class));   // true
        System.out.println(internalHelper.isAnnotationPresent(Auditable.class)); // false

        Auditable auditable = checkBalance.getAnnotation(Auditable.class);
        System.out.println(auditable.value()); // balance-check
    }
}
```

Beklenen Çıktı

```
true
false
balance-check
```

`getAnnotations()` ile `getDeclaredAnnotations()` arasındaki fark, alan/metot okumadaki `getFields()` / `getDeclaredFields()` ayrımına benzemez — burada devreye giren annotation'ın kendisinin `@Inherited` ile işaretlenip işaretlenmediğidir. `@Inherited` **yalnızca sınıf düzeyindeki** annotation'lar için anlamlıdır ve varsayılan olarak kapalıdır: bir annotation `@Inherited` değilse, alt sınıf onu üst sınıftan miras almaz, `getAnnotations()` de göstermez.

> Bir annotation'ı tanımlarken `@Retention`'ı unutmak (ya da varsayılan `CLASS`'ta bırakmak), en sık karşılaşılan reflection tuzaklarından biridir — kod derlenir, annotation kaynak kodda "duruyor gibi görünür", ama `isAnnotationPresent()` her zaman `false` döner ve saatlerce "neden görünmüyor?" diye debug edersin. Reflection ile okumayı planladığın her annotation'da `@Retention(RetentionPolicy.RUNTIME)` yazdığını mutlaka kontrol et.

Gerçek Dünya Kullanım Alanları

Buraya kadar gördüğümüz her parça — `Class` keşfi, dinamik nesne oluşturma, dinamik metot çağırma, annotation okuma — gerçek framework'lerin temel çalışma mekanizmasını oluşturur:

- **Spring**, classpath'i tarayıp `@Component` / `@Service` / `@Controller` ile işaretli sınıfları bulur (annotation okuma), her birinin constructor'ını inceleyip parametre tiplerine göre bağımlılıkları çözer (`@Autowired` — "Nesneleri Dinamik Olarak Oluşturmak" bölümünde gördüğümüz `Constructor.newInstance()` mantığının ölçeklendirilmiş hali), `@Value` / `@ConfigurationProperties` ile alanlara değer atar (`Field.set()`).
- **Hibernate/JPA**, bir entity'nin alanlarını inceleyip veritabanı kolonlarıyla eşler, sorgu sonuçlarından nesneleri `private` alanlara doğrudan yazarak doldurur (genelde parametresiz constructor + `setAccessible(true)` + `Field.set()` kombinasyonuyla), lazy loading için çalışma zamanında senin entity sınıfını extend eden bir proxy sınıfı bile üretebilir.
- **Jackson**, bir JSON'u senin sınıfına dönüştürürken alanları/setter'ları (`Field` / `Method`) reflection ile bulur — "Gerçek Dünya Örnekleri" bölümünde (Record konusunda) gördüğümüz gibi, bir record için doğrudan canonical constructor'ı tanıyıp kullanır.
- **JUnit**, `@Test` ile işaretli metotları `getDeclaredMethods()` + annotation kontrolüyle bulur, her birini `Method.invoke()` ile teker teker çalıştırır. Bu mekanizmayı, çok sadeleştirilmiş bir haliyle kendimiz de yazabiliriz:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.reflect.Method;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyTest {
}

class Calculator {
    @MyTest
    void additionWorks() {
        if (2 + 2 != 4) {
            throw new AssertionError("Addition is broken");
        }
        System.out.println("additionWorks: PASSED");
    }

    @MyTest
    void subtractionWorks() {
        if (5 - 3 != 2) {
            throw new AssertionError("Subtraction is broken");
        }
        System.out.println("subtractionWorks: PASSED");
    }

    void notATest() {
        System.out.println("This should never run");
    }
}

// A drastically simplified sketch of what JUnit does internally: scan for
// annotated methods, then invoke each one reflectively.
class MiniTestRunner {
    public static void main(String[] args) throws Exception {
        Calculator target = new Calculator();

        for (Method method : Calculator.class.getDeclaredMethods()) {
            if (method.isAnnotationPresent(MyTest.class)) {
                method.invoke(target);
            }
        }
    }
}
```

Bu dört framework'ün ortak noktası şu: **senin** yazdığın sınıfları, kendi kaynak kodlarına hiç `import` etmeden, çalışma zamanında keşfedip kullanıyorlar — "Neden Var?" bölümünde tanımladığımız problemin, üretim kalitesindeki kütüphanelerde tam olarak nasıl çözüldüğünü şimdi somut olarak gördün.

Performans Değerlendirmeleri

Reflection çağrıları, doğrudan çağrılara göre her zaman daha yavaştır — üç ana sebepten: (1) her çağrıda erişim kontrolü yapılır (`setAccessible(true)` bunu atlar, ama tamamen değil); (2) primitive parametreler `Object[]` içine kutulanıp (autoboxing) geri çözülmek zorundadır; (3) JIT derleyicisi, `Method.invoke()` gibi tek bir generic çağrı noktasını, doğrudan bir metot çağrısı kadar agresif optimize edemez.

Bunun pratikteki karşılığı iki tavsiye:

Aramaları önbelleğe al. `getMethod()` / `getDeclaredField()` gibi arama işlemleri, `invoke()` / `get()` 'in kendisinden daha maliyetlidir — bir `Method` / `Field` nesnesini bir kez bulup bir değişkende/ `static final` alanda saklamak, her çağrıda yeniden aramaktan çok daha iyidir.

Sık çağrılan bir yol için `MethodHandle`'ı değerlendir. Java 7 ile gelen `java.lang.invoke` paketi, klasik reflection'a göre daha modern bir alternatif sunar — `MethodHandles.Lookup` ile bir kez çözülüp elde edilen bir `MethodHandle`, JIT tarafından çok daha iyi optimize edilebilir (tekrarlanan çağrılarda doğrudan çağrıya yaklaşan bir performansa ulaşabilir):

```
import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodType;
import java.lang.reflect.Method;

class Greeter {
    public String greet(String name) {
        return "Hello, " + name + "!";
    }
}

class MethodHandleExample {
    public static void main(String[] args) throws Throwable {
        Greeter greeter = new Greeter();

        // Classic reflection
        Method method = Greeter.class.getMethod("greet", String.class);
        System.out.println(method.invoke(greeter, "Ada")); // Hello, Ada!

        // MethodHandle — resolved once via a Lookup, then invoked directly.
        // The JIT can optimize a MethodHandle call site much more aggressively
        // than a classic Method.invoke() call.
        MethodHandles.Lookup lookup = MethodHandles.lookup();
        MethodHandle handle = lookup.findVirtual(Greeter.class, "greet",
                MethodType.methodType(String.class, String.class));
        System.out.println((String) handle.invoke(greeter, "Grace")); // Hello, Grace!
    }
}
```

> **Beklenen Çıktı**

```
Hello, Ada!
Hello, Grace!
```

Java 9, `VarHandle`'ı da ekledi — `MethodHandle`'ın alan erişimine (get/set, hatta atomik compare-and-swap işlemlerine) odaklanan kardeşi; eskiden `sun.misc.Unsafe` ile yapılan düşük seviye alan manipülasyonlarının resmî, desteklenen yerini alıyor.

> Reflection ile doğrudan çağrıyı, basit bir döngü + `System.currentTimeMillis()` ile "benchmark etmeye" kalkışma — JIT'in ısınma (warmup) süreci, dead-code elimination gibi optimizasyonlar, birkaç satırlık naif bir ölçümü kolayca anlamsız hale getirir. Gerçek bir performans karşılaştırması için JMH (Java Microbenchmark Harness) kullan; bu derste performans farkının **var olduğunu** ve **neden** var olduğunu anlamak yeterli, kesin sayılar için asla naif bir döngüye güvenme.

> Pratikte, uygulama kodunun büyük çoğunluğu bu optimizasyonlarla hiç uğraşmaz — Spring, Hibernate gibi framework'ler bu önbellekleme ve `MethodHandle` kullanımını zaten kendi içlerinde yapıyor. Bunu bilmen gereken asıl durum, kendi reflection tabanlı bir araç (bir mini framework, bir test runner, bir serializer) yazıyor olman.

Güvenlik Değerlendirmeleri

"Private Alan ve Metotlara Erişmek" bölümünde değindiğimiz `InaccessibleObjectException`, aslında Java 9'daki modül sisteminin (JPMS) bilinçli bir güvenlik/encapsulation kararı. Bir modül, kendi paketini reflection'a açmak istiyorsa, bunu `module-info.java` içinde açıkça belirtmek zorunda:

```
module com.example.app {
    opens com.example.app.internal to some.reflecting.module;
}
```

Modülü değiştiremediğin durumlarda (örneğin üçüncü parti bir kütüphaneyse), aynı etkiyi JVM başlatma parametresiyle de elde edebilirsin:

```
java --add-opens com.example.app/com.example.app.internal=ALL-UNNAMED -jar app.jar
```

Daha eski Java sürümlerinde reflection erişimini kısıtlamanın bir başka yolu da `SecurityManager`'dı (`ReflectPermission` kontrolleriyle) — ama bu mekanizma Java 17'den itibaren kullanımdan kaldırılmaya (deprecated for removal) başladı, yeni tasarımlarda buna güvenmemelisin; JPMS'in `opens` kısıtlaması artık önerilen yol.

Reflection'ın kendisi de bir saldırı yüzeyi oluşturabilir: özellikle Java deserialization ile ilgili bilinen güvenlik açıklarının çoğu ("Serialization ve Reflection" bölümünde Record konusunda

değindiğimiz deserialization mekanizmasını hatırla), saldırganın kontrol ettiği bir byte akışının, reflection ile rastgele metotları zincirleme çağırmasına (gadget chain) dayanır.

> `Class.forName(String)`'e **kullanıcıdan gelen bir string'i** doğrudan geçirmekten kaçın — bu, saldırganın classpath'teki herhangi bir sınıfı (hatta bazı durumlarda istenmeyen yan etkileri olan static initializer'ları tetikleyerek) yüklemesine izin verebilir. Sınıf adı dışarıdan geliyorsa, mutlaka bilinen/izin verilen bir liste (allow-list) ile doğrula.

Best Practices

- Reflection'ı **son çare** olarak gör — sıradan iş mantığında neredeyse hiçbir zaman gerekmez; asıl yeri framework/library/test-altyapısı kodudur.
- `Method` / `Field` / `Constructor` aramalarını önbelleğe al ("Performans Değerlendirmeleri" bölümünde detaylandırdık).
- Sık çağrılan bir yol için klasik reflection yerine `MethodHandle`'ı değerlendir.
- `setAccessible(true)`'ı yalnızca gerçekten gereken tek üye üzerinde, dar bir kapsamda çağır ("Private Alan ve Metotlara Erişmek" bölümünde değindik).
- Dışarıdan gelen sınıf/metot adlarını asla doğrulamadan reflection'a besleme ("Güvenlik Değerlendirmeleri" bölümü).
- Reflection kullanımını kodda **belirgin** hale getir (yorum, isimlendirme) — bir IDE'nin "rename" refactor'ı, `getDeclaredField("title")` gibi bir string literal'i **asla** güncellemez; bir alanı yeniden adlandırdığında, derleme hiçbir hata vermez ama reflection çağrısı çalışma zamanında sessizce `NoSuchFieldException` fırlatır.

> Bir sınıfın `private` üyelerine reflection ile sürekli erişme ihtiyacı duyuyorsan, bu genellikle bir tasarım kokusu (design smell) işaretidir — o sınıfın public API'si, gerçekte ihtiyaç duyulanı karşılamıyor olabilir. Reflection'ı bir "etrafından dolaşma" aracı olarak değil, gerçekten genel amaçlı bir mekanizmaya (framework, test, serializer) ihtiyaç duyduğunda kullan.

Yaygın Hatalar

1. `IllegalAccessException`'ı unutup `private` bir üyeye doğrudan erişmeye çalışmak. Önce `setAccessible(true)` çağırman gerektiğini unutma (bkz. "Private Alan ve Metotlara Erişmek").

2. `InvocationTargetException`'ı unutup çağrılan metodun fırlattığı asıl exception'ı yakalamaya çalışmak. `invoke()` her zaman gerçek hatayı sarmalar; `getCause()` ile açman gerekir (bkz. "Metotları Dinamik Olarak Çağırmak").

3. Deprecated `Class.newInstance()`'ı kullanmaya devam etmek. Yerine `getDeclaredConstructor(...)` + `Constructor.newInstance(...)` kullan (bkz. "Nesneleri Dinamik Olarak Oluşturmak").

4. Generic tip bilgisinin çalışma zamanında silindiğini (type erasure) unutmak. `List<String>` bir listesi, çalışma zamanında yalnızca `List` olarak görünür — reflection, generic parametre tipini (`String`) çoğu durumda göremez. `getGenericType()` gibi metotlar bazı sınırlı bilgi verse de (örneğin bir alanın bildirimindeki generic imza üzerinden), gerçek çalışma zamanı nesnesinin generic tipini garantili şekilde öğrenmenin bir yolu yoktur.

5. Refactoring sonrası reflection'ın derleme zamanında yakalanmayan bir kırılganlığa sahip olduğunu unutmak. Bir alanı ya da metodu IDE ile yeniden adlandırmak (rename), `getDeclaredField("eskiAd")` gibi bir string literal'i **güncellemez** — kod sorunsuz derlenir, ama çalışma zamanında `NoSuchFieldException` / `NoSuchMethodException` alırsın.

6. Sıcak bir döngüde (hot path) her seferinde `Method` / `Field` aramasını tekrarlamak. `getMethod()`'u her çağrıdan önce yeniden çağırmak yerine, bir kez bulup önbelleğe al (bkz. "Performans Değerlendirmeleri").

Özet, Cheat Sheet ve Terimler Sözlüğü

Reflection, Java'nın JDK 1.1'den beri sahip olduğu, bir programın kendi tiplerini çalışma zamanında inceleyip dinamik olarak kullanmasını sağlayan API'dir. Öne çıkan noktalar:

- Her şey bir `Class` nesnesinden başlar — `getClass()`, `.class` ya da `Class.forName()` ile elde edilir, JVM başına tek bir örnek garanti edilir
- `getX()` (public + miras alınan) / `getDeclaredX()` (yalnızca o sınıfa ait, erişim belirleyicisi fark etmeksizin) ikilisi; alan, metot ve constructor'lar için aynı desen geçerlidir
- Nesne oluşturmak için `Constructor.newInstance(...)` (deprecated `Class.newInstance()` değil), metot çağırmak için `Method.invoke(hedef, argümanlar...)`
- `private` üyelere `setAccessible(true)` ile erişilir — ama Java 9+ modül sisteminde bu her zaman garanti değildir (`InaccessibleObjectException`)
- Bir annotation'ı reflection ile okuyabilmek için `@Retention(RetentionPolicy.RUNTIME)` şart
- Spring, Hibernate, Jackson, JUnit — hepsi aynı temel mekanizmayı (keşfet, oluştur, çağır) kullanır
- Performans: aramaları önbelleğe al, sık çağrılan yollarda `MethodHandle`'ı değerlendir, asla naif benchmark'a güvenme
- Güvenlik: sınıf adlarını dışarıdan doğrulamadan almadan `Class.forName()`'e besleme, JPMS'in `opens` kısıtlamasına saygı duy

Hızlı referans:

```
// Class nesnesi elde etmek
Class<?> type = obj.getClass();
Class<?> type2 = MyClass.class;
Class<?> type3 = Class.forName("com.example.MyClass");

// Alan okumak/yazmak
Field field = type.getDeclaredField("fieldName");
field.setAccessible(true);
Object value = field.get(obj);
field.set(obj, newValue);

// Metot çağırmak
Method method = type.getMethod("methodName", ParamType.class);
Object result = method.invoke(obj, arg);

// Nesne oluşturmak
Constructor<?> constructor = type.getDeclaredConstructor(ParamType.class);
Object instance = constructor.newInstance(arg);

// Annotation okumak
if (method.isAnnotationPresent(MyAnnotation.class)) {
    MyAnnotation ann = method.getAnnotation(MyAnnotation.class);
}
```

Terimler Sözlüğü

Reflection — Bir programın çalışma zamanında kendi tiplerini inceleyip dinamik olarak kullanabilmesini sağlayan API (`java.lang.reflect` paketi).

`Class` **nesnesi** — Bir tipin çalışma zamanındaki temsili; her tip için JVM başına tek bir örnek vardır.

`AccessibleObject` — `Field`, `Method` ve `Constructor`'ın ortak üst sınıfı; `setAccessible(boolean)` metodunu buradan miras alırlar.

`setAccessible(true)` — Derleme zamanı erişim kontrolünü (private/protected/ package-private) reflection çağrıları için devre dışı bırakan çağrı.

`InvocationTargetException` — `Method.invoke()` sırasında çağrılan metodun fırlattığı asıl exception'ı sarmalayan wrapper exception; gerçek hataya `getCause()` ile ulaşılır.

`InaccessibleObjectException` — Java 9+ modül sisteminde, `opens` edilmemiş bir pakete `setAccessible(true)` ile erişmeye çalışıldığında fırlatılan exception.

`MethodHandle` — Java 7 ile gelen, klasik reflection'a göre JIT dostu, daha performanslı bir dinamik çağrı mekanizması (`java.lang.invoke` paketi).

`VarHandle` — Java 9 ile gelen, alan erişimi ve atomik işlemler için `MethodHandle` ailesinin bir parçası; eski `sun.misc.Unsafe` kullanımlarının yerini alır.

`AnnotatedElement` — `Class`, `Field`, `Method` ve `Constructor`'ın annotation okuma metotlarını (`getAnnotation()`, `isAnnotationPresent()` vb.) sağlayan ortak arayüz.

`RetentionPolicy.RUNTIME` — Bir annotation'ın çalışma zamanında reflection ile görülebilmesi için gereken retention politikası.

JPMS (Java Platform Module System) — Java 9 ile gelen modül sistemi; modüller, paketlerini reflection'a `opens` direktifiyle açıkça açmadıkça, dışarıdan `setAccessible(true)` ile bile erişilemez.

Ek: Mini Proje — Basit Dependency Injection Container

Şimdiye kadar öğrendiklerimizi ("Constructor'ları Okumak", "Nesneleri Dinamik Olarak Oluşturmak") birleştirip, Spring'in `@Autowired` ile yaptığı constructor injection'ın özünü, birkaç satırda kendimiz yazalım. Fikir basit: bir tipi "çözmek" (resolve) istediğinde, container önce onun constructor'ının hangi parametrelere ihtiyaç duyduğuna bakar, her parametreyi **recursive olarak** kendisi çözer, sonra hepsini constructor'a geçirip nesneyi oluşturur:

```
import java.lang.reflect.Constructor;
import java.util.HashMap;
import java.util.Map;

class Engine {
    String start() {
        return "Engine started";
    }
}

class Car {
    private final Engine engine;

    Car(Engine engine) {
        this.engine = engine;
    }

    String drive() {
        return engine.start() + " -> Car is driving";
    }
}

// A minimal constructor-injection container: given a type, it recursively
// resolves each constructor parameter, instantiates it, and caches the
// result as a singleton — the same core idea Spring's ApplicationContext
// uses, stripped down to its essence.
class SimpleContainer {
    private final Map<Class<?>, Object> singletons = new HashMap<>();

    @SuppressWarnings("unchecked")
    <T> T resolve(Class<T> type) throws ReflectiveOperationException {
        if (singletons.containsKey(type)) {
            return (T) singletons.get(type);
        }

        Constructor<?> constructor = type.getDeclaredConstructors()[0];
        Class<?>[] paramTypes = constructor.getParameterTypes();
        Object[] args = new Object[paramTypes.length];

        for (int i = 0; i < paramTypes.length; i++) {
            args[i] = resolve(paramTypes[i]);
        }

        Object instance = constructor.newInstance(args);
        singletons.put(type, instance);
        return (T) instance;
    }
}
```

```
class SimpleContainerDemo {
    public static void main(String[] args) throws ReflectiveOperationException {
        SimpleContainer container = new SimpleContainer();

        Car car = container.resolve(Car.class);
        System.out.println(car.drive()); // Engine started -> Car is driving
    }
}
```

`Car.class`'ı çözmek istediğimizde, container önce `Car`'ın tek constructor'ının bir `Engine` parametresi istediğini görüyor, `Engine`'i (hiçbir bağımlılığı olmadığı için doğrudan) oluşturuyor, sonra onu `Car`'ın constructor'ına geçirip `Car`'ı inşa ediyor — `Engine`'i sana hiç `new Engine()` yazdırmadan. Gerçek bir DI container (Spring dahil) bunun üzerine; scope yönetimi (singleton/prototype), döngüsel bağımlılık tespiti, arayüz-implementasyon eşlemesi gibi katmanlar ekler — ama çekirdek mekanizma tam olarak burada gördüğün recursive constructor çözümlemesidir.

> `resolve()` metodunun `throws ReflectiveOperationException` bildirdiğine dikkat et — bu, `InstantiationException`, `IllegalAccessException`, `InvocationTargetException` ve `NoSuchMethodException` gibi reflection'a özgü checked exception'ların **ortak üst sınıfıdır** (Java 7'den beri). Her birini ayrı ayrı yakalamak yerine, tek bir ortak tip üzerinden yönetmek, bu tür "birden fazla reflection exception'ı olabilir" kodunu önemli ölçüde sadeleştirir.

> Bu basit container, döngüsel bağımlılıkları (`A`, `B`'ye ihtiyaç duyuyor, `B` de `A`'ya) tespit etmiyor — böyle bir durumda `resolve()` sonsuz döngüye girip `StackOverflowError` ile çöker. Gerçek DI container'lar, çözümleme sürecinde "şu an inşa edilmekte olan tipler" kümesini takip edip, aynı tip ikinci kez karşımıza çıkarsa anlamlı bir hata fırlatır.

Ek: Mini Proje — Object Inspector

Son mini proje, elindeki **herhangi bir** nesneyi alıp, tüm alanlarını (isim, tip, değer) ve tüm metotlarını (imza) döken genel amaçlı bir araç — bu derste öğrendiğimiz hemen her şeyi (Class keşfi, alan/metot listeleme, `setAccessible`, `Modifier`) tek bir yerde birleştiriyor:

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

// A generic "dump everything you can see" utility: given any object, prints
// all of its declared fields (with their current values) and all of its
// declared methods (with their signatures) — regardless of access modifier.
class ObjectInspector {
    static void inspect(Object obj) throws IllegalAccessException {
        Class<?> type = obj.getClass();
        System.out.println("Class: " + type.getName());

        System.out.println("Fields:");
        for (Field field : type.getDeclaredFields()) {
            field.setAccessible(true);
            System.out.println("  " + Modifier.toString(field.getModifiers())
                    + " " + field.getType().getSimpleName()
                    + " " + field.getName()
                    + " = " + field.get(obj));
        }

        System.out.println("Methods:");
        for (Method method : type.getDeclaredMethods()) {
            System.out.println("  " + Modifier.toString(method.getModifiers())
                    + " " + method.getReturnType().getSimpleName()
                    + " " + method.getName() + "(...)");
        }
    }
}
```

Bu derste ilk bölümden beri kullandığımız `Book` sınıfını, bir kez daha, ama bu sefer tamamen dışarıdan (`Book`'un iç yapısını hiç bilmeyen bir araçla) inceleyelim:

```
class Book {
    private String title;
    private String author;
    private int pages;

    Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }

    String getTitle() {
        return title;
    }
}

class ObjectInspectorDemo {
    public static void main(String[] args) throws IllegalAccessException {
        Book book = new Book("Clean Architecture", "Robert C. Martin", 432);
        ObjectInspector.inspect(book);
    }
}
```

`inspect()` metodu, `Book`'un `private` alanlarını (`title`, `author`, `pages`) ve `getTitle()` metodunu, `Book`'un kaynak koduna tek satır `import` bile etmeden listeleyip değerlerini okuyabiliyor — bu dersin başından beri anlattığımız "kod, kendi kendini inceleyebilir" fikrinin, çalışan, elle dokunabildiğin bir örneği.

> `ObjectInspector`, IDE'lerin "Evaluate Expression" / değişken izleme (watch) panellerinin ve debugger'ların temelde yaptığı şeyin çok basitleştirilmiş bir versiyonu — bir debugger, durdurduğu her nesnenin alanlarını sana göstermek için de tam olarak bu mekanizmayı (reflection + `setAccessible`) kullanır.