Lambda Expressions

Lambda expression syntax'ının tamamı: parametre yazım kuralları, expression body vs block body, derleyicinin lambda'ya bağlamdan tip vermesi (target typing), effectively final değişken yakalama, ve lambda'nın anonymous inner class'tan farkı.

Orta 20 dk

Son güncelleme: 10 Kasım 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/lambda-expressions

---

Lambda Expressions

"Interface" dersinin "Functional Interface ve Lambda" bölümünde lambda'yı kısaca tanımıştık: tam olarak tek soyut metotlu bir interface'in (*functional interface*) ayrı bir sınıf yazmadan, doğrudan bir ifadeyle örneklenmesi. Bu ders, Java kursunun yeni kategorisi **Functional Interface'ler ve Stream'ler**'in ilk konusu -- orada bilerek kısa tutulan syntax'ı burada tam olarak açıyoruz: parametreler nasıl yazılır, gövde hangi iki biçimi alabilir, `return` ne zaman zorunlu, ve lambda dış scope'taki değişkenlere nasıl erişir. Bu kategori, kod örnekleri açısından bu kursun geri kalanından biraz farklı -- `java.util.function` / `java.util.stream` saf JDK olduğu için (Spring Boot'un aksine), harici bir bağımlılık gerektirmiyor.

Lambda Expression Nedir?

Bir lambda expression, adı olmayan, doğrudan bir değişkene atanabilen ya da bir metoda argüman olarak geçilebilen kısa bir fonksiyon tanımıdır. Java'da fonksiyonlar sınıflardan bağımsız var olamaz -- bir lambda da aslında bir sınıf değil, "Functional Interface ve Lambda" bölümünde gördüğümüz gibi, tam olarak tek soyut metotlu bir interface'in **anlık bir örneği**. `parametre -> gövde` biçimindeki bu kısa syntax, Java'ya (diğer birçok dilin zaten sahip olduğu) fonksiyonları **veri gibi taşıma** yeteneğini kazandırır -- bir fonksiyonu bir değişkende tutabilir, bir listeye ekleyebilir, başka bir metoda geçirebilirsin.

Neden Var?

Lambda'dan önce, "bir davranışı parametre olarak geçirmek" istediğinde tek araç anonymous inner class'tı -- tek bir satırlık mantık için bile beş altı satırlık bir iskelet yazmak gerekiyordu:

```
// Lambda öncesi: tek satırlık bir karşılaştırma mantığı için bile
// bütün bir anonymous inner class iskeleti
Comparator<String> byLength = new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return a.length() - b.length();
    }
};
```

Aynı mantık, lambda ile tek satıra iniyor:

```
Comparator<String> byLength = (a, b) -> a.length() - b.length();
```

Bu, yalnızca "daha az yazmak" değil -- kodun **niyeti** (iki string'i uzunluğa göre karşılaştır) artık `@Override` / `public int compare(...)` gibi zorunlu ama anlam taşımayan tekrarların arkasında kaybolmuyor. "Anonymous Inner Class'a Karşı Lambda" bölümünde bu farkın yalnızca görsel olmadığını, `this` davranışının da değiştiğini göreceğiz.

Tarihçe

Lambda expression'lar Java'ya **Java 8** ile (2014) geldi -- "Project Lambda" adıyla yürütülen, Java'nın o zamana kadarki en büyük dil değişikliklerinden biri. Java'nın kendisi 1995'ten beri saf nesne yönelimli bir dildi; fonksiyonel programlama dillerinde (Lisp, sonra Scala/Haskell gibi daha yeni dillerde) uzun süredir var olan "fonksiyonları veri gibi taşıma" fikri, Java'ya ancak `Comparator` / `Runnable` gibi tek-metotlu interface'lerin zaten yaygın kullanılıyor olması sayesinde nispeten kolay entegre edilebildi -- derleyici, bir lambda'yı hangi interface'in örneği yapacağını bu interface'lerin şeklinden (tek soyut metodun imzasından) çıkarabiliyordu. Aynı Java 8 sürümüyle birlikte gelen Stream API (bu kategorinin ilerleyen konuları), lambda'nın asıl gücünü ortaya çıkaran API oldu -- ikisi birlikte tasarlandı, birbirinden ayrı düşünülemez.

Parametreler: Sıfır, Bir, Birden Çok

Lambda syntax'ının parametre kısmı, parametre sayısına göre üç farklı görünüme sahip:

```
import java.util.function.BinaryOperator;

// A functional interface with one abstract method -- everything this file's lambdas
// need to demonstrate parameter/return syntax against.
interface Greeting {
    String greet(String name);
}

// Every form lambda syntax can take, side by side: parameter count, parentheses
// rules, and expression-body vs block-body.
class LambdaSyntaxAndReturnExample {
    public static void main(String[] args) {
        // Zero parameters -- empty parentheses are required, never optional.
        Runnable sayHello = () -> System.out.println("Hello!");
        sayHello.run();

        // One parameter -- parentheses are OPTIONAL when there's exactly one and it
        // has no type annotation, so both of these compile:
        Greeting shout = name -> name.toUpperCase() + "!";
        Greeting shoutParens = (name) -> name.toUpperCase() + "!";
        System.out.println(shout.greet("ayse"));

        // Two or more parameters -- parentheses become MANDATORY.
        BinaryOperator<Integer> max = (a, b) -> a > b ? a : b;
        System.out.println(max.apply(3, 7));

        // Expression body: a single expression, its value IS the return value --
        // no "return" keyword, no braces, no semicolon inside.
        Greeting expressionBody = name -> "Hi, " + name;
        System.out.println(expressionBody.greet("Fatma"));

        // Block body: braces required as soon as you need more than one statement --
        // and once you're in a block body, "return" becomes EXPLICIT and mandatory
        // for every path that produces a value.
        Greeting blockBody = name -> {
            String trimmed = name.trim();
            if (trimmed.isEmpty()) {
                return "Hi, stranger";
            }
            return "Hi, " + trimmed;
        };
        System.out.println(blockBody.greet("  Mehmet  "));
        System.out.println(blockBody.greet("   "));
    }
}
```

Beklenen Çıktı

```
Hello!
AYSE!
7
Hi, Fatma
Hi, Mehmet
Hi, stranger
```

Sıfır parametre için parantezler **zorunlu** -- `() -> ...` yazmadan boş bir lambda yazamazsın. Tam olarak tek parametre için parantezler **opsiyonel** (`name -> ...` ve `(name) -> ...` ikisi de derlenir), ama iki ya da daha fazla parametre için parantezler **zorunlu** hâle gelir -- `a, b -> ...` derlenmez, `(a, b) -> ...` yazman gerekir. Parametrelerin tipini yazmak hemen hiçbir zaman gerekmez (derleyici, "Lambda'nın Functional Interface ile Bağlantısı: Target Typing" bölümünde göreceğimiz gibi, hedef interface'in metodundan çıkarır) -- ama istersen `(String name) -> ...` gibi açıkça da yazabilirsin.

Gövde: Expression Body vs Block Body

Aynı örnekte gördüğün gibi, bir lambda'nın gövdesi iki biçimden birini alabilir. **Expression body**, tek bir ifadedir -- `name -> "Hi, " + name` -- süslü parantez yok, `return` yok, noktalı virgül yok; ifadenin değeri **doğrudan** dönüş değeri olur. **Block body**, `{ }` içine alınmış bir ya da daha fazla ifadedir -- birden fazla satıra ihtiyaç duyduğun an bu biçime geçmen gerekir, ve block body'ye girer girmez `return` **açık ve zorunlu** hâle gelir (değer döndüren her kod yolunda). Bu ayrım küçük görünse de sık yapılan bir hata kaynağı -- "Yaygın Hatalar" bölümünde bunu tekrar göreceğiz.

Lambda'nın Functional Interface ile Bağlantısı: Target Typing

Bir lambda'nın **kendi başına** bir tipi yoktur -- derleyici ona, bulunduğu **bağlamdan** (target type) bakarak bir tip verir:

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.function.BiFunction;

// The SAME lambda literal has no type of its own -- the compiler decides what it
// becomes purely from context (its "target type": a variable's declared type, a
// method parameter's type, or a return type). Two functional interfaces with the
// exact same shape (two Strings in, one int out) can both host this literal.
class TargetTypingExample {
    public static void main(String[] args) {
        // Target type: Comparator<String> -- int compare(String, String)
        Comparator<String> byLength = (a, b) -> a.length() - b.length();
        System.out.println(byLength.compare("hi", "hello"));

        // Target type: BiFunction<String, String, Integer> -- structurally identical
        // shape, a completely different interface, the exact same lambda syntax.
        BiFunction<String, String, Integer> lengthDiff = (a, b) -> a.length() - b.length();
        System.out.println(lengthDiff.apply("hi", "hello"));

        // Passing a lambda directly as a method argument -- the target type comes
        // from the parameter's declared type, List.sort(Comparator<? super E>).
        List<String> names = new ArrayList<>(List.of("Ahmet", "Al", "Ayse"));
        names.sort((a, b) -> a.length() - b.length());
        System.out.println(names);
    }
}
```

> **Beklenen Çıktı**

```
-3
-3
[Al, Ayse, Ahmet]
```

`(a, b) -> a.length() - b.length()` ifadesinin kendisi hiçbir şey söylemiyor -- bir değişkene `Comparator<String>` olarak atandığında `Comparator`, `BiFunction<String, String, Integer>` olarak atandığında `BiFunction` oluyor; ikisinin de tek soyut metodu aynı şekle (iki `String` içeri, bir sonuç dışarı) sahip olduğu için **aynı** lambda ifadesi ikisine de uyuyor. `List.sort(...)`'a doğrudan bir lambda geçirdiğimizde de aynı mekanizma işliyor -- hedef tip, `sort` metodunun parametre tipinden (`Comparator<? super E>`) geliyor. Bu, "Built-in Functional Interfaces" dersinde göreceğimiz `Predicate` / `Function` / `Consumer` gibi hazır tiplerin neden bu kadar sık işe yaradığının da temeli -- aynı şekle sahip birçok farklı senaryoyu tek bir hazır interface karşılayabiliyor.

Değişken Yakalama: Effectively Final

Bir lambda, kendi gövdesinin dışındaki (enclosing scope'taki) yerel değişkenleri okuyabilir -- ama yalnızca o değişken **effectively final** ise, yani ilk atamasından sonra bir daha **hiç** yeniden atanmamışsa (açıkça `final` yazılmasa bile):

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

// A lambda can read any local variable from its enclosing scope -- but ONLY if that
// variable is "effectively final": never reassigned after its first assignment, even
// if it's never explicitly marked `final`.
class EffectivelyFinalExample {
    public static void main(String[] args) {
        String prefix = "Order #"; // never reassigned -> effectively final, capturable

        Supplier<String> label = () -> prefix + "42";
        System.out.println(label.get());

        // The line below would NOT compile if uncommented -- reassigning "prefix"
        // anywhere after the lambda captures it makes it no longer effectively final:
        // prefix = "Ticket #";

        // A mutable object (not a local variable) sidesteps the restriction entirely
        // -- this is the usual escape hatch when a lambda genuinely needs to
        // accumulate state across multiple calls.
        List<String> collected = new ArrayList<>(); // the REFERENCE never changes
        Runnable addOne = () -> collected.add("item"); // mutating the LIST is fine
        addOne.run();
        addOne.run();
        System.out.println(collected);
    }
}
```

Beklenen Çıktı

```
Order #42
[item, item]
```

`prefix` değişkeni bir daha atanmadığı için lambda onu sorunsuzca yakalayabiliyor; eğer `label` tanımlandıktan sonra herhangi bir yerde `prefix = "..."` satırı olsaydı, derleme hatası alırdın -- ve bu hata, o satırı lambda'dan **sonra** yazsan bile oluşur, çünkü kural "yakalandıktan sonra" değil "en baştan sona kadar hiç yeniden atanmamış olmak". Bunun pratikteki en yaygın çözümü, örnekte de gördüğün gibi, referansı değil referansın **içeriğini** değiştirmek -- `collected` değişkeninin kendisi hiç yeniden atanmıyor (hâlâ effectively final), ama işaret ettiği liste `add(...)` ile değişebiliyor.

> Effectively final kuralının sebebi thread safety değil (bir lambda mutlaka başka bir thread'de çalışmak zorunda değil) -- asıl sebep, lambda'nın değişkeni **kopyalayarak** yakalaması. Değişken yeniden atanabilseydi, lambda'nın kendi kopyası ile dış scope'taki gerçek değer birbirinden sessizce sapardı; derleyici bunu daha en baştan, derleme zamanında engelliyor.

Anonymous Inner Class'a Karşı Lambda

"Neden Var?" bölümünde gördüğümüz syntax farkının ötesinde, ikisi arasında gerçek bir davranış farkı daha var -- `this` referansının ne anlama geldiği:

```
interface ClickHandler {
    void onClick(String source);
}

// Same functional interface, two ways to provide an implementation -- side by side,
// to make both the syntax reduction and the "this" difference concrete.
class AnonymousClassVsLambdaExample {

    private final String owner = "AnonymousClassVsLambdaExample";

    void demoThis() {
        // Anonymous inner class: creates a REAL, separate class -- "this" inside its
        // body refers to the anonymous class instance itself.
        ClickHandler anonymous = new ClickHandler() {
            @Override
            public void onClick(String source) {
                System.out.println("anonymous this: " + this.getClass().getName());
            }
        };

        // Lambda: does NOT create a new class in that same sense -- "this" inside a
        // lambda body is inherited from the ENCLOSING scope, exactly as if the
        // lambda's body were pasted directly into the surrounding method.
        ClickHandler lambda = source -> System.out.println("lambda this: " + this.owner);

        anonymous.onClick("button");
        lambda.onClick("button");
    }

    public static void main(String[] args) {
        new AnonymousClassVsLambdaExample().demoThis();
    }
}
```

Anonymous inner class, gerçek ve **ayrı** bir sınıf üretir (derlenmiş halinde `AnonymousClassVsLambdaExample$1` gibi bir isimle görürsün) -- içindeki `this`, o anonymous sınıfın kendi örneğine işaret eder. Lambda ise yeni bir sınıf üretmiyormuş **gibi** davranır -- içindeki `this`, sanki lambda'nın gövdesi olduğu gibi çevreleyen metodun içine yapıştırılmış gibi, **çevreleyen nesneye** işaret eder. Bu fark küçük görünse de, bir lambda'nın içinde `this.owner` yazdığında dış sınıfın alanına ulaşabilmenin (anonymous class'ta bunun için `DışSınıf.this.owner` yazman gerekirdi) tam sebebi bu.

Best Practices

- **Lambda gövdesini kısa tut** -- birkaç satırı aşan bir block body, genelde adlı bir metoda (ya da ayrı bir sınıfa) çıkarılmayı işaret eder; lambda'nın gücü kısalığından gelir.
- **Parametre tiplerini yazma, derleyicinin çıkarmasına izin ver** -- "Parametreler: Sıfır, Bir, Birden Çok" bölümünde gördüğümüz gibi, açık tip yazmak neredeyse hiçbir zaman gerekli değil ve gereksiz gürültü ekler.
- **Effectively final kısıtını bir engel değil bir tasarım sinyali olarak gör** -- "Değişken Yakalama: Effectively Final" bölümünde gördüğümüz gibi, bir lambda'nın dış bir

değişkeni yeniden atamaya ihtiyaç duyması genelde o mantığın aslında bir lambda olarak değil, adlı bir metot olarak yazılması gerektiğinin işaretidir.

- **Tek satırlık mantık için lambda'yı, birden fazla ilişkili metot gerekiyorsa anonymous class'ı (ya da adlı bir sınıfı) tercih et** -- "Anonymous Inner Class'a Karşı Lambda" bölümünde gördüğümüz gibi, functional interface'ler tanım gereği tek metotlu.
- **Kendi functional interface'ini yazmadan önce `java.util.function` paketini kontrol et** ("Interface" dersinin "Functional Interface ve Lambda" bölümünde de değindiğimiz bir prensip) -- bu kategorinin bir sonraki konusu tam olarak bu pakete ayrılacak.

Yaygın Hatalar

1. Block body'de `return` yazmayı unutmak. Expression body'nin alışkanlığıyla `{ "Hi, " + name }` gibi bir şey yazmak derlenmez -- "Gövde: Expression Body vs Block Body" bölümünde gördüğümüz gibi, süslü parantez açtığın an `return` zorunlu hâle gelir.

2. Tek parametreli bir lambda'da bazen parantez yazıp bazen yazmamak, tutarsız bir kod tabanı bırakmak. İkisi de derlense de ("Parametreler: Sıfır, Bir, Birden Çok" bölümü), aynı dosyada/takımda tek bir kural seçip ona sadık kalmak okunabilirliği artırır.

3. Effectively final olmayan bir değişkeni yakalamaya çalışıp derleme hatasını anlamamak. "Değişken Yakalama: Effectively Final" bölümünde gördüğümüz gibi, hata mesajı genelde "yakalandığı" satırı işaret eder, ama gerçek sebep o değişkenin **başka bir yerde** yeniden atanmış olmasıdır -- çözüm için değişkenin tüm kullanım yerlerine bakmak gerekir.

4. Bir lambda içinde `this`'in anonymous class'taki gibi davranacağını sanmak. "Anonymous Inner Class'a Karşı Lambda" bölümünde gördüğümüz gibi, bir lambda'nın içindeki `this` çevreleyen nesneye işaret eder -- anonymous class'takinin aksine, lambda'nın "kendi" bir `this`'i yoktur.

5. Her `Comparator` / `Runnable` /vb. ihtiyacında yeni bir isimli sınıf yazmaya devam etmek. "Neden Var?" bölümünde gördüğümüz gibi, lambda tam olarak bu tekrarı ortadan kaldırmak için var -- tek metotlu bir interface'in tek seferlik bir implementasyonuna ihtiyaç duyduğunda önce lambda'yı düşün.

Özet, Cheat Sheet ve Terimler Sözlüğü

Bu derste lambda expression syntax'ını uçtan uca gördük: parametre yazım kuralları, expression/block body ayrımı ve `return`'ün ne zaman zorunlu olduğu, derleyicinin lambda'ya bağlamdan bir tip vermesi (target typing), effectively final kısıtı ve sebebi, ve lambda'nın anonymous inner class'tan `this` davranışı açısından farkı. Öne çıkan noktalar:

- Lambda, tam olarak tek soyut metotlu bir interface'in (functional interface) anlık bir örneğidir -- kendi başına bir tipi yoktur

- Sıfır parametre için parantez zorunlu, tek parametre için opsiyonel, ikiden fazlası için yine zorunlu
- Expression body'de `return` yok ve örtük; block body'de `return` açık ve zorunlu
- Bir lambda yalnızca effectively final (bir daha hiç yeniden atanmamış) yerel değişkenleri yakalayabilir
- Lambda içindeki `this`, çevreleyen nesneye işaret eder -- anonymous class'takinin aksine kendi bir `this`'i yoktur

Hızlı referans:

```
() -> ...                   // parametresiz
x -> ...                    // tek parametre, parantezsiz
(x) -> ...                  // tek parametre, parantezli
(x, y) -> ...               // birden çok parametre, parantez zorunlu
x -> x * 2                  // expression body, örtük return
x -> { return x * 2; }      // block body, açık return zorunlu
```

Terimler Sözlüğü

Lambda expression — Adı olmayan, bir functional interface'in tek soyut metodunu uygulayan kısa fonksiyon tanımı.

Functional interface — Tam olarak tek soyut metoda sahip interface ("Interface" dersi).

Expression body — Tek bir ifadeden oluşan, `return` gerektirmeyen lambda gövdesi.

Block body — `{ }` içine alınmış, birden fazla ifade içerebilen, değer döndüren her yolda açık `return` gerektiren lambda gövdesi.

Target type (hedef tip) — Derleyicinin bir lambda'ya, bulunduğu bağlamdan (değişken tipi, parametre tipi, dönüş tipi) çıkardığı functional interface tipi.

Effectively final — Bir yerel değişkenin, ilk atamasından sonra hiç yeniden atanmamış olması durumu; lambda'lar yalnızca bu tür değişkenleri yakalayabilir.