Record

Java'da sabit (immutable) veri taşıyıcıları tek satırda tanımlama.

Orta 45 dk

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

Record

Java'da **record**, sabit (immutable) veri taşımak için tasarlanmış, çoğu boilerplate'i derleyicinin senin yerine yazdığı özel bir sınıf türüdür. Bir DTO, bir değer nesnesi (value object) ya da bir API request/response modeli yazarken ihtiyaç duyduğun constructor, accessor, `equals()`, `hashCode()` ve `toString()`'i elle yazmak yerine, tek satırda tanımlarsın.

Record Nedir?

Bir record, "bu sınıfın tek işi birkaç değeri bir arada taşımak" dediğin her yerde kullanılır — bir koordinat (`x`, `y`), bir para tutarı (`amount`, `currency`), bir HTTP yanıtı (`status`, `body`) gibi. Bu tür sınıfları normal `class` ile yazmak, aynı beş metodu (constructor, getter'lar, `equals`, `hashCode`, `toString`) her seferinde elle ya da IDE ile üretmek anlamına gelir — record, bunu derleyiciye devreder.

Neden Eklendi?

Java'nın en sık eleştirilen yanlarından biri, basit bir veri taşıyıcı sınıf yazmanın bile ne kadar çok tekrar eden kod gerektirdiğiydi. Aynı beş metot, her alan eklendiğinde ya da değiştiğinde elle senkron tutulmak zorundaydı — biri `equals()`'ı güncellemeyi unutursa, sessizce hatalı bir karşılaştırma mantığıyla baş başa kalırdın. Record, bu senkronizasyon yükünü tamamen ortadan kaldırıyor: bileşenleri bir kere tanımlarsın, geri kalan her şey onlardan türetilir.

Tarihçe (Java 14 Preview → Java 16)

Record, JEP 359 ile Java 14'e önizleme (preview) özelliği olarak girdi, JEP 384 ile Java 15'te ikinci bir önizleme turundan geçti, ve JEP 395 ile Java 16'da kalıcı, standart bir dil özelliği hâline geldi. Yani bu projede kullandığımız Java 21, record'u sorunsuz ve tam destekle kullanabiliyor — herhangi bir preview flag'i gerekmiyor.

İlk Record'unu Yazmak

Bir record tanımlamak, aynı işi gören bir sınıfa göre çarpıcı derecede kısadır. Aşağıdaki tek satır, `x` ve `y` adında iki **bileşen** (component) taşıyan bir `Point` record'u tanımlar:

```
record Point(int x, int y) {
}
```

Bu tek satırla derleyiciye şunu söylemiş oluyorsun: "bu tipin tek işi, bir `x` ve bir `y` değerini bir arada, değişmez şekilde taşımak." Derleyici bunun karşılığında, aşağıdaki üyeleri **senin yerine** üretir (her birini ilerleyen "Üretilen Üyeler" bölümünde tek tek inceleyeceğiz):

- Her iki bileşeni de parametre olarak alan bir **canonical constructor**
- `x()` ve `y()` adında, bileşenlerle aynı isme sahip **accessor** metotları
- Tüm bileşenleri karşılaştıran bir `equals()`
- Bileşenlerle tutarlı bir `hashCode()`
- `Point[x=.., y=..]` biçiminde okunaklı bir `toString()`

Kullanımı sıradan bir sınıfla birebir aynıdır — `new` ile örnek oluşturursun, metotlarını çağırırsın:

```
class PointUsage {
    public static void main(String[] args) {
        Point p1 = new Point(3, 4);
        Point p2 = new Point(3, 4);

        System.out.println(p1);                          // Point[x=3, y=4]
        System.out.println("x koordinati: " + p1.x());   // x koordinati: 3
        System.out.println(p1.equals(p2));                // true
        System.out.println(p1 == p2);                     // false — two different objects
    }
}
```

> `x()` ve `y()` — dikkat, bunlar `getX()` / `getY()` **değil**. Record accessor'ları, Java Bean konvansiyonunu değil, doğrudan bileşen adının kendisini kullanır. Bu, record'ları fonksiyonel dillerdeki "data" tiplerine yaklaştıran bilinçli bir tasarım kararıdır ve bir sonraki bölümde (Record vs Class) bunun neden önemli olduğuna değineceğiz.

> `record`, Java 16'dan beri bir **keyword**'dür ama *rezerve edilmiş* bir kelime değildir — yani hâlâ değişken ya da metot adı olarak `record` kullanabilirsin (`var record = ...` derlenir). Java bunu yalnızca bir tip tanımının başında, bağlama duyarlı (contextual) bir keyword olarak yorumlar; bu, mevcut kod tabanlarının geriye dönük uyumluluğunu bozmamak için bilinçli bir tasarım tercihidir.

Record vs Class

"İlk Record'unu Yazmak" bölümündeki `Point` örneğinin, klasik bir `class` ile yazılsaydı nasıl görüneceğine bakalım. Aynı iki alanı (`name` , `age`) taşıyan, değişmez (immutable) bir `PersonClassic` sınıfı, elle yazıldığında şöyle olurdu:

```
final class PersonClassic {

    private final String name;
    private final int age;

    PersonClassic(String name, int age) {
        this.name = name;
        this.age = age;
    }

    String name() {
        return name;
    }

    int age() {
        return age;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof PersonClassic other)) return false;
        return age == other.age && name.equals(other.name);
    }

    @Override
    public int hashCode() {
        return java.util.Objects.hash(name, age);
    }

    @Override
    public String toString() {
        return "PersonClassic[name=" + name + ", age=" + age + "]";
    }
}
```

Aynı davranışı bir record ile tanımlamak tek satır:

```
record PersonRecord(String name, int age) {
}
```

Yirmi küsur satırlık boilerplate, tek satıra iniyor — ama fark yalnızca satır sayısından ibaret değil. Record tanımı, derleyiciye ekstra garantiler de veriyor:

- Bir record **örtük olarak** `final` 'dır — başka bir sınıf tarafından extend edilemez. `PersonClassic` 'i `final` yapmak bizim tercihimizdi (ve immutability için doğru bir tercihti); `PersonRecord` için bu, dilin kendisi tarafından dayatılıyor.

- Bir record, örtük olarak `java.lang.Record`'ı extend eder — tıpkı her enum'un örtük olarak `java.lang.Enum`'ı extend etmesi gibi. Java'da tek kalıtım geçerli olduğu için bu, bir record'un **başka bir sınıfı extend edemeyeceği** anlamına gelir (arayüz implement etmek serbesttir — bkz. "Arayüz İmplementasyonu" bölümü).
- Bileşenlere karşılık gelen alanlar örtük olarak `private final`'dır — `PersonClassic`'te bunu elle yazdık, record'da bu bir seçenek değil, kuraldır.
- Record için setter **üretilmez** — yalnızca accessor'lar (`name()`, `age()`) vardır. Değişmezliği garanti altına almanın bir parçası bu.

> `PersonClassic`'e üçüncü bir alan (örneğin `email`) eklemeyi düşün: alan tanımını, constructor parametresini, atamasını, accessor'ı, hem `equals()` hem `hashCode()` hem de `toString()`'i — toplam **altı** yeri elle güncellemen gerekir. `PersonRecord`'da tek yapman gereken, bileşen listesine `String email` eklemek; geri kalan her şey otomatik senkronize kalır. Bu, "Neden Eklendi?" bölümünde bahsettiğimiz senkronizasyon yükünün tam olarak ortadan kalktığı yer.

> Record'un JPA/Hibernate **entity**'si olamayacağını unutma — JPA, proxy oluşturmak için parametresiz bir constructor ve `final` olmayan bir sınıf ister; ikisi de record'un doğasına aykırı. Record'lar bunun yerine DTO, request/response modeli ve read-only projeksiyonlar için idealdir (bkz. "Gerçek Dünya Örnekleri" bölümü) — entity'ler hâlâ normal `class` olarak kalmalı.

Bileşenler (Components)

Record tanımının parantez içindeki listesine (`(String name, int age)`) **bileşen listesi** denir; her bileşen üç şeyi aynı anda temsil eder: bir `private final` alan, bir accessor metodu ve canonical constructor'daki bir parametre.

Bileşen sayısında bir sınır yok — sıfır bileşenli bir record bile geçerlidir (genellikle bir "marker" ya da tekil bir olay/sinyal temsil etmek için kullanılır):

```
record Heartbeat() {
}
```

Bileşenler herhangi bir tip olabilir — primitive, referans tipi, generic tip parametresi ya da dizi (diziyle ilgili tuzağı bir sonraki bölümde, Immutability'de göreceğiz). Generic bir record örneği:

```
record Pair<A, B>(A first, B second) {
}
```

```
Pair<String, Integer> kisi = new Pair<>("Ayşe", 30);
System.out.println(kisi); // Pair[first=Ayşe, second=30]
```

> Bileşenlere eklenen bir annotation, Java'nın "annotation hedefleri" (target) uygunsa, derleyici tarafından otomatik olarak alana, constructor parametresine **ve** accessor metoduna da uygulanır — örneğin `record User(@NotBlank String username) {}` yazdığında, Bean Validation `@NotBlank` hem alanda hem parametrede hem de `username()` üzerinde etkili olur, tek bir yere yazmana rağmen.

Üretilen Üyeler (Derinlemesine)

"İlk Record'unu Yazmak" bölümünde kısaca listelediğimiz üretilen üyelere şimdi tek tek, tam olarak ne yaptıklarını görecek şekilde bakalım.

Canonical constructor, tüm bileşenleri, tanımlandıkları sırayla parametre olarak alır ve her birini aynı isimdeki alana atar — `Point(int x, int y)` durumunda tam olarak `this.x = x; this.y = y;` yapar, başka hiçbir şey.

Accessor'lar, her bileşen için, bileşenle aynı isimde (Java Bean `get` önekiyle değil) üretilir ve alanın değerini doğrudan döner.

`equals()`, iki record örneğinin **aynı sınıftan** olup olmadığını kontrol eder, sonra her bileşeni sırayla karşılaştırır. Referans tipli bileşenler için `equals()`, primitive tipli bileşenler için `==` kullanılır — **tek istisna** `float` / `double`'dır: bunlar için `==` değil, `Float.compare()` / `Double.compare()` semantiği kullanılır:

```
record Measurement(double value) {
}

class EqualsSemanticsExample {
    public static void main(String[] args) {
        Measurement a = new Measurement(Double.NaN);
        Measurement b = new Measurement(Double.NaN);

        System.out.println(Double.NaN == Double.NaN);   // false — primitive ==
        System.out.println(a.equals(b));                // true  — Double.compare semantics
        System.out.println(a.value() == b.value());      // false — the accessor still
returns a primitive double
    }
}
```

Beklenen Çıktı

```
false
true
false
```

> Bu istisna önemli, çünkü `Double.compare()` semantiğinde `NaN`, kendisine **eşittir** (`Double.NaN == Double.NaN` primitive `==` ile `false` döner, ama record'un ürettiği `equals()` içinde `true` döner) — çünkü record, iki bileşeni `Double.compare(a, b) == 0` ile karşılaştırır, çıplak `==` ile değil. Bu, `Double.equals()`'ın da izlediği aynı semantiktir; yani record burada Java'nın kutulanmış (boxed) `Double` davranışıyla tutarlı, ama çıplak `double == double` alışkanlığından farklı davranıyor.

`hashCode()`, tüm bileşenlerin hash değerlerini (belirtilmemiş ama `equals()` ile tutarlı bir algoritmayla) birleştirir — iki eşit record her zaman aynı `hashCode()`'u döner, ki bu `HashMap`/`HashSet` gibi koleksiyonlarda doğru çalışmak için zorunludur.

`toString()`, `KayıtAdı[bileşen1=değer1, bileşen2=değer2]` biçiminde, sınıfın basit adını (paket öneki olmadan) ve tüm bileşenleri sırayla listeler — hata ayıklarken (debug) ekstra bir `toString()` yazma ihtiyacını neredeyse tamamen ortadan kaldırır.

Immutability (Değişmezlik)

Record'ların "immutable" olması, sık yanlış anlaşılan bir konudur: record garanti ettiği şey, **kendi referanslarının** (bileşenlerinin) değiştirilemeyeceğidir — `final` alanlar, setter yokluğu. Ama bir bileşen **mutable bir nesneye referans** tutuyorsa, o nesnenin içeriği record'un dışından hâlâ değiştirilebilir. Buna "shallow immutability" (sığ değişmezlik) denir:

```
record Team(java.util.List<String> members) {
}

class TeamMutableTrap {
    public static void main(String[] args) {
        java.util.List<String> names = new java.util.ArrayList<>();
        names.add("Ada");
        names.add("Grace");

        Team team = new Team(names);
        System.out.println(team); // Team[members=[Ada, Grace]]

        // The content changes through the externally held reference, without
        // ever going through Team's own API:
        names.add("Linus");

        System.out.println(team); // Team[members=[Ada, Grace, Linus]] -- the "immutable"
object changed!
    }
}
```

Yukarıdaki örnekte `Team` record'unun kendisi değişmez görünür — ama `members` alanına atanan `ArrayList` referansı, çağıran kod tarafından hâlâ elde tutulup sonradan değiştirilebiliyor; bu da `Team` örneğinin *içeriğinin*, `Team`'in kendi API'si hiç kullanılmadan değişmesine yol açıyor.

Bunun standart çözümü, **compact constructor** içinde savunmacı bir kopya (defensive copy) almaktır — `List.copyOf()`, hem kopyalar hem de sonucu değiştirilemez (unmodifiable) hale getirir:

```
record Team(java.util.List<String> members) {

    // Compact constructor: no need to repeat the parameter list, just the
    // defensive copy. The assignment (this.members = members;) is done
    // implicitly by the compiler after this block.
    Team {
        members = java.util.List.copyOf(members);
    }
}

class TeamDefensiveCopyUsage {
    public static void main(String[] args) {
        java.util.List<String> names = new java.util.ArrayList<>();
        names.add("Ada");
        names.add("Grace");

        Team team = new Team(names);
        names.add("Linus"); // no longer has any effect on team.members()

        System.out.println(team); // Team[members=[Ada, Grace]]

        try {
            team.members().add("Dennis"); // UnsupportedOperationException
        } catch (UnsupportedOperationException e) {
            System.out.println("cannot be modified via members(): " + e.getClass().
getSimpleName());
        }
    }
}
```

> Bileşen olarak **dizi (array)** kullanmaktan kaçın. İki sebebi var: (1) diziler her zaman mutable'dır, yukarıdaki `List` tuzağının aynısı dizilerde de geçerlidir — üstelik `List.copyOf()`'un dizi karşılığı yoktur, elle `clone()`'lamak gerekir. (2) Daha da sinsi olanı: bir dizi bileşeni için üretilen `equals()`, `Arrays.equals()` **değil**, `Object.equals()`'ı (yani referans eşitliğini) kullanır — iki record aynı elemanlara sahip iki *farklı* dizi tutuyorsa `equals()` `false` döner, `toString()` da `[I@1b6d3586` gibi anlamsız bir çıktı üretir. Dizi yerine neredeyse her zaman `List` tercih et; dizi zorunluysa, `equals()` / `hashCode()` / `toString()`'i elle override et.

> `List.copyOf()`, verilen listede `null` eleman varsa `NullPointerException` fırlatır — bu da compact constructor'ı, hem savunmacı kopyalama hem de örtük bir null-eleman doğrulaması için tek bir satırda kullanmanı sağlar.

Constructors (Canonical, Compact, Validation)

"Üretilen Üyeler" bölümünde gördüğümüz canonical constructor'ı, **elle de yazabilirsin** — bunu genellikle doğrulama (validation) veya normalizasyon eklemek için yaparsın. İki şekilde yazılabilir.

Tam (explicit) canonical constructor, tüm parametreleri tekrar yazıp atamaları elle yapar — üretilenin birebir aynı imzasına sahip olmalı:

```
record Range(int min, int max) {
    Range(int min, int max) {
        if (min > max) {
            throw new IllegalArgumentException("min (" + min + ") max'tan (" + max + ")
büyük olamaz");
        }
        this.min = min;
        this.max = max;
    }
}
```

Compact constructor, parametre listesini ve atamaları tekrar yazmadan, yalnızca doğrulama/normalizasyon mantığını yazmanı sağlar — atamalar, derleyici tarafından blok sonunda **örtük olarak** yapılır:

```
record PersonValidated(String name, int age) {

    // Compact constructor: validation and normalization happen here.
    PersonValidated {
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException("name must not be blank");
        }
        if (age < 0 || age > 150) {
            throw new IllegalArgumentException("invalid age: " + age);
        }
        name = name.trim(); // reassigning the parameter — not the field
    }
}
```

> Compact constructor içinde parametrelere (örneğin `name = name.trim();`) yeniden atama yapabilirsin — bu, alana değil, henüz alana atanmamış yerel parametre değişkenine atamadır; derleyici bu güncellenmiş değeri, bloktan sonra örtük atamada kullanır. Alanın kendisine (`this.name = ...`) compact constructor içinde **doğrudan atama yapamazsın** — derleyici bunu hata verir, çünkü örtük atama zaten senin için yapılacaktır.

Bir record, canonical constructor'a **ek olarak** başka constructor'lar da tanımlayabilir — ama bunların ilk satırı, `this(...)` ile mutlaka canonical constructor'ı (doğrudan ya da zincirleme) çağırmak zorundadır:

```
record PersonOverloadedConstructor(String name, int age) {

    // Extra constructor: delegates to the canonical constructor with a default
    // age of 0 when the age is unknown. The first line must always be this(...).
    PersonOverloadedConstructor(String name) {
        this(name, 0);
    }
}
```

> Bu zorunluluk bilinçli bir tasarım kararı: normal bir `class`'ta, bir constructor doğrulamayı atlayıp alanlara doğrudan atama yapabilir — bu da bazı nesnelerin geçersiz durumda oluşturulmasına izin verebilir. Record'da **her yol** canonical constructor'dan geçmek zorunda olduğu için, orada yazdığın doğrulama, o record'u oluşturmanın **hiçbir** yolu tarafından atlanamaz.

Özel Metotlar (Custom Methods)

Bir record'un gövdesi, accessor'lara ek olarak sıradan instance metotları da barındırabilir — tıpkı bir sınıfta olduğu gibi:

```
record Rectangle(double width, double height) {

    double area() {
        return width * height;
    }

    double perimeter() {
        return 2 * (width + height);
    }

    boolean isSquare() {
        return width == height;
    }
}
```

Ayrıca, üretilen bir accessor'ı **override** edebilirsin — örneğin mutable bir bileşeni dışarı verirken savunmacı bir kopya döndürmek için (Immutability bölümündeki `List.copyOf()` deseninin bir alternatifi: constructor'da değil, okuma anında kopyalamak):

```
record Snapshot(List<String> items) {
    List<String> items() {
        return List.copyOf(items); // her çağrıda değişmez bir kopya
    }
}
```

> Bir record'un gövdesine, bileşen listesinde olmayan **ekstra bir instance alanı** ekleyemezsin — bu derleme hatasıdır. `record Point(int x, int y) { private int z; }` derlenmez. Bunun nedeni tutarlılık: record'un "durumu" tamamen bileşen listesinden ibarettir, `equals()` / `hashCode()` / `toString()` de yalnızca o listeyi baz alır — gizli, bu üyelerin görmediği bir alan olması, record'un temel garantisini bozar. (Static alanlar bu kısıtlamaya tabi değildir — bir sonraki bölüm.)

Static Üyeler

Diğer sınırlamaların aksine, bir record **static** alan, metot ve initializer block barındırma konusunda tamamen sıradan bir sınıf gibi davranır. En yaygın kullanım, static factory metotları ve önceden tanımlı sabitlerdir:

```
record PointWithFactory(int x, int y) {

    static final PointWithFactory ORIGIN = new PointWithFactory(0, 0);

    static PointWithFactory origin() {
        return ORIGIN;
    }

    static PointWithFactory of(int x, int y) {
        return new PointWithFactory(x, y);
    }
}
```

`PointWithFactory.origin()` gibi bir factory metodu, `new PointWithFactory(0, 0)` yazmaktan daha okunaklıdır ve niyeti (intent) açıkça ifade eder — özellikle sık kullanılan "özel" değerler için tercih edilir.

Arayüz İmplementasyonu

"Record vs Class" bölümünde bir record'un başka bir sınıfı extend edemeyeceğini gördük (`java.lang.Record` zaten extend edilmiş durumda) — ama enum'larda olduğu gibi, istediğin kadar **arayüz** implement edebilirsin. En sık görülen örneklerden biri `Comparable<T>`:

```
record ComparablePointExample(int x, int y) implements Comparable<ComparablePointExample> {

    // Ordering rule: x first, then y if equal (simple lexicographic order).
    @Override
    public int compareTo(ComparablePointExample other) {
        int byX = Integer.compare(x, other.x);
        return byX != 0 ? byX : Integer.compare(y, other.y);
    }
}
```

> Record + arayüz kombinasyonu, `sealed interface` ile birleştiğinde özellikle güçlü hâle gelir — örneğin bir `sealed interface Shape permits Circle, Rectangle {}` tanımlayıp her alt tipi bir record olarak yazabilirsin. Bu deseni ve modern `switch` ile nasıl kullanılacağını, ileride ayrı bir bölüm olarak ele alacağımız **Record Patterns** bölümünde derinlemesine işleyeceğiz.

İç İçe Record'lar (Nested Records)

Bir record, başka bir record'un (ya da sınıfın) içinde tanımlanabilir. Enum'larda olduğu gibi, iç içe (nested) bir record **örtük olarak** `static`'tir — dış sınıfın bir örneğine ihtiyaç duymadan kullanılabilir, çünkü zaten `static` olmayan iç record tanımlamak mümkün değildir:

```
record Address(String city, String zip) {
}

record Employee(String name, Address address) {
}

class NestedRecordExampleUsage {
    public static void main(String[] args) {
        Employee e1 = new Employee("Ada", new Address("Istanbul", "34000"));
        Employee e2 = new Employee("Ada", new Address("Istanbul", "34000"));

        System.out.println(e1);              // Employee[name=Ada, address=Address
[city=Istanbul, zip=34000]]
        System.out.println(e1.equals(e2));    // true — the nested Address.equals() is used
too
        System.out.println(e1.address().city()); // Istanbul
    }
}
```

`Employee` record'unun `address()` accessor'ı, `Address` tipinde bir değer döner; bu da `equals()` / `hashCode()` / `toString()` zincirinin doğal olarak **iç içe** çalıştığı anlamına gelir — `Address`'in kendi `equals()`'ı doğru yazıldığı sürece (ki bir record için bu otomatik), `Employee.equals()` de doğru çalışır, çünkü bileşen karşılaştırması `Address` için `Address.equals()`'ı çağırır.

Serialization ve Reflection

Enum'ın aksine, bir record **otomatik olarak** `Serializable` değildir — bunu, tıpkı sıradan bir sınıfta olduğu gibi açıkça belirtmen gerekir. Belirttiğinde ise, record'a özgü önemli bir fark ortaya çıkar: deserialization, klasik Java serialization'ının aksine reflection ile alanları doğrudan doldurmaz, **canonical constructor'ı çağırır**:

```
import java.io.*;

record Score(String player, int points) implements Serializable {

    // The validation in the compact constructor also runs during deserialization --
    // unlike a classic class, where a custom readObject() could bypass it.
    Score {
        if (points < 0) {
            throw new IllegalArgumentException("points negatif olamaz: " + points);
        }
    }
}

class SerializableRecordExampleUsage {
    public static void main(String[] args) throws Exception {
        Score original = new Score("Ada", 100);

        ByteArrayOutputStream bytes = new ByteArrayOutputStream();
        try (ObjectOutputStream out = new ObjectOutputStream(bytes)) {
            out.writeObject(original);
        }

        Score restored;
        try (ObjectInputStream in = new ObjectInputStream(new ByteArrayInputStream(bytes.
toByteArray()))) {
            restored = (Score) in.readObject();
        }

        System.out.println(restored);                  // Score[player=Ada, points=100]
        System.out.println(original.equals(restored));  // true
    }
}
```

> Bu, pratikte önemli bir güvenlik avantajı sağlar: "Constructors (Canonical, Compact, Validation)" bölümündeki compact constructor doğrulaman (örneğin `points < 0` kontrolü), deserialization sırasında da **atlanamaz**. Klasik bir `Serializable` sınıfta, elle yazılmış bir `readObject()` olmadığı sürece, saldırgan kontrollü bir byte akışı doğrulamaları by-pass ederek geçersiz bir nesne "inşa edebilir" — record'da bu yol baştan kapalı.

Reflection API de record'lara özel iki yeni araç sunar: `Class.isRecord()` ve `Class.getRecordComponents()` — ikincisi, bileşenlerin adlarını ve tiplerini çalışma zamanında listelemeni sağlar (JSON serializer'lar, ORM'ler ve validation kütüphaneleri tam olarak bunu kullanarak record'ları tanır):

```
import java.lang.reflect.RecordComponent;

record Score2(String player, int points) {
}

class ReflectionExample {
    public static void main(String[] args) {
        Class<Score2> type = Score2.class;

        System.out.println(type.isRecord()); // true

        for (RecordComponent component : type.getRecordComponents()) {
            System.out.println(component.getName() + " : " + component.getType().
getSimpleName());
        }
        // player : String
        // points : int
    }
}
```

Beklenen Çıktı

```
true
player : String
points : int
```

> `getRecordComponents()`, sınıf bir record **değilse** boş bir dizi değil, `null` döner — çağırmadan önce mutlaka `isRecord()` ile kontrol et, aksi halde beklenmedik bir `NullPointerException` alabilirsin.

Best Practices

Şimdiye kadar gördüklerimizi, ne zaman record kullanıp ne zaman kullanmaman gerektiğine dair somut önerilere dönüştürelim.

Record kullan:

- DTO'lar, request/response modelleri (bkz. "Gerçek Dünya Örnekleri")
- Değer nesneleri (value object) — para tutarı, koordinat, aralık (range)
- `switch` / `instanceof` ile örüntü eşleştirmesi (pattern matching) yapacağın veri modelleri — bkz. **Record Patterns** eki
- Bir metottan birden fazla değer döndürmen gerektiğinde (özel bir "sonuç" tipi olarak)

Record kullanma:

- JPA/Hibernate entity'leri ("Record vs Class" bölümünde neden değindik)
- İç durumu zamanla değişmesi gereken nesneler (örneğin bir builder'ın kendisi, bir önbellek girişi sayaç tutuyorsa)

- Çok sayıda (6-7'den fazla) bileşeni olan yapılar — bu genelde "ilgili alanları bir alt record'a grupla" sinyalidir (bkz. "İç İçe Record'lar")

Tasarım önerileri:

- Mutable bileşenler (`List` , `Map` , `Set`) için her zaman compact constructor'da savunmacı kopya al (bkz. "Immutability")
- Karmaşık doğrulama mantığını compact constructor'da tut, çağıran kodun sorumluluğuna bırakma
- Dizi bileşenlerden kaçın (bkz. "Immutability")
- İsimlendirmede `Record` son ekine gerek yok — bu derste birden fazla `Point` çeşidini ayırt edebilmek için (`PersonClassic` / `PersonRecord` gibi) kullandık, gerçek kodda yalnızca `Person` , `Point` , `Range` gibi düz isimler tercih edilir

> Proje Lombok kullanıyorsa, `@Data` / `@Value` ile record arasında seçim yapman gerekebilir — kısa cevap: yeni yazdığın, gerçekten immutable olması gereken, harici bir annotation processor'a ihtiyaç duymadan derlenmesini istediğin veri tipleri için **record**; JPA entity'leri gibi mutable kalması gereken ya da record'un desteklemediği (`@Builder` gibi) özelliklere ihtiyaç duyan sınıflar için **Lombok**. Ayrıntılı karşılaştırmayı "Record vs Lombok" ekinde işliyoruz.

Yaygın Hatalar

Buraya kadar tek tek karşılaştığımız tuzakları toparlayalım, artı birkaç yeni tanesini ekleyelim.

1. Bileşen olarak dizi kullanmak. `equals()` , dizi bileşenler için referans eşitliği kullanır, `Arrays.equals()` değil — "Immutability" bölümünde detaylandırdık.

2. Mutable bir nesneyi olduğu gibi saklamak. Çağıran kodun elinde tuttuğu bir `ArrayList` / `HashMap` referansını compact constructor'da kopyalamadan almak, record'un "değişmezlik" garantisini görünüşte bırakır ama gerçekte delik bırakır.

3. Record'u JPA entity yapmaya çalışmak. Parametresiz constructor ve mutable alan gereksinimleri, record'un doğasıyla çelişir.

4. `getX()` / `getY()` beklemek. Record accessor'ları bileşenin adını kullanır, Java Bean önekini değil — "İlk Record'unu Yazmak" bölümünde değindik.

5. Farklı record tiplerini "şekil aynı" diye eşit sanmak. `equals()` , önce çalışma zamanı sınıfının **birebir aynı** olup olmadığını kontrol eder — iki record'un bileşenleri aynı görünse bile, tipleri farklıysa asla eşit değildir:

```
record Point(int x, int y) {}
record Coordinate(int x, int y) {}

Point p = new Point(1, 2);
Coordinate c = new Coordinate(1, 2);
System.out.println(p.equals(c)); // false — Coordinate bir Point değil
```

6. Record'u "sonra extend ederiz" diye tasarlamak. Record'lar örtük olarak `final` olduğu için genişletilemez (bkz. "Record vs Class") — ortak davranış paylaşmak istiyorsan composition (bir record'u başka bir record'un bileşeni yapmak, bkz. "İç İçe Record'lar") ya da ortak bir arayüz implement etmek (bkz. "Arayüz İmplementasyonu") doğru yoldur.

Gerçek Dünya Örnekleri

Record'ların en doğal yaşam alanı, bir Spring Boot uygulamasının **sınır katmanı** (controller'lar) ve **okuma modelleri**dir. Bir kullanıcı oluşturma isteği ve yanıtı, tipik olarak şöyle modellenir:

```
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;

// As we saw in the "Components" section, validation annotations are automatically
// applied to the field, the constructor parameter, and the accessor alike.
record CreateUserRequest(
        @NotBlank String fullName,
        @NotBlank @Email String email
) {
}
```

```
import java.time.Instant;

record UserResponse(Long id, String fullName, String email, Instant createdAt) {

    // Static factory: a common pattern for keeping the entity -> response
    // conversion in one place (the same pattern we saw in "Static Members").
    static UserResponse from(Long id, String fullName, String email, Instant createdAt) {
        return new UserResponse(id, fullName, email, createdAt);
    }
}
```

Bir controller'da kullanımı, sıradan bir sınıfla birebir aynıdır — Spring, `@RequestBody` için record'ları da normal bir sınıf gibi (Jackson üzerinden) deserialize eder:

```
import jakarta.validation.Valid;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import java.time.Instant;

@RestController
class UserController {

    // In a real application a UserService would be injected here; to keep the
    // example simple, we build the UserResponse directly.
    @PostMapping("/api/users")
    ResponseEntity<UserResponse> create(@Valid @RequestBody CreateUserRequest request) {
        UserResponse response = UserResponse.from(1L, request.fullName(), request.email(),
Instant.now());
        return ResponseEntity.ok(response);
    }
}
```

`@Valid` ile bileşen üzerindeki `@NotBlank` / `@Email` gibi kısıtların tetiklenmesi için classpath'te `spring-boot-starter-validation` bağımlılığının bulunması gerekir — `spring-boot-starter-web` bunu otomatik getirmez.

Jackson (Spring Boot'un varsayılan JSON kütüphanesi), bir record'un **tek** constructor'ı olan canonical constructor'ı otomatik olarak tanır ve JSON alanlarını doğrudan ona eşler — `@JsonCreator` ya da `@JsonProperty` eklemene genellikle gerek kalmaz. Bu, record'ları request/response DTO'su olarak kullanmayı, elle yazılmış bir POJO'ya göre hem daha az kod hem daha az "unutma" riski hâline getirir.

Spring Data JPA, `@Query` ile yazılan JPQL constructor expression'larında (`SELECT new com.cdurgun.learning.dto.UserSummary(u.id, u.email) FROM User u`) ve interface tabanlı projection'larda hedef tip olarak da record kabul eder — entity'nin kendisini record yapamasak bile, ondan türetilen **salt okunur bir görünümü** (view/projection) rahatlıkla record ile modelleyebiliriz.

Mülakat Soruları

Record nedir, normal bir class'tan temel farkı nedir? Record, sabit veri taşımak için tasarlanmış özel bir sınıf türüdür; constructor, accessor, `equals()`, `hashCode()` ve `toString()`'i derleyici otomatik üretir. Örtük olarak `final`'dır, `java.lang.Record`'ı extend eder ve tüm alanları `private final`'dır.

Bir record neden JPA entity'si olarak kullanılamaz? JPA, proxy oluşturmak için parametresiz bir constructor ve `final` olmayan, mutable bir sınıf gerektirir; record'un tasarımı (immutable, tek constructor, örtük `final`) bu gereksinimlerle doğrudan çelişir.

Compact constructor nedir, ne zaman kullanılır? Parametre listesini ve atamaları tekrarlamadan, yalnızca doğrulama/normalizasyon mantığı yazmanı sağlayan özel bir canonical constructor biçimidir; atamalar blok bittikten sonra derleyici tarafından örtük olarak yapılır. Genellikle giriş doğrulama ve savunmacı kopyalama (`List.copyOf()`) için kullanılır.

Record'lar `Serializable` mıdır? Hayır, enum'ların aksine otomatik değildir — açıkça `implements Serializable` yazman gerekir. Yazdığında, deserialization sıradan sınıflardan farklı olarak canonical constructor'ı çağırır, bu da compact constructor'daki doğrulamanın deserialization'da da çalışmasını garanti eder.

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

Bir record'un gövdesine ekstra bir instance alanı ekleyebilir misin? Hayır, bu derleme hatasıdır — record'un durumu tamamen bileşen listesinden ibarettir. Static alanlar bu kısıtlamaya tabi değildir.

Record'lar ile sealed interface'lerin ilişkisi nedir? Bir `sealed interface`'in izin verilen (permits) tüm alt tiplerini record olarak tanımlamak, modern `switch` ile örüntü eşleştirmesinde (pattern matching) derleyicinin tüm olası durumların ele alındığını garanti etmesini sağlar — ayrıntısını **Record Patterns** ekinde işliyoruz.

Özet ve Cheat Sheet

Record, Java 16 ile kalıcı hâle gelen, sabit veri taşıyıcıları için tek satırlık bir tanımın; constructor, accessor, `equals()`, `hashCode()` ve `toString()`'i derleyiciye devrettiği özel bir sınıf türü. Öne çıkan noktalar:

- Bileşen listesi = alan + accessor + canonical constructor parametresi, hepsi bir arada
- Örtük olarak `final`, `java.lang.Record`'ı extend eder, tüm alanları `private final`
- `equals()` / `hashCode()` / `toString()` bileşenlere göre otomatik üretilir (`float` / `double` için `Float.compare()` / `Double.compare()` semantiği)
- Immutability **sığdır** (shallow) — mutable bileşenler için compact constructor'da savunmacı kopya al, dizi bileşenlerden kaçın
- Doğrulama/normalizasyon için compact constructor; ek constructor'lar mutlaka canonical'a delege etmek zorunda
- Static alan/metot serbest, ekstra **instance** alanı yasak
- Arayüz implement edebilir, extend edemez
- İç içe tanımlanabilir (örtük `static`)

- `Serializable` otomatik değildir; belirtilirse deserialization canonical constructor'ı çağırır
- İdeal kullanım alanı: DTO, request/response, value object, pattern matching hedefleri
- Kaçınılması gereken alan: JPA entity, mutable durum gerektiren sınıflar

Hızlı referans:

```
// Temel tanım
record Point(int x, int y) {}

// Compact constructor (doğrulama/normalizasyon)
record Point(int x, int y) {
    Point {
        if (x < 0 || y < 0) throw new IllegalArgumentException("negatif olamaz");
    }
}

// Static factory + sabit
record Point(int x, int y) {
    static final Point ORIGIN = new Point(0, 0);
    static Point of(int x, int y) { return new Point(x, y); }
}

// Arayüz implementasyonu
record Point(int x, int y) implements Comparable<Point> {
    public int compareTo(Point o) { return Integer.compare(x, o.x); }
}

// Generic record
record Pair<A, B>(A first, B second) {}

// İç içe record
record Address(String city, String zip) {}
record Employee(String name, Address address) {}
```

Ek: Record vs Lombok

Bu projenin bağımlılıkları arasında zaten Lombok var — o yüzden "neden `@Data` ya da `@Value` yerine record yazdık?" sorusunu somut bir karşılaştırmayla cevaplayalım.

Lombok, derleme sırasında çalışan bir **annotation processor**'dır: kaynak kodunu değiştirmeden, `.class` dosyasına constructor/getter/setter/`equals()` gibi üyeleri enjekte eder. `@Value`, amaç olarak record'a en yakın olanıdır — immutable bir sınıf üretir:

```
// Lombok ile: @Value, immutable bir sınıf üretir
import lombok.Value;

@Value
public class PersonLombok {
    String name;
    int age;
}

// Record ile: aynı sonuç, dil düzeyinde
record PersonRecord(String name, int age) {
}
```

İkisi de sonuçta constructor, accessor, `equals()`, `hashCode()`, `toString()` üretir — ama altta yatan mekanizma ve esneklik önemli noktalarda ayrışıyor:

- **Mekanizma:** Record, `javac`'ın kendisinin bir parçasıdır — hiçbir ek bağımlılık ya da IDE eklentisi gerekmez. Lombok harici bir kütüphanedir; IDE'nin kodu doğru göstermesi için Lombok eklentisinin kurulu olması gerekir (bu projede zaten kurulu olduğunu varsayıyoruz, ama yeni bir katkıda bulunan için ekstra bir kurulum adımıdır).
- **Accessor ismi:** Record `name()` üretir; Lombok `@Value` / `@Data` ise Java Bean konvansiyonuna uyarak `getName()` üretir. "İlk Record'unu Yazmak" bölümünde değindiğimiz gibi bu bilinçli bir tasarım farkı — Lombok, Bean tabanlı eski framework'lerle (bazı reflection tabanlı serializer'lar, form binding kütüphaneleri) uyumluluğu önceliklendirir.
- **Mutability seçeneği:** Record her zaman immutable'dır. Lombok'ta bu bir seçimdir — `@Value` immutable, `@Data` ise mutable (getter **ve** setter) üretir. Yani Lombok, hem immutable hem mutable veri sınıfları için tek bir araçtır.
- **Kalıtım:** Record örtük olarak `final`'dır, extend edilemez. Lombok'un ürettiği sınıf sıradan bir sınıftır — istersen extend edebilir, ek alan/metot ekleyebilirsin (immutable garantisini bozma pahasına).
- **Builder:** Lombok'un `@Builder`'ı, çok bileşenli nesneler için hazır bir builder API'si verir. Record'da yerleşik bir builder yok — ihtiyaç varsa elle yazman ya da ayrı bir annotation processor (örneğin harici bir "record builder" kütüphanesi) eklemen gerekir.
- **Pattern matching:** Yalnızca gerçek record'lar, Java'nın `switch` / `instanceof` örüntü eşleştirmesiyle (bkz. **Record Patterns** eki) doğrudan çalışır — Lombok'un ürettiği sınıflar bu mekanizmadan yararlanamaz, çünkü derleyici onları "record" olarak tanımaz.
- **Doğrulama garantisi:** "Constructors (Canonical, Compact, Validation)" bölümünde gördüğümüz gibi, record'da ek constructor'lar mutlaka canonical'a delege etmek zorundadır — tek giriş noktası derleyici tarafından zorlanır. Lombok'ta benzer bir garanti için elle constructor yazıp `@Value`'i alanlara uygulaman gerekir, bu da disipline dayanır, derleyici zorlaması değildir.

> Kural olarak: yeni yazılan, gerçekten immutable olması gereken ve pattern matching'e konu olabilecek veri tipleri için **record**; builder API'sine ihtiyaç duyan karmaşık nesneler, mutable durum gerektiren sınıflar ya da zaten Lombok'a bağımlı, Bean konvansiyonuna ihtiyaç duyan eski bir kod tabanına eklenen sınıflar için **Lombok** tercih et. İkisi birbirini dışlamaz — aynı projede (tıpkı burada olduğu gibi) bir arada yaşayabilirler.

Ek: Record Patterns (Java 21)

Enum konusunda modern `switch` söz dizimini zaten kullanmıştık; Java 21 bu söz dizimini record'lar için bir adım öteye taşıyor: **record pattern**'ler, bir record'u tek satırda hem tip kontrolüne tabi tutup hem de bileşenlerine ayrıştırmanı (deconstruct) sağlıyor.

En basit hâliyle, `instanceof` ile:

```
Object obj = new Point(3, 4);

if (obj instanceof Point(int x, int y)) {
    System.out.println("x=" + x + ", y=" + y); // x ve y burada doğrudan kullanılabilir
}
```

Klasik yaklaşımda önce `instanceof Point` ile tip kontrolü yapıp sonra `((Point) obj).x()` ile bileşenlere erişmen gerekirdi — record pattern, bu iki adımı tek satırda birleştiriyor ve `x` / `y` değişkenlerini doğrudan sonucu kullanılabilir yerel değişkenler olarak açığa çıkarıyor.

Asıl gücünü, bir `sealed interface`'in tüm alt tiplerini record olarak modelleyip `switch` ile birleştirdiğinde gösteriyor:

```
sealed interface Shape permits Circle, Rectangle, Square {
}

record Circle(double radius) implements Shape {
}

record Rectangle(double width, double height) implements Shape {
}

record Square(double side) implements Shape {
}

class SealedShapeExample {

    static double area(Shape shape) {
        return switch (shape) {
            case Circle(double r) -> Math.PI * r * r;
            case Rectangle(double w, double h) -> w * h;
            case Square(double s) -> s * s;
        };
    }

    public static void main(String[] args) {
        System.out.printf("%.2f%n", area(new Circle(2)));      // 12.57
        System.out.println(area(new Rectangle(3, 4)));          // 12.0
        System.out.println(area(new Square(5)));                // 25.0
    }
}
```

> Yukarıdaki switch ifadesinde `default` dalı **yok** — çünkü `Shape` `sealed` olduğu ve yalnızca üç `permits` edilen alt tipi olduğu için derleyici, olası tüm durumların karşılandığını doğrulayabiliyor (tıpkı enum'daki exhaustive switch gibi, bkz. Enum konusundaki "switch ile Kullanım" bölümü). `Shape`'e yeni bir alt tip eklenip bu switch güncellenmezse, derleme **hata verir** — çalışma zamanında değil, derleme zamanında yakalanan bir eksiklik.

Record pattern'ler **iç içe** de kullanılabilir — "İç İçe Record'lar" bölümündeki `Employee` / `Address` örneğini, bileşenlere tek satırda erişecek şekilde yeniden yazalım:

```
record Address(String city, String zip) {
}

record Employee(String name, Address address) {
}

class NestedPatternExample {

    static String describe(Object obj) {
        if (obj instanceof Employee(String name, Address(String city, String zip))) {
            return name + " - " + city + " (" + zip + ")";
        }
        return "unknown";
    }

    public static void main(String[] args) {
        Employee e = new Employee("Ada", new Address("Istanbul", "34000"));
        System.out.println(describe(e)); // Ada - Istanbul (34000)
    }
}
```

Beklenen Çıktı

```
Ada - Istanbul (34000)
```

Son olarak, bir pattern'e ek bir koşul eklemek istediğinde **guarded pattern** (`when` anahtar kelimesi) kullanılır:

```
static String describe(Shape shape) {
    return switch (shape) {
        case Circle(var r) when r > 100 -> "Devasa bir daire";
        case Circle(var r) -> "Yarıçapı " + r + " olan bir daire";
        case Rectangle(var w, var h) -> "Dikdörtgen (" + w + "x" + h + ")";
        case Square(var s) -> "Kenarı " + s + " olan bir kare";
    };
}
```

> `case Circle(var r) when r > 100 -> ...` satırının, genel `case Circle(var r) -> ...` satırından **önce** gelmesi gerekiyor — switch dalları yukarıdan aşağıya sırayla denenir, tıpkı klasik `if/else if` zincirinde olduğu gibi. Sıra ters olsaydı, genel `Circle` dalı her zaman önce eşleşir ve guarded (`when`'li) dal hiçbir zaman çalışmazdı.

Java Temelleri kapsamındaki Record konusu tamamlandı — 17 ana bölüm + 2 ek bölümün tümü yazıldı. Sıradaki adaylar: bu konunun İngilizce çevirisi (şu an taslak) ya da yeni bir Java Temelleri konusu.