Enum

Java'da sabit değer kümelerini tip güvenli şekilde tanımlama.

Başlangıç 35 dk

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

Enum

Java'da **enum** (enumeration), sabit bir değer kümesini tip güvenli şekilde tanımlamanı sağlayan özel bir referans türüdür. Java 5 ile dile eklendi ve o günden beri "bu değişken şu birkaç değerden birini alabilir" durumlarının standart çözümü hâline geldi.

Enum Nedir?

Bir değişkenin alabileceği değerler önceden bilinen, sınırlı bir kümeyse (örneğin haftanın günleri, bir siparişin durumları, bir kartın rengi), bunu string veya int sabitlerle değil enum ile modellemelisin. Java'da her enum, arka planda gizlice `java.lang.Enum` sınıfını extend eden bir sınıftır — yani enum sabitleri aslında birer **nesnedir**, sadece sayısı sabittir:

```
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Bu tek satırda Java, arka planda yedi tane `Day` nesnesi (`MONDAY`, `TUESDAY`, ...) oluşturur ve bunları `static final` alanlar olarak saklar. Bu yüzden enum sabitlerini `==` ile karşılaştırmak tamamen güvenlidir — `equals()`'a ihtiyaç yoktur.

Enum vs String

Enum kullanmak, string sabitlere göre derleme zamanında tip güvenliği sağlar. Şu iki yaklaşımı karşılaştır:

```
// String ile — derleyici seni korumaz
void setStatus(String status) { ... }
setStatus("APPROVEDD"); // yazım hatası, ama kod derlenir ve sorun çalışma zamanında çıkar

// Enum ile — derleyici seni korur
void setStatus(OrderStatus status) { ... }
setStatus(OrderStatus.APPROVEDD); // derleme hatası — böyle bir sabit yok
```

String ile yazım hatası ancak çalışma zamanında (belki de üretimde) fark edilir; enum ile aynı hata derleme anında yakalanır. Ayrıca IDE'ler enum kullanırken otomatik tamamlama ve "olası tüm değerler" listesi sunabilir — string ile bu mümkün değildir.

Temel Enum Kullanımı

En basit haliyle bir enum şöyle tanımlanır:

```
enum Gender {
    MALE,
    FEMALE
}
```

> Enum sabitleri, gelenek olarak büyük harfle yazılır (`MALE` , `FEMALE`) — tıpkı `static final` sabitler gibi, çünkü aslında onlar da öyledir.

Constructor

Enum sabitleri birer nesne olduğu için constructor'ları da olabilir — her sabit oluşturulurken kendi parametrelerini geçebilir:

```
enum Planet {
    MERCURY(3.303e+23, 2.4397e6),
    VENUS(4.869e+24, 6.0518e6),
    EARTH(5.976e+24, 6.37814e6);

    private final double massKg;
    private final double radiusM;

    // The constructor must always be private (or package-private) — enum
    // constants are only ever "constructed" on the definition line above,
    // by the enum itself.
    Planet(double massKg, double radiusM) {
        this.massKg = massKg;
        this.radiusM = radiusM;
    }

    public double getMassKg() {
        return massKg;
    }

    public double getRadiusM() {
        return radiusM;
    }

    public double surfaceGravity() {
        final double gravitationalConstant = 6.67300E-11;
        return gravitationalConstant * massKg / (radiusM * radiusM);
    }
}
```

Burada her gezegen sabiti (`MERCURY`, `VENUS`, `EARTH`), enum'un tanımlandığı satırda kendi kütle ve yarıçap değerleriyle "inşa ediliyor". Bu constructor çağrısı, sınıf ilk yüklendiğinde (class loading anında) bir kereye mahsus çalışır — her sabit için tek bir nesne örneği vardır, tıpkı bir Singleton gibi.

> Enum constructor'ları asla `public` ya da `protected` olamaz — Java bunu derleme zamanında zorunlu kılar (yalnızca `private` ya da paket-private/varsayılan erişim geçerlidir). Çünkü enum sabitleri yalnızca enum'un kendisi tarafından, tanım satırında oluşturulabilir; dışarıdan `new Planet(...)` çağırmak mümkün değildir.

Alanlar (Fields)

Constructor'a geçirilen değerler genellikle `private final` alanlarda saklanır — bu, her enum sabitinin kendine özgü, değişmez (immutable) verisi olduğu anlamına gelir. Yukarıdaki `Planet` örneğinde `massKg` ve `radiusM` tam olarak bu şekilde tanımlandı; dışarıya ise yalnızca getter metotlarıyla açıldı:

```
class PlanetUsageExample {
    public static void main(String[] args) {
        Planet earth = Planet.EARTH;

        System.out.println("Mass: " + earth.getMassKg() + " kg");
        System.out.printf("Surface gravity: %.2f m/s^2%n", earth.surfaceGravity());
    }
}
```

> Enum alanlarını `final` yapmak bir zorunluluk değil ama güçlü bir alışkanlıktır: bir sabitin (örneğin `EARTH`) çalışma zamanında farklı bir kütleye "değişmesi" mantıksal olarak anlamsızdır — `final`, bunu derleyiciye garanti ettirir.

Metotlar

Enum'lar sıradan sınıflar gibi instance metotları barındırabilir. `Planet` örneğindeki `surfaceGravity()` buna zaten bir örnekti. Constructor'a hiç ihtiyaç duymadan, sabit değerlere göre karar veren bir metot da ekleyebiliriz:

```
enum DayWithMethod {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

    boolean isWeekend() {
        return this == SATURDAY || this == SUNDAY;
    }
}
```

```
for (DayWithMethod day : DayWithMethod.values()) {
    System.out.println(day + " hafta sonu mu? " + day.isWeekend());
}
```

values()

`values()`, derleyicinin her enum için otomatik ürettiği statik bir metottur; tüm sabitleri, tanımlandıkları sırayla bir dizi (array) olarak döner — yukarıdaki döngüde zaten kullandık:

```
DayWithMethod[] days = DayWithMethod.values();
System.out.println(days.length); // 7
```

> `values()` her çağrıldığında yeni bir dizi kopyalar — bir döngü içinde tekrar tekrar çağırmak (örneğin `for (int i = 0; i < DayWithMethod.values().length; i++)`) gereksiz performans kaybına yol açar. Diziyi bir kere alıp bir değişkende tutmak daha doğrudur.

valueOf()

`valueOf(String)`, verilen ismi tam olarak eşleşen sabiti döner — eşleşme bulunamazsa `IllegalArgumentException` fırlatır:

```
DayWithMethod day = DayWithMethod.valueOf("MONDAY"); // MONDAY
DayWithMethod hata = DayWithMethod.valueOf("monday"); // IllegalArgumentException! Büyük
/küçük harfe duyarlı
```

Kullanıcıdan gelen serbest metni doğrudan `valueOf()`'a vermeden önce mutlaka doğrula ya da çağrıyı `try/catch` ile sarmala.

name()

`name()`, sabitin kaynak kodda yazıldığı ismi **birebir** döner — `toString()` override edilmiş olsa bile `name()` her zaman orijinal ismi verir:

```
System.out.println(DayWithMethod.MONDAY.name()); // "MONDAY"
```

> Kullanıcıya gösterilecek metin için `name()` yerine, ayrı bir `displayName` alanı (constructor ile geçirilen) ya da `toString()` override'ı kullan — `name()` sabittir ve yerelleştirilemez (i18n'e uygun değildir).

ordinal()

`ordinal()`, sabitin tanımlandığı sıradaki pozisyonunu (0'dan başlayarak) döner:

```
System.out.println(DayWithMethod.MONDAY.ordinal()); // 0
System.out.println(DayWithMethod.SUNDAY.ordinal());  // 6
```

Bu bölümün en başında verdiğimiz uyarıyı burada tekrar hatırlatmakta fayda var: `ordinal()` değerini asla kalıcı veri (veritabanı, dosya, API sözleşmesi) olarak saklama — enum tanımına yeni bir sabit eklemek ya da sırayı değiştirmek, mevcut verinin anlamını sessizce bozar.

switch ile Kullanım

Enum'lar, `switch` ifadeleriyle doğal olarak çok iyi çalışır. Java 21 ile gelen modern switch söz dizimiyle:

```
String mesaj = switch (DayWithMethod.SATURDAY) {
    case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> "Çalışma günü";
    case SATURDAY, SUNDAY -> "Hafta sonu";
};
```

> `case` etiketlerinde enum sabitinin adını tek başına yazıyoruz (`MONDAY`), `DayWithMethod.MONDAY` değil — Java bunu switch edilen tipten çıkarım yaparak otomatik tamamlar. Ayrıca tüm sabitleri kapsayan bir switch ifadesinde (yukarıdaki gibi) `default` dalına gerek yoktur; derleyici olası tüm sabitlerin karşılandığını kontrol eder ve eksik varsa hata verir — yeni bir sabit eklendiğinde unutulmuş switch bloklarını derleme zamanında yakalamanın harika bir yolu.

Arayüz (Interface) İmplementasyonu

Bir enum, sınıf gibi extend edemez (zaten örtük olarak `Enum`'ı extend eder) ama istediğin kadar arayüz implement edebilir. Bu, enum sabitlerine ortak bir "sözleşme" kazandırmanın güzel bir yoludur:

```
interface Describable {
    String describe();
}

enum TrafficLight implements Describable {
    RED, YELLOW, GREEN;

    @Override
    public String describe() {
        return "Traffic light: " + name();
    }
}
```

Burada her `TrafficLight` sabiti aynı `describe()` implementasyonunu paylaşıyor. Bir sonraki bölümde, her sabitin **kendi** implementasyonunu nasıl yazabileceğini göreceğiz.

Soyut Metot (Abstract Method) ve Sabite Özel Gövde

Enum'ın en güçlü özelliklerinden biri, her sabitin bir metodun **kendi** implementasyonunu yazabilmesidir — buna "constant-specific method body" denir:

```
enum Operation {
    PLUS("+") {
        @Override
        public double apply(double a, double b) {
            return a + b;
        }
    },
    MINUS("-") {
        @Override
        public double apply(double a, double b) {
            return a - b;
        }
    },
    TIMES("*") {
        @Override
        public double apply(double a, double b) {
            return a * b;
        }
    };

    private final String symbol;

    Operation(String symbol) {
        this.symbol = symbol;
    }

    public abstract double apply(double a, double b);

    @Override
    public String toString() {
        return symbol;
    }
}
```

> Her sabit için ayrı bir gövde yazmak, arka planda her sabit için gizli bir anonim alt sınıf oluşturur — güçlü ama "ağır" bir özelliktir. Sadece birkaç sabit ve gerçekten farklılaşan davranış varsa kullan; onlarca sabit ve karmaşık mantık varsa ayrı strateji sınıfları daha okunur olabilir (bkz. bir sonraki bölüm).

EnumSet

`EnumSet`, yalnızca enum sabitleri için tasarlanmış, bit vektörü tabanlı çok verimli bir `Set` implementasyonudur — `HashSet<MyEnum>` kullanmaktan neredeyse her zaman daha hızlı ve daha az bellek harcar:

```
import java.util.EnumSet;

class EnumSetExample {
    public static void main(String[] args) {
        EnumSet<DayWithMethod> weekend = EnumSet.of(DayWithMethod.SATURDAY, DayWithMethod.
SUNDAY);
        EnumSet<DayWithMethod> weekdays = EnumSet.complementOf(weekend);

        System.out.println("Weekend: " + weekend);
        System.out.println("Weekdays: " + weekdays);
    }
}
```

Beklenen Çıktı

```
Weekend: [SATURDAY, SUNDAY]
Weekdays: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY]
```

EnumMap

`EnumMap<K,V>`, anahtarları enum olan bir `Map` implementasyonudur; içeride bir `HashMap` yerine, sabitlerin `ordinal()` değerine göre indekslenmiş bir dizi kullanır — bu da onu `HashMap<MyEnum, V>`'den belirgin şekilde hızlı yapar:

```
import java.util.EnumMap;
import java.util.Map;

class EnumMapExample {
    public static void main(String[] args) {
        EnumMap<DayWithMethod, String> plan = new EnumMap<>(DayWithMethod.class);
        plan.put(DayWithMethod.MONDAY, "Study Spring Boot");
        plan.put(DayWithMethod.SATURDAY, "Rest");

        for (Map.Entry<DayWithMethod, String> entry : plan.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
}
```

Beklenen Çıktı

```
MONDAY -> Study Spring Boot
SATURDAY -> Rest
```

> `EnumMap`, sabitleri her zaman `ordinal()` sırasına göre (yani tanımlandıkları sırayla) dolaşır — bu, `HashMap`'in aksine öngörülebilir bir iterasyon sırası ister, örneğin haftanın günlerini sırayla listelemek gibi durumlarda işine yarar.

Singleton Deseni

Effective Java kitabında Joshua Bloch'un önerdiği gibi, tek elemanlı bir enum, Java'da bir Singleton yazmanın en güvenli yoludur:

```
enum ConfigurationManager {
    INSTANCE;

    private String environment = "production";

    public String getEnvironment() {
        return environment;
    }

    public void setEnvironment(String environment) {
        this.environment = environment;
    }
}
```

Kullanım: `ConfigurationManager.INSTANCE.getEnvironment()`.

> Bu deseni bu kadar güvenli yapan şey, JVM'in enum sabitlerinin serialization ve reflection yoluyla bile ikinci bir örneğinin oluşturulmasına izin vermemesidir — klasik "private constructor + static getInstance()" Singleton'ında bu garantiler elle (ve hataya açık şekilde) sağlanmalıdır.

Strategy Pattern

Az önceki `Operation` örneği aslında bir Strategy Pattern uygulamasıydı: her sabit, aynı arayüzün (burada `apply` metodu) farklı bir "stratejisini" taşıyor. Bunu daha iş odaklı bir örnekle pekiştirelim:

```
enum ShippingStrategy {
    STANDARD {
        @Override
        public double calculate(double weightKg) {
            return weightKg * 2.5;
        }
    },
    EXPRESS {
        @Override
        public double calculate(double weightKg) {
            return weightKg * 5.0 + 10;
        }
    },
    SAME_DAY {
        @Override
        public double calculate(double weightKg) {
            return weightKg * 8.0 + 25;
        }
    };

    public abstract double calculate(double weightKg);
}
```

Klasik Strategy Pattern'de her strateji için ayrı bir sınıf ve bunları birbirine bağlayan bir `Factory` yazman gerekirdi; enum ile hem stratejiler hem de "hangi strateji hangi anahtara karşılık geliyor" eşlemesi tek bir yapıda, ekstra kod olmadan birleşiyor.

Gerçek Dünya Örnekleri

Enum'ların üretim kodunda en sık göründüğü yerlerden biri durum makineleridir (state machine). Bir siparişin geçerli durum geçişlerini enum ile modelleyelim:

```
import java.util.EnumSet;
import java.util.Set;

enum OrderStatus {
    PENDING,
    PAID,
    SHIPPED,
    DELIVERED,
    CANCELLED;

    private static final Set<OrderStatus> TERMINAL = EnumSet.of(DELIVERED, CANCELLED);

    public boolean isTerminal() {
        return TERMINAL.contains(this);
    }

    public boolean canTransitionTo(OrderStatus next) {
        return switch (this) {
            case PENDING -> next == PAID || next == CANCELLED;
            case PAID -> next == SHIPPED || next == CANCELLED;
            case SHIPPED -> next == DELIVERED;
            case DELIVERED, CANCELLED -> false;
        };
    }
}
```

Bu tür bir yapı, "hangi durumdan hangi duruma geçilebilir" kuralını dağınık `if/else` bloklarına değil, enum'ın kendisine yerleştirir — kural tek bir yerde yaşar ve derleyici, olası tüm durumların (`switch` exhaustiveness) ele alındığını garanti eder.

Diğer yaygın gerçek dünya kullanımları: HTTP durum kodu kategorileri, kullanıcı rolleri /yetkileri, ödeme yöntemleri, log seviyeleri (`DEBUG`, `INFO`, `WARN`, `ERROR`) — hepsi aynı desenin farklı uygulamalarıdır.

Mülakat Soruları

Enum, `int` sabitlere göre ne gibi avantajlar sağlar? Tip güvenliği (derleme zamanı kontrolü), okunabilirlik, `switch` ile exhaustiveness kontrolü ve `values()` / `valueOf()` gibi hazır API.

Bir enum başka bir sınıfı extend edebilir mi? Hayır — her enum örtük olarak `java.lang.Enum`'ı extend eder ve Java tek kalıtımı desteklediği için başka bir sınıfı extend edemez. Ancak istediği kadar arayüz implement edebilir.

`==` ile `equals()` enum sabitleri için aynı sonucu verir mi? Evet. Her enum sabiti JVM'de tek bir örnek olarak var olduğundan, `==` ile referans karşılaştırması `equals()` ile aynı sonucu üretir. Yine de gelenek olarak `equals()` tercih edilir.

`ordinal()`'ı neden veri saklamak için kullanmamalıyız? Çünkü sabitlerin sırası (dolayısıyla ordinal değerleri) kaynak kodda değişebilir; bu durumda önceden kaydedilmiş bir ordinal değeri artık yanlış sabiti işaret edebilir. Bunun yerine `name()` (ya da özel bir kod alanı) sakla.

Enum'lar `Serializable` mıdır? Evet, `java.lang.Enum` zaten `Serializable`'ı implement eder ve JVM enum serialization'ını özel olarak ele alır (sabit, `name()` üzerinden yeniden çözülür) — bu yüzden özel bir `readObject`/`writeObject` yazmana gerek kalmaz, hatta yazman önerilmez.

Bir enum `clone()`'lanabilir mi? Hayır. `Enum.clone()` `CloneNotSupportedException` fırlatır — çünkü her sabitin JVM'de tek bir örneği olması garanti edilmelidir; klonlama bu garantiyi bozar.