switch

Control Flow kategorisinin ikinci topic'i: klasik `case`/`break` sözdizimi ve fall-through tuzağı, modern ok (`->`) sözdizimi, switch İFADESİ ve `yield`, birden fazla etiket ve enum kapsayıcılık (exhaustiveness) kontrolü, ve `String`/enum üzerinde `switch` kullanımı.

Başlangıç 20 dk

Oluşturulma tarihi: 2 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/switch

---

switch

Control Flow kategorisinin ikinci konusu switch -- tek bir değeri birden fazla olası sabitle karşılaştırıp buna göre dallanan bir kontrol yapısı. Uzun bir "else if Zinciri" (bkz. "if / else" dersi) her zaman bir `switch`'e çevrilebilir, ama `switch` bu özel durumda -- TEK bir değişkenin BİRDEN FAZLA sabit değerle karşılaştırılması -- hem daha kısa hem derleyicinin daha fazla kontrol yapabildiği (örn. bir enum'daki her sabitin ele alındığını doğrulama) bir sözdizimi sunar. Java 21 hedefiyle bu ders hem klasik `switch` sözdizimini hem 2017'den beri gelen ve önce inceleme (preview), sonra Java 14'te (2020) kalıcı hale gelen modern ok (`->`) sözdizimini ve `switch` İFADESİNİ birlikte ele alıyor.

switch Nedir?

`switch`, parantez içindeki bir değeri (`int`, `String`, bir enum sabiti, vb.) sırayla `case` etiketleriyle karşılaştırır -- eşleşen `case` çalışır, hiçbiri eşleşmezse (varsa) `default` çalışır. Java iki farklı sözdizimi sunar: KLASİK sözdizimi (`case deger:` ve isteğe bağlı `break;`) ve MODERN ok sözdizimi (`case deger -> ifade;`, Java 14'ten beri). Aynı zamanda `switch` bir İFADE (expression) olarak da kullanılabilir -- doğrudan bir değere değerlendirilip bir değişkene atanabilir (bkz. "switch İfadesi ve yield").

Neden Var?

Aynı değişkeni art arda birçok sabitle karşılaştıran bir `else if` zinciri (bkz. "else if Zinciri") teknik olarak her zaman çalışır ama iki sorunu vardır: (1) her koşulda değişken adı TEKRAR TEKRAR yazılır (`if (day == 1) ... else if (day == 2) ...`), okunabilirliği düşürür; (2) derleyici bunun "tek bir değeri sabit kümesiyle karşılaştırma" deseni olduğunu ANLAMAZ, bu yüzden -- örneğin bir enum'un TÜM sabitlerinin ele alınıp alınmadığını -- kontrol edemez.

`switch`, bu deseni doğrudan dile tanıtarak hem daha kısa yazım hem derleyici destekli tamlık (exhaustiveness) kontrolü sağlar.

Tarihçe

`switch`, C'den (1972) Java'ya (1995) neredeyse birebir geçti -- klasik `case`/`break` sözdizimi ve fall-through davranışı (bkz. "Fall-Through: break Unutmanın Bedeli") C'nin doğrudan mirasıdır. Bu davranış onlarca yıl gerçek bir hata kaynağı olarak kaldı; Java, JEP 325 ile 2017'de bir ÖNİZLEME (preview) özelliği olarak modern ok sözdizimini (`->`, fall-through YOK) tanıttı, bu Java 14'te (2020) kalıcı bir dil özelliği hâline geldi -- aynı JEP, `switch`'in bir İFADE olarak kullanılabilmesini ve `yield` anahtar kelimesini de getirdi.

Klasik switch Sözdizimi

Klasik sözdizimde her `case` bir DEĞER etiketi ve ardından çalıştırılacak ifadelerdir; `break`, `switch`'ten çıkışı sağlar. `default`, hiçbir `case` eşleşmediğinde çalışır -- zorunlu değildir ama iyi bir pratiktir (bkz. "Yaygın Hatalar").

```
public class SwitchBasicsExample {
    public static void main(String[] args) {
        int day = 3;
        String name;

        switch (day) {
            case 1:
                name = "Monday";
                break;
            case 2:
                name = "Tuesday";
                break;
            case 3:
                name = "Wednesday";
                break;
            default:
                name = "Unknown";
                break;
        }

        System.out.println("Day " + day + " -> " + name);
    }
}
```

Fall-Through: break Unutmanın Bedeli

Klasik sözdizimde bir `case`'in sonuna `break` KONULMAZSA, çalışma bir SONRAKİ `case`'in içine de "düşer" (fall-through) -- eşleşip eşleşmediğine BAKILMAKSIZIN, bir `break` ya da

`switch`'in sonu bulunana kadar. Bazı nadir durumlarda (birden fazla `case`'in AYNI davranışı paylaşması) bu kasıtlı olarak kullanılır, ama çoğu zaman unutulmuş bir `break` gerçek bir hatadır.

```
public class FallThroughExample {
    public static void main(String[] args) {
        System.out.println("Without break (BUGGY):");
        printDayBuggy(2);

        System.out.println();
        System.out.println("With break (FIXED):");
        printDayFixed(2);
    }

    // Missing break statements -- once a case matches, execution keeps
    // running into every case BELOW it until a break (or the end) is hit.
    private static void printDayBuggy(int day) {
        switch (day) {
            case 1:
                System.out.println("Monday");
            case 2:
                System.out.println("Tuesday");
            case 3:
                System.out.println("Wednesday");
            default:
                System.out.println("Unknown");
        }
    }

    private static void printDayFixed(int day) {
        switch (day) {
            case 1:
                System.out.println("Monday");
                break;
            case 2:
                System.out.println("Tuesday");
                break;
            case 3:
                System.out.println("Wednesday");
                break;
            default:
                System.out.println("Unknown");
                break;
        }
    }
}
```

> Fall-through'u KASITLI kullanıyorsan (örn. `case 6: case 7: // hafta sonu`) bunu açıklayan bir yorum eklemek güçlü bir pratiktir -- okuyan kişi bunun unutulmuş bir `break` mi yoksa bilinçli bir tasarım mı olduğunu bilemez. Modern ok sözdizimi (bkz. "Modern Ok (Arrow) Sözdizimi") bu belirsizliği tamamen ortadan kaldırır.

Modern Ok (Arrow) Sözdizimi

Java 14'ten beri kalıcı olan `->` sözdizimi, HER `case`'i kendi bağımsız dalı olarak çalıştırır -- fall-through YOKTUR, `break` GEREKMEZ. Birden fazla değer virgülle ayrılarak tek bir dala bağlanabilir (`case 1, 2, 3 -> ...`).

```
public class ArrowSwitchExample {
    public static void main(String[] args) {
        printDayType(3);
        printDayType(6);
        printDayType(7);
    }

    // The arrow (->) form never falls through -- each case runs ONLY its own
    // branch, no break needed. Multiple labels can share one branch.
    private static void printDayType(int day) {
        switch (day) {
            case 1, 2, 3, 4, 5 -> System.out.println(day + " is a weekday.");
            case 6, 7 -> System.out.println(day + " is a weekend day.");
            default -> System.out.println(day + " is not a valid day.");
        }
    }
}
```

switch İfadesi ve yield

`switch`, bir İFADE olarak da yazılabilir -- doğrudan bir değere değerlendirilir ve bir değişkene atanabilir (`String sonuc = switch (x) { ... };`), tıpkı üçlü operatör gibi (bkz. "if / else" dersindeki "Üçlü (Ternary) Operatör" bölümü) ama daha fazla dal destekler. Ok sözdiziminde tek bir ifade doğrudan değeri üretir; birden fazla adım gereken bir BLOK gövdesi kullanılıyorsa, üretilen değer `yield` anahtar kelimesiyle belirtilmelidir.

```
public class SwitchExpressionExample {
    public static void main(String[] args) {
        System.out.println("1 -> " + classify(1));
        System.out.println("3 -> " + classify(3));
        System.out.println("6 -> " + classify(6));
        System.out.println("9 -> " + classify(9));
    }

    // A switch EXPRESSION evaluates directly to a value -- no need for a
    // separate variable that gets assigned inside each branch.
    private static String classify(int day) {
        return switch (day) {
            case 1, 2, 3, 4, 5 -> "weekday";
            case 6, 7 -> "weekend";
            default -> {
                // A block body needs `yield` to produce the expression's value.
                System.out.println("  (unexpected value logged: " + day + ")");
                yield "invalid";
            }
        };
    }
}
```

Birden Fazla Etiket ve Kapsayıcılık (Exhaustiveness)

Bir `case`'e virgülle birden fazla değer bağlamak (`case SATURDAY, SUNDAY -> ...`) kod tekrarını azaltır. Bir enum üzerindeki `switch` İFADESİ, enum'un TÜM sabitleri en az bir `case` 'de ele alınmışsa `default` YAZMADAN da derlenir -- derleyici bunu KAPSAYICILIK (exhaustiveness) kontrolüyle doğrular. Bu, ileride enum'a yeni bir sabit eklenip bu `switch`'in güncellenmesi UNUTULURSA, sessiz bir çalışma zamanı hatası yerine bir DERLEME HATASI almanı sağlar.

```
public class MultipleLabelsAndDefaultExample {

    enum Season { WINTER, SPRING, SUMMER, FALL }

    public static void main(String[] args) {
        System.out.println(describe(Season.WINTER));
        System.out.println(describe(Season.SPRING));
        System.out.println(describe(Season.SUMMER));
        System.out.println(describe(Season.FALL));
    }

    // A switch EXPRESSION over an enum can skip `default` entirely as long as
    // EVERY constant is covered -- the compiler checks this exhaustiveness for
    // you. Adding a new enum constant later without updating this switch
    // becomes a COMPILE ERROR, not a silent bug.
    private static String describe(Season season) {
        return switch (season) {
            case WINTER, FALL -> "cold";
            case SPRING, SUMMER -> "warm";
        };
    }
}
```

> `switch` bir İFADE değil bir "statement" (klasik ya da ok sözdizimiyle yazılmış ama bir değer ÜRETMEYEN) olarak kullanıldığında bu kapsayıcılık kontrolü ÇALIŞMAZ -- derleyici zorlamaz. Kapsayıcılık garantisinden faydalanmak istiyorsan `switch`'i bir DEĞER üreten ifade olarak yazmak (yalnızca yan etki için değil) bilinçli bir tercih olmalı.

switch ile String ve Enum Üzerinde Çalışmak

`switch`, `int` / `char` gibi ilkellerin yanı sıra `String` ve enum sabitleriyle de çalışır. Bir `String` üzerindeki `switch`, İÇERİĞİ karşılaştırır (`.equals()` gibi, `==` gibi DEĞİL) -- bkz. "if / else" dersindeki "Karşılaştırma Operatörleri" bölümü. Bir enum üzerindeki `switch`'te `case` etiketleri enum adı ÖN EKİ OLMADAN yazılır (`case ADMIN ->`, `case Role.ADMIN ->` DEĞİL) -- bu, enum'un kendi "switch ile Kullanım" bölümünde de gösterilen bir davranıştır.

```
public class SwitchOnStringAndEnumExample {

    enum Role { ADMIN, EDITOR, VIEWER }

    public static void main(String[] args) {
        System.out.println(permissionsFor("admin"));
        System.out.println(permissionsFor("editor"));
        System.out.println(permissionsFor("guest"));

        System.out.println(roleLabel(Role.ADMIN));
        System.out.println(roleLabel(Role.VIEWER));
    }

    // Switching on a String compares by CONTENT (like .equals(), not ==) --
    // no need to worry about the == vs equals() trap from the "Comparison
    // Operators" section of the if / else lesson.
    private static String permissionsFor(String role) {
        return switch (role) {
            case "admin" -> "read, write, delete";
            case "editor" -> "read, write";
            default -> "read only";
        };
    }

    private static String roleLabel(Role role) {
        return switch (role) {
            case ADMIN -> "Administrator";
            case EDITOR -> "Editor";
            case VIEWER -> "Viewer";
        };
    }
}
```

Best Practices

- **Klasik sözdizimde her `case`'in sonuna `break` koymayı asla unutma** -- ya da tamamen fall-through'suz modern ok sözdizimini tercih et.
- **Fall-through'u kasıtlı kullanıyorsan bunu bir yorumla açıkça belirt** -- aksi hâlde unutulmuş bir `break` ile karıştırılır.
- **Bir enum üzerinde tüm sabitleri kapsayan bir `switch` yazarken `default`'u atlayıp kapsayıcılık kontrolünden faydalan** -- yeni bir sabit eklendiğinde derleme hatası almak, sessiz bir hatadan çok daha iyidir.
- **Bir değer üretmesi gereken `switch`'leri bir İFADE olarak yaz, klasik statement + geçici değişken deseni yerine.**
- **Uzun bir `else if` zinciri TEK bir değişkeni birden fazla sabitle karşılaştırıyorsa, onu bir `switch`'e çevirmeyi düşün.**

Yaygın Hatalar

- **Klasik sözdizimde bir `case`'in sonuna `break` koymayı unutup istenmeyen fall-through yaşamak.** Çalışma, eşleşmeyen sonraki `case`'lere de "düşer".
- **`String` üzerindeki bir `switch`'i `==` ile karşılaştırma yapıyormuş gibi düşünmek.** Aslında içerik karşılaştırması yapılır, `String` referans tuzağı burada geçerli DEĞİLDİR.
- **Enum `case` etiketlerinin önüne yanlışlıkla enum adını eklemek** (`case Role.ADMIN ->` yerine `case ADMIN ->` olmalı) -- bu bir derleme hatasıdır.
- **Bir `switch` İFADESİNİN blok gövdesinde `yield` yazmayı unutmak** -- derleyici, bloğun hangi değeri ürettiğini bilemez.

Özet, Cheat Sheet ve Terimler Sözlüğü

`switch`, tek bir değeri birden fazla sabitle karşılaştıran bir kontrol yapısıdır -- uzun bir `else if` zincirinin daha kısa, derleyici destekli bir alternatifi. Klasik sözdizim `case`/`break` kullanır ve `break` unutulursa FALL-THROUGH yaşanır; modern ok (`->`) sözdizimi fall-through'suzdur. `switch` bir İFADE olarak yazılabilir, bir değere değerlendirilir; blok gövdelerde `yield` gerekir. Bir enum'un tüm sabitlerini kapsayan bir `switch` İFADESİ `default` olmadan derlenebilir -- derleyici kapsayıcılığı kontrol eder.

Hızlı referans:

```
// Klasik
switch (gun) {
    case 1: sonuc = "Pazartesi"; break;
    default: sonuc = "Bilinmiyor"; break;
}

// Modern ok sözdizimi
switch (gun) {
    case 1, 2, 3, 4, 5 -> System.out.println("Hafta ici");
    case 6, 7 -> System.out.println("Hafta sonu");
}

// switch ifadesi + yield
String tur = switch (gun) {
    case 1, 2, 3, 4, 5 -> "hafta ici";
    case 6, 7 -> "hafta sonu";
    default -> {
        yield "gecersiz";
    }
};
```

Terimler Sözlüğü

Fall-Through — Klasik `switch` sözdiziminde bir `break` bulunana kadar çalışmanın sonraki `case`'lere de devam etmesi.

Switch Expression (switch İfadesi) — Doğrudan bir değere değerlendirilen, bir değişkene atanabilen `switch` biçimi.

yield — Bir `switch` ifadesinin BLOK gövdesi içinde üretilecek değeri belirten anahtar kelime.

Exhaustiveness (Kapsayıcılık) — Bir `switch` ifadesinin, üzerinde çalıştığı tüm olası değerleri (örn. bir enum'un tüm sabitlerini) `default` olmadan kapsadığının derleyici tarafından doğrulanması.

Daha ileri düzey bir konu olarak, `switch`'in nesne TİPLERİYLE ve record'ların alanlarıyla eşleştirilmesi (pattern matching) "Record" dersindeki "Ek: Record Patterns (Java 21)" bölümünde ayrıca ele alınıyor.