# Date & Time API

Java'da java.time paketi; LocalDate/LocalTime/LocalDateTime, Instant, ZonedDateTime /OffsetDateTime, Duration/Period/ChronoUnit, formatlama/parse, saat dilimleri ve legacy Date/Calendar'dan geçiş.

**Orta** 55 dk

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

---

# Date & Time API

Bu derste, Java'nın tarih/saat işleme mekanizmasını — `java.time` paketini — baştan sona ele alacağız: `LocalDate`'ten `Instant`'a, saat dilimlerinin (time zone) nasıl işlediğinden Legacy `Date` / `Calendar` API'siyle nasıl birlikte çalışılacağına kadar. Tarih/saat, ilk bakışta basit görünen ama saat dilimleri ve yaz saati uygulaması (DST) gibi ayrıntılarla hızla karmaşıklaşan bir alan — bu yüzden bu dersi, "hangi sınıfı ne zaman kullanmalıyım?" sorusuna net bir cevapla bitireceğiz.

## Konu Nedir?

`java.time` paketi, bir tarihi, bir saati, ikisinin birleşimini ya da zaman çizelgesi üzerindeki bir anı temsil eden, hepsi **değişmez (immutable)** bir sınıf ailesidir. En temel dördü: yalnızca tarih tutan `LocalDate`, yalnızca saat tutan `LocalTime`, ikisini birleştiren `LocalDateTime`, ve saat dilimi belirsizliği olmadan evrensel bir zaman noktasını temsil eden `Instant`:

```
LocalDate today = LocalDate.now();       // 2026-08-10
LocalTime now = LocalTime.now();         // 14:32:07
LocalDateTime dateTime = LocalDateTime.now(); // 2026-08-10T14:32:07
Instant instant = Instant.now();         // 2026-08-10T11:32:07Z (UTC)
```

"Local" öneki kafa karıştırabilir — burada "yerel" olan, bir kullanıcının saatinin bulunduğu saat dilimi değil, sınıfın **hiçbir saat dilimi bilgisi taşımaması**dır. Bu ayrımın neden önemli olduğunu "ZonedDateTime ve Time Zone Kavramı" bölümünde detaylı göreceğiz.

## Neden Var?

Gerçek hayattan bir örnek: İstanbul'da yaşayan bir kullanıcının bir toplantıyı 15:00'te kurduğunu, New York'taki bir katılımcının ise bunu **kendi** saatinde (07:00 EDT) görmesi

gerektiğini düşün. Bunu doğru yapmak için üç ayrı bilgiye ihtiyacın var: mutlak bir zaman noktası (herkesin üzerinde anlaştığı "gerçek" an), bir saat dilimi kuralı (bu anın belirli bir bölgede hangi yerel saate denk geldiği) ve bu ikisini kullanıcıya okunabilir şekilde göstermenin bir yolu. `java.time`'ın farklı sınıflara ayrılmış olması tam olarak bu üç ihtiyacı ayrı ayrı, birbirine karıştırmadan çözebilmek için — bunu ilk mini projede uçtan uca göreceğiz.

## Tarihçe

Java'nın ilk tarih/saat API'si (`java.util.Date`, sonra `Calendar`) 1996'dan beri vardı ama ciddi tasarım sorunları taşıyordu: `Date` hem değişebilirdi (mutable) hem de ay değerleri 0'dan başlıyordu (Ocak = 0), `Calendar` API'si aşırı karmaşıktı, ve `SimpleDateFormat` thread-safe değildi — bunların hepsine "Legacy API'den java.time'a Geçiş" bölümünde döneceğiz. Bu sorunlar o kadar yaygın şikayet konusuydu ki, Stephen Colebourne popüler **Joda-Time** kütüphanesini yazdı; Joda-Time o kadar başarılı oldu ki doğrudan Java'nın resmi standardına ilham verdi. Oracle, Colebourne'u da işin içine katarak JSR-310'u başlattı ve sonucu Java 8'de (2014) `java.time` paketi olarak yayınladı — değişmez (immutable), thread-safe ve net bir sorumluluk ayrımına sahip, sıfırdan tasarlanmış bir API.

## LocalDate

`LocalDate`, yalnızca bir takvim tarihini (yıl, ay, gün) tutar — saat ya da saat dilimi bilgisi içermez:

```
import java.time.LocalDate;

class LocalDateExample {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now(); // varies depending on when you run this
        System.out.println("Today: " + today);

        LocalDate fixedDate = LocalDate.of(2026, 3, 15);
        System.out.println("Fixed date: " + fixedDate);

        LocalDate nextWeek = fixedDate.plusDays(7); // returns a NEW LocalDate, doesn't
mutate fixedDate
        System.out.println("A week later: " + nextWeek);
        System.out.println("Original unchanged: " + fixedDate);

        System.out.println("Day of week: " + fixedDate.getDayOfWeek());
        System.out.println("Is 2026 a leap year? " + fixedDate.isLeapYear());
    }
}
```

`LocalDate.of(2026, 3, 15)` ile doğrudan bir tarih oluşturabilir, `plusDays(...)` gibi metotlarla yeni tarihler türetebilirsin — her `plus`/`minus` çağrısı, tüm `java.time` sınıflarında olduğu gibi

**orijinali değiştirmez**, yeni bir `LocalDate` nesnesi döndürür. `getDayOfWeek()` ve `isLeapYear()` gibi metotlar, takvim hesaplarını (artık yıl kuralları, hafta günleri) senin yerine doğru şekilde yapıyor — bunları elle hesaplamaya çalışmak, bilinen bir hata kaynağıdır.

## LocalTime

`LocalTime`, `LocalDate`'in saat karşılığı — yalnızca gün içindeki bir saati (saat, dakika, saniye, nanosaniye) tutar, hiçbir tarih ya da saat dilimi bilgisi içermez:

```
import java.time.LocalTime;

class LocalTimeExample {
    public static void main(String[] args) {
        LocalTime now = LocalTime.now(); // varies depending on when you run this
        System.out.println("Now: " + now);

        LocalTime openingTime = LocalTime.of(9, 0); // seconds and nanos default to 0
        System.out.println("Opening time: " + openingTime);

        LocalTime closingTime = openingTime.plusHours(8);
        System.out.println("Closing time: " + closingTime);

        System.out.println("Closing hour: " + closingTime.getHour());
        System.out.println("Is opening time before noon? " + openingTime.isBefore(LocalTime.
NOON));
    }
}
```

`LocalTime.of(14, 30)` gibi bir çağrı saniye ve nanosaniyeyi örtük olarak sıfır kabul eder. `LocalTime`'ın en tipik kullanım alanı, bir tarihten bağımsız, tekrar eden bir saat ifade etmek — örneğin "her gün saat 09:00'da açılış" gibi bir iş kuralı, belirli bir takvim gününe bağlı olmadığı için `LocalDateTime` yerine `LocalTime` ile daha doğru modellenir.

## LocalDateTime

`LocalDateTime`, `LocalDate` ile `LocalTime`'ı birleştirir — pratikte en sık kullanılan `java.time` sınıfıdır, çünkü çoğu uygulama "ne zaman" sorusunu hem tarih hem saat olarak düşünür:

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

class LocalDateTimeExample {
    public static void main(String[] args) {
        LocalDate date = LocalDate.of(2026, 3, 15);
        LocalTime time = LocalTime.of(14, 30);

        LocalDateTime combinedA = date.atTime(time);
        LocalDateTime combinedB = LocalDateTime.of(date, time);
        System.out.println("Same result either way? " + combinedA.equals(combinedB));

        System.out.println("Combined: " + combinedA);
        System.out.println("Back to date: " + combinedA.toLocalDate());
        System.out.println("Back to time: " + combinedA.toLocalTime());
    }
}
```

**Beklenen Çıktı**

```
Same result either way? true
Combined: 2026-03-15T14:30
Back to date: 2026-03-15
Back to time: 14:30
```

`LocalDate.atTime(LocalTime)` ve `LocalDateTime.of(date, time)` aynı sonucu iki farklı yoldan üretiyor — hangisini kullanacağın elindeki parçalara bağlı. `toLocalDate()` ve `toLocalTime()` ise tersini yapıyor: birleşik bir `LocalDateTime`'dan parçaları geri ayıklıyor.

> `LocalDateTime`'ın hâlâ **hiçbir saat dilimi bilgisi** taşımadığını unutma — "15:00" yazan bir `LocalDateTime`, İstanbul'da mı New York'ta mı olduğunu bilmiyor. Bunu bir saat dilimiyle ilişkilendirmen gerektiğinde "ZonedDateTime ve Time Zone Kavramı" bölümüne geç.

## Instant

`Instant`, UTC zaman çizelgesi üzerinde (Ocak 1970 epoch'undan bu yana geçen saniye ve nanosaniye olarak) tek bir noktayı temsil eder — hiçbir "takvim günü" ya da "saat dilimi" kavramı içermez, yalnızca **evrensel, belirsizliksiz bir an**dır:

```
import java.time.Instant;

class InstantExample {
    public static void main(String[] args) {
        Instant now = Instant.now(); // varies depending on when you run this
        System.out.println("Now: " + now);

        Instant epoch = Instant.EPOCH;
        System.out.println("Epoch: " + epoch);

        Instant fixed = Instant.ofEpochSecond(1_700_000_000L);
        System.out.println("Fixed instant: " + fixed);

        Instant later = fixed.plusSeconds(3600); // one hour later
        System.out.println("An hour later: " + later);
    }
}
```

`Instant`, makineler arası iletişim için idealdir: bir log kaydının, bir veritabanı zaman damgasının ya da bir olayın "gerçekte ne zaman olduğu" sorusuna, hangi saat diliminde okunduğundan tamamen bağımsız, tek bir doğru cevap verir. Bu yüzden "Gerçek Dünya Örnekleri: Spring Boot'ta java.time" bölümünde de göreceğimiz gibi, veritabanında bir zaman damgası saklarken genelde `Instant` (ya da sabit ofsetli `OffsetDateTime`) tercih edilir — kullanıcıya gösterirken bu evrensel ana, kullanıcının saat dilimi uygulanır.

## ZonedDateTime ve Time Zone Kavramı

`ZonedDateTime`, bir `LocalDateTime`'ı bir `ZoneId` (örneğin `"Europe/Istanbul"`) ile birleştirir — böylece hem "yerel saat neydi" hem de "bu, hangi bölgenin kurallarına göre yorumlanmalı" bilgisini bir arada taşır:

```
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;

class ZonedDateTimeExample {
    public static void main(String[] args) {
        LocalDateTime local = LocalDateTime.of(2026, 7, 15, 15, 0);
        ZonedDateTime istanbul = local.atZone(ZoneId.of("Europe/Istanbul"));
        System.out.println("Istanbul: " + istanbul);

        // withZoneSameInstant -- the INSTANT stays the same, the local time changes
        ZonedDateTime newYorkSameInstant = istanbul.withZoneSameInstant(ZoneId.of("America
/New_York"));
        System.out.println("Same instant in New York: " + newYorkSameInstant);

        // withZoneSameLocal -- the LOCAL time stays "15:00", the instant changes
        ZonedDateTime newYorkSameLocal = istanbul.withZoneSameLocal(ZoneId.of("America
/New_York"));
        System.out.println("Same local time relabeled to New York (different instant): " +
newYorkSameLocal);

        System.out.println("Istanbul and 'same instant' New York represent the same
instant? "
            + istanbul.toInstant().equals(newYorkSameInstant.toInstant()));
        System.out.println("Istanbul and 'same local' New York represent the same instant? "
            + istanbul.toInstant().equals(newYorkSameLocal.toInstant()));
    }
}
```

**Beklenen Çıktı**

```
Istanbul: 2026-07-15T15:00+03:00[Europe/Istanbul]
Same instant in New York: 2026-07-15T08:00-04:00[America/New_York]
Same local time relabeled to New York (different instant): 2026-07-15T15:00-04:00[America
/New_York]
Istanbul and 'same instant' New York represent the same instant? true
Istanbul and 'same local' New York represent the same instant? false
```

`ZoneId.of("Europe/Istanbul")`, sabit bir sayı değil, bir **bölgenin tüm tarihsel ve gelecekteki kurallarını** (yaz saati geçişleri dahil) temsil eder — aynı `ZonedDateTime` nesnesi, hangi tarihte olduğuna bağlı olarak farklı bir UTC ofsetine karşılık gelebilir. `withZoneSameInstant(...)`, aynı evrensel anı **başka bir bölgenin yerel saatiyle** gösterir (saat değişir, an aynı kalır); `withZoneSameLocal(...)` ise tam tersini yapar (yerel saat aynı kalır, an değişir) — bu ikisini karıştırmak yaygın bir hata kaynağıdır.

> `ZonedDateTime` üzerinde `equals()` çağırmak, hem anı hem **saat dilimini** birlikte karşılaştırır — aynı evrensel anı temsil eden ama farklı bölgelerde ifade edilmiş iki `ZonedDateTime` `equals()`'a göre eşit **değildir**. Yalnızca temsil ettikleri anın aynı olup olmadığını kontrol etmek istiyorsan `isEqual(...)` kullan.

## OffsetDateTime

`OffsetDateTime`, bir `LocalDateTime`'ı `ZoneId` yerine sabit bir `ZoneOffset` (örneğin `+03:00`) ile birleştirir. Aradaki fark kritik: bir `ZoneOffset` yalnızca sabit bir sayıdır, hiçbir yaz saati geçiş kuralı taşımaz — oysa bir `ZoneId`, "Europe/Istanbul" gibi bir **bölgenin** zaman içinde değişebilen kurallarını temsil eder:

```
import java.time.LocalDateTime;
import java.time.OffsetDateTime;
import java.time.ZoneOffset;

class OffsetDateTimeExample {
    public static void main(String[] args) {
        LocalDateTime local = LocalDateTime.of(2026, 3, 15, 14, 30);

        OffsetDateTime withOffset = local.atOffset(ZoneOffset.of("+03:00"));
        System.out.println("With fixed offset: " + withOffset);

        // A ZoneOffset is just a fixed number -- it carries no DST transition rules,
        // unlike a ZoneId region such as "Europe/Istanbul".
        System.out.println("Offset: " + withOffset.getOffset());
    }
}
```

**Beklenen Çıktı**

```
With fixed offset: 2026-03-15T14:30+03:00
Offset: +03:00
```

Bu ayrım, ne zaman hangisini kullanacağını belirler: bir kullanıcıya "İstanbul saatiyle" göstermek istiyorsan `ZonedDateTime` + `ZoneId` doğru araçtır (yaz saati kurallarını otomatik uygular); bir API sözleşmesinde ya da veritabanı sütununda **sabit bir ofsetli** zaman damgası (ISO-8601'in `OffsetDateTime` biçimi) saklamak istiyorsan `OffsetDateTime` daha öngörülebilirdir — bölge kurallarının gelecekte değişmesi (bir ülkenin yaz saati uygulamasını kaldırması gibi) geçmiş kayıtları etkilemez.

## Duration ve Period

İki zaman noktası arasındaki farkı ifade etmenin iki yolu var, ve hangisini seçtiğin neyi ölçtüğüne bağlı: **`Duration`**, saat/dakika/saniye gibi **zaman bazlı** bir miktarı temsil eder (`Instant`, `LocalTime`, `LocalDateTime` arasında); **`Period`** ise yıl/ay/gün gibi **takvim bazlı** bir miktarı temsil eder (yalnızca `LocalDate` arasında):

```
import java.time.Duration;
import java.time.Instant;
import java.time.LocalDate;
import java.time.Period;

class DurationAndPeriodExample {
    public static void main(String[] args) {
        Instant start = Instant.parse("2026-01-01T09:00:00Z");
        Instant end = Instant.parse("2026-01-01T17:30:00Z");

        Duration workDay = Duration.between(start, end);
        System.out.println("Duration: " + workDay);
        System.out.println("Hours: " + workDay.toHours());
        System.out.println("Minutes: " + workDay.toMinutes());

        LocalDate projectStart = LocalDate.of(2024, 1, 10);
        LocalDate projectEnd = LocalDate.of(2026, 4, 20);

        Period projectLength = Period.between(projectStart, projectEnd);
        System.out.println("Period: " + projectLength.getYears() + " years, "
            + projectLength.getMonths() + " months, " + projectLength.getDays() + " days");
    }
}
```

**Beklenen Çıktı**

```
Duration: PT8H30M
Hours: 8
Minutes: 510
Period: 2 years, 3 months, 10 days
```

`Duration.between(start, end)`, iki `Instant` arasındaki farkı saniye/nanosaniye hassasiyetinde verir — "3 ay" gibi bir kavram `Duration`'a hiç uymaz, çünkü ayların uzunluğu değişkendir. `Period.between(start, end)` ise tam tersini yapar: "2 yıl, 3 ay, 10 gün" gibi bir takvim farkı üretir, ama saat/dakika hassasiyeti sunmaz. İkisini birbirinin yerine kullanmaya çalışmak (örneğin bir `Period`'u saniyeye çevirmek) anlamsız sonuçlar verir — sonraki bölümde göreceğimiz `ChronoUnit`, bu iki dünya arasında daha esnek bir köprü kuruyor.

## ChronoUnit ile Zaman Farkı Hesaplama

`ChronoUnit`, `TemporalUnit` interface'ini implement eden bir enum'dur (Enum dersinin "Arayüz (Interface) İmplementasyonu" bölümünde gördüğümüz deseni hatırla) — `DAYS`, `HOURS`, `MONTHS` gibi sabitler sunar ve `Duration` / `Period` gibi bir nesne oluşturmadan, doğrudan **tek bir sayı** olarak fark hesaplamana izin verir:

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

class ChronoUnitExample {
    public static void main(String[] args) {
        LocalDate start = LocalDate.of(2024, 1, 10);
        LocalDate end = LocalDate.of(2026, 4, 20);

        long totalDays = ChronoUnit.DAYS.between(start, end);
        long totalMonths = ChronoUnit.MONTHS.between(start, end);
        long totalYears = ChronoUnit.YEARS.between(start, end);

        System.out.println("Total days: " + totalDays);
        System.out.println("Total months: " + totalMonths);
        System.out.println("Total years: " + totalYears);
    }
}
```

**Beklenen Çıktı**

```
Total days: 831
Total months: 27
Total years: 2
```

`ChronoUnit.DAYS.between(start, end)`, `Period.between(...).getDays()`'ten farklı olarak, **toplam** gün farkını (örneğin 400 gün) tek bir `long` olarak döndürür — `Period` ise bunu "1 yıl, 1 ay, 5 gün" gibi parçalara ayırır. Hangi tür soruya cevap vermek istediğine göre seç: "aralarında kaç gün var?" sorusunun cevabı `ChronoUnit`, "aralarında ne kadar zaman var, insan diliyle?" sorusunun cevabı `Period` / `Duration`'dır. `ChronoUnit`'in bir başka avantajı da genelliği: `LocalDate`, `LocalDateTime`, `Instant` gibi birçok farklı tip arasında aynı `between(...)` çağrısıyla çalışır.

## DateTimeFormatter: Formatlama ve Parse Etme

`DateTimeFormatter`, bir tarih/saat nesnesini metne (`format(...)`) ya da metni bir tarih/saat nesnesine (`parse(...)`) çevirmenin standart yoludur — hem hazır ISO-8601 biçimleri hem de `ofPattern(...)` ile özel biçimler tanımlayabilirsin:

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;

class FormattingAndParsingExample {
    public static void main(String[] args) {
        LocalDate date = LocalDate.of(2026, 3, 15);

        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
        String formatted = date.format(formatter);
        System.out.println("Formatted: " + formatted);

        LocalDate parsedBack = LocalDate.parse(formatted, formatter);
        System.out.println("Parsed back: " + parsedBack);
        System.out.println("Round-trip matches original? " + parsedBack.equals(date));

        try {
            LocalDate.parse("2026-13-45"); // invalid month and day
        } catch (DateTimeParseException e) {
            System.out.println("Caught expected parse failure: " + e.getMessage());
        }
    }
}
```

`DateTimeFormatter.ofPattern("dd/MM/yyyy")`, bir `LocalDate`'i `"15/03/2026"` gibi insana uygun bir biçimde yazdırır; aynı formatter, ters yönde `LocalDate.parse(text, formatter)` ile metni geri bir `LocalDate`'e çevirir. Metin, beklenen kalıba uymuyorsa (örneğin `"2026-13-45"` gibi geçersiz bir tarih) `DateTimeParseException` fırlatılır — bu, kullanıcı girdisini işlerken **mutlaka** `try/catch` ile ele alman gereken bir istisnadır.

## Tarih Hesaplamaları

Her `java.time` sınıfı, `plusDays()`, `minusMonths()`, `plusYears()` gibi bir dizi `plus`/`minus` metodu sunar — hepsi değişmezliği koruyarak (bkz. "LocalDate" bölümü) **yeni** bir nesne döndürür, orijinali asla değiştirmez:

```
import java.time.LocalDate;

class DateCalculationsExample {
    public static void main(String[] args) {
        LocalDate jan31 = LocalDate.of(2026, 1, 31);

        LocalDate result = jan31.plusMonths(1); // January 31 + 1 month
        System.out.println("Jan 31 + 1 month: " + result); // clamped -- 2026 is not a leap
year

        LocalDate chained = jan31.plusYears(1).minusDays(5);
        System.out.println("Chained calculation: " + chained);
        System.out.println("Original still unchanged: " + jan31);
    }
}
```

**Beklenen Çıktı**

```
Jan 31 + 1 month: 2026-02-28
Chained calculation: 2027-01-26
Original still unchanged: 2026-01-31
```

`plusMonths(1)` gibi bir çağrı, ay taşmalarını (örneğin 31 Ocak + 1 ay) senin yerine akıllıca çözer — sonucu "31 Şubat" gibi var olmayan bir tarihe değil, o ayın son gününe (28 ya da 29 Şubat) sabitler. Zincirleme çağrılar ( `date.plusYears(1).minusDays(5)` ) her adımda yeni bir nesne ürettiği için gayet güvenlidir — hiçbir ara adım, bir öncekini etkilemez.

## TemporalAdjusters

Bazı tarih hesaplamaları basit bir `plus` / `minus` ile ifade edilemeyecek kadar kural-bazlıdır — "bir sonraki Pazartesi" ya da "bu ayın son günü" gibi. `TemporalAdjusters` sınıfı, `with(...)` metoduna verebileceğin hazır kurallar (adjuster) sunar:

```
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;

class TemporalAdjustersExample {
    public static void main(String[] args) {
        LocalDate date = LocalDate.of(2026, 3, 15);
        System.out.println("Starting date: " + date + " (" + date.getDayOfWeek() + ")");

        LocalDate nextMonday = date.with(TemporalAdjusters.next(DayOfWeek.MONDAY));
        System.out.println("Next Monday: " + nextMonday);

        LocalDate lastDayOfMonth = date.with(TemporalAdjusters.lastDayOfMonth());
        System.out.println("Last day of the month: " + lastDayOfMonth);

        LocalDate firstDayOfYear = date.with(TemporalAdjusters.firstDayOfYear());
        System.out.println("First day of the year: " + firstDayOfYear);
    }
}
```

**Beklenen Çıktı**

```
Starting date: 2026-03-15 (SUNDAY)
Next Monday: 2026-03-16
Last day of the month: 2026-03-31
First day of the year: 2026-01-01
```

`date.with(TemporalAdjusters.next(DayOfWeek.MONDAY))` , "bugünden sonraki ilk Pazartesi"yi tek satırda hesaplıyor — bunu elle yazmaya çalışsan haftanın gün sayısı, ay sonu gibi birçok kenar durumu düşünmen gerekirdi. `lastDayOfMonth()` , `firstDayOfYear()` gibi hazır adjuster'lar da aynı felsefeyi izler: sık karşılaşılan takvim kurallarını, hataya açık elle hesaplama yerine, test edilmiş tek bir metot çağrısına indirger.

## Tarihleri Karşılaştırma

`LocalDate`, `LocalDateTime` ve `Instant` gibi sınıflar, iki değeri karşılaştırmak için `isBefore()`, `isAfter()` ve `compareTo()` sunar — `equals()` de çalışır, ama "ZonedDateTime ve Time Zone Kavramı" bölümünde gördüğümüz gibi bazı tiplerde ekstra bilgiyi (saat dilimi gibi) de karşılaştırmaya dahil edebilir:

```
import java.time.LocalDate;

class ComparingDatesExample {
    public static void main(String[] args) {
        LocalDate first = LocalDate.of(2026, 3, 15);
        LocalDate second = LocalDate.of(2026, 6, 1);

        System.out.println("first isBefore second? " + first.isBefore(second));
        System.out.println("first isAfter second? " + first.isAfter(second));
        System.out.println("compareTo: " + first.compareTo(second)); // negative -- first
comes before second

        LocalDate sameAsFirst = LocalDate.of(2026, 3, 15);
        System.out.println("equals (value comparison): " + first.equals(sameAsFirst));
    }
}
```

`isBefore()` / `isAfter()`, okunabilirlik açısından `compareTo() < 0` yazmaktan çok daha nettir — kod, "bu tarih şu tarihten önce mi?" sorusunu neredeyse İngilizce gibi okunan bir çağrıyla ifade eder. `LocalDate` ve `LocalDateTime` için `equals()` zaten yalnızca değeri karşılaştırır (Record dersindeki otomatik `equals()` gibi, alan alan karşılaştırma) — asıl dikkat etmen gereken istisna, bir önceki bölümde gördüğümüz `ZonedDateTime`'dır.

## Legacy API'den java.time'a Geçiş

Eski kod tabanlarında hâlâ `java.util.Date`, `Calendar` ve `SimpleDateFormat` ile karşılaşabilirsin — üçünün de "Tarihçe" bölümünde değindiğimiz ciddi sorunları var, ama tamamen kaçınamayacağın (üçüncü parti bir kütüphaneden gelen) durumlar için dönüştürme köprüleri mevcut:

```
import java.time.Instant;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.Date;

class LegacyInteropExample {
    public static void main(String[] args) {
        Date legacyDate = new Date(1_700_000_000_000L); // old API, milliseconds since epoch

        Instant instant = legacyDate.toInstant(); // lossless -- Date is just an epoch
timestamp underneath
        System.out.println("Converted to Instant: " + instant);

        ZonedDateTime inIstanbul = instant.atZone(ZoneId.of("Europe/Istanbul"));
        System.out.println("Viewed in Istanbul: " + inIstanbul);

        Date backToLegacy = Date.from(instant); // bridging back, for an old API that still
needs a Date
        System.out.println("Round-trip matches original? " + backToLegacy.equals
(legacyDate));
    }
}
```

**Beklenen Çıktı**

```
Converted to Instant: 2023-11-14T22:13:20Z
Viewed in Istanbul: 2023-11-15T01:13:20+03:00[Europe/Istanbul]
Round-trip matches original? true
```

`Date.toInstant()`, eski bir `Date`'i modern bir `Instant`'a çevirir — `Date`'in kendisi zaten içeride bir epoch zaman damgasından başka bir şey tutmadığı için bu dönüşüm kayıpsızdır. `Instant.atZone(zoneId)` ise tersi yönde köprü kurar. En kritik uyarı `SimpleDateFormat` için: bu sınıf **thread-safe değildir** — birden fazla thread'in (Threads dersinin "Race Condition" bölümünü hatırla) aynı `SimpleDateFormat` nesnesini paylaşması bozuk sonuçlara yol açabilir; `DateTimeFormatter` ise değişmezdir ve thread'ler arasında güvenle paylaşılabilir.

> Yeni kod asla `Date`, `Calendar` ya da `SimpleDateFormat` ile başlamamalı — bunlar yalnızca eski bir API'yle (üçüncü parti bir kütüphane, eski bir veritabanı sürücüsü gibi) sınırda köprü kurmak için kullanılmalı. "Yaygın Hatalar" bölümünde bu kuralı tekrar vurgulayacağız.

## Time Zone'lar ve Daylight Saving Time

Saat dilimleri, bu dersin en çok göz ardı edilen ama en çok hataya yol açan konusu. `UTC` (Koordineli Evrensel Zaman), tüm saat dilimlerinin referans aldığı sabit noktadır; `GMT` pratikte UTC ile aynı ofseti paylaşır ama tarihsel olarak farklı bir tanıma dayanır. `"Europe/Istanbul"`, `"America/New_York"` gibi bölge kimlikleri ise sabit bir ofis değil, zaman içinde **değişebilen kurallar** taşır — en önemlisi de yaz saati uygulaması (DST):

```
import java.time.ZoneId;
import java.time.ZonedDateTime;

class TimeZoneAndDstExample {
    public static void main(String[] args) {
        // Istanbul stopped observing DST in 2016 -- fixed at UTC+3 year round.
        ZonedDateTime istanbulWinter = ZonedDateTime.of(2026, 1, 15, 12, 0, 0, 0, ZoneId.of
("Europe/Istanbul"));
        ZonedDateTime istanbulSummer = ZonedDateTime.of(2026, 7, 15, 12, 0, 0, 0, ZoneId.of
("Europe/Istanbul"));
        System.out.println("Istanbul winter offset: " + istanbulWinter.getOffset());
        System.out.println("Istanbul summer offset: " + istanbulSummer.getOffset() + "
(same -- no DST anymore)");

        // New York still observes DST -- the offset differs between winter and summer.
        ZonedDateTime newYorkWinter = ZonedDateTime.of(2026, 1, 15, 12, 0, 0, 0, ZoneId.of
("America/New_York"));
        ZonedDateTime newYorkSummer = ZonedDateTime.of(2026, 7, 15, 12, 0, 0, 0, ZoneId.of
("America/New_York"));
        System.out.println("New York winter offset: " + newYorkWinter.getOffset());
        System.out.println("New York summer offset: " + newYorkSummer.getOffset() + "
(different!)");
    }
}
```

**Beklenen Çıktı**

```
Istanbul winter offset: +03:00
Istanbul summer offset: +03:00 (same -- no DST anymore)
New York winter offset: -05:00
New York summer offset: -04:00 (different!)
```

DST geçişleri iki tuhaf duruma yol açar: saatlerin ileri alındığı günde bazı yerel saatler **hiç var olmaz** (örneğin 02:30, doğrudan 03:30'a atlanır), saatlerin geri alındığı günde ise bazı yerel saatler **iki kez** yaşanır. `ZonedDateTime`, bu belirsizlikleri senin için otomatik bir kuralla çözer — ama bu kuralın var olduğunu bilmemek, "neden bu tarih 25 saatlik bir gün gösteriyor?" gibi şaşırtıcı hatalara yol açar. Türkiye özelinde ilginç bir örnek: `"Europe/Istanbul"` bölgesi 2016'dan beri yaz saati uygulamasını kaldırdı ve yıl boyu sabit UTC+3'te kaldı — bu da `ZoneId` kurallarının gerçekten **zamanla değişebildiğinin** somut bir kanıtı.

## Gerçek Dünya Örnekleri: Spring Boot'ta java.time

`java.time`, modern bir Spring Boot uygulamasının neredeyse her katmanında karşına çıkar. JSON tarafında, Spring Boot'un varsayılan Jackson yapılandırması `jackson-datatype-jsr310` modülünü otomatik kaydeder — bu sayede bir `Instant` ya da `LocalDate` alanı, hiçbir ek anotasyon gerekmeden ISO-8601 metnine (`"2026-08-10T11:32:07Z"` gibi) serileşir. Veritabanı tarafında, Hibernate 5.2'den beri `java.time` tiplerini native olarak destekler — bir JPA `@Entity`'deki `Instant` alanı, PostgreSQL'in zaman dilimi farkındalıklı `timestamptz` sütununa doğrudan eşlenir:

```
import java.time.Instant;
import java.time.LocalDate;

class Event {
    private final String name;
    private final Instant createdAt; // absolute timestamp -- what Jackson/Hibernate map
automatically
    private final LocalDate eventDate; // calendar date -- no time zone needed for "which
day"

    Event(String name, Instant createdAt, LocalDate eventDate) {
        this.name = name;
        this.createdAt = createdAt;
        this.eventDate = eventDate;
    }

    @Override
    public String toString() {
        return name + " (created " + createdAt + ", scheduled for " + eventDate + ")";
    }
}

class EventExample {
    public static void main(String[] args) {
        Event event = new Event(
            "Java Meetup",
            Instant.parse("2026-03-01T10:15:30Z"),
            LocalDate.of(2026, 4, 12)
        );

        System.out.println(event);
        // In a real Spring Boot app, this exact class (with @Entity/@JsonFormat added)
        // would serialize createdAt as ISO-8601 JSON and map to a PostgreSQL timestamptz
        // column -- no special handling needed for the java.time types themselves.
    }
}
```

**Beklenen Çıktı**

```
Java Meetup (created 2026-03-01T10:15:30Z, scheduled for 2026-04-12)
```

`Event` sınıfı, `Instant createdAt` ve `LocalDate eventDate` alanlarını hiçbir framework anotasyonu olmadan tutuyor — asıl önemli olan nokta bu: framework entegrasyonu, sınıfının kendisinde değil, framework'ün bu tipleri **zaten tanıyor** olmasında. Aynı `Event` sınıfı, bir JPA `@Entity`'ye ya da bir Jackson DTO'suna dönüştürülse bile, alan tipleri (`Instant`, `LocalDate`) hiç değişmez — çünkü hem Hibernate hem Jackson bu tipleri kutudan çıktığı gibi anlıyor.

## Best Practices

- Veritabanında bir zaman damgası saklarken `Instant` (ya da sabit ofsetli `OffsetDateTime`) kullan, kullanıcıya gösterirken `ZonedDateTime`'a çevir — "Instant" bölümünde değindiğimiz gibi bu, saat dilimi belirsizliğini en erken noktada ortadan kaldırır.
- Bir kullanıcıya tarih/saat gösterirken her zaman `ZonedDateTime` (ya da en azından bilinen bir `ZoneId`) kullan — çıplak bir `LocalDateTime`, "hangi saat diliminde?" sorusuna asla cevap vermez (bkz. "LocalDateTime" bölümündeki uyarı).
- `Duration` / `Period` / `ChronoUnit` arasında seçim yaparken neyi ölçtüğünü sor: saat bazlı bir süre mi (`Duration`), takvim bazlı bir aralık mı (`Period`), yoksa tek bir sayı mı (`ChronoUnit`) istiyorsun (bkz. "Duration ve Period" ve "ChronoUnit ile Zaman Farkı Hesaplama").
- Yeni kodda asla `Date`, `Calendar` ya da `SimpleDateFormat` başlatma — yalnızca eski bir API'yle sınırda köprü kurarken kullan (bkz. "Legacy API'den java.time'a Geçiş").
- Kullanıcı girdisinden tarih parse ederken `DateTimeParseException`'ı her zaman ele al — geçersiz bir tarih string'i kaçınılmaz bir gerçektir, istisna değil (bkz. "DateTimeFormatter: Formatlama ve Parse Etme").

## Yaygın Hatalar

**1. `LocalDateTime`'ı bir saat dilimi biliyormuş gibi kullanmak.** `LocalDateTime` hiçbir zaman dilimi bilgisi taşımaz — kullanıcıya kesin bir an göstermen gerekiyorsa `ZonedDateTime` ya da `Instant` kullan (bkz. "LocalDateTime" bölümündeki uyarı).

**2. İki `ZonedDateTime`'ı `equals()` ile karşılaştırıp aynı anı temsil ettiklerini sanmak.** `equals()` saat dilimini de karşılaştırır — yalnızca anı karşılaştırmak için `isEqual(...)` gerekir (bkz. "ZonedDateTime ve Time Zone Kavramı" bölümündeki uyarı).

**3. Bir `SimpleDateFormat` nesnesini birden fazla thread arasında paylaşmak.** Bu sınıf thread-safe değildir ve bozuk sonuçlara yol açar; `DateTimeFormatter` bunun yerini güvenle alır (bkz. "Legacy API'den java.time'a Geçiş").

**4. Saat dilimi kurallarının (özellikle DST'nin) hiç değişmeyeceğini varsaymak.** "Europe/Istanbul" örneğinde gördüğümüz gibi, bir bölgenin yaz saati kuralları yıllar içinde tamamen değişebilir (bkz. "Time Zone'lar ve Daylight Saving Time").

**5. `Period`'u saat/dakika hassasiyetinde bir hesaplama için kullanmaya çalışmak.** `Period` yalnızca takvim bazlıdır (yıl/ay/gün); saat bazlı farklar için `Duration`, yalnızca ham bir sayı için `ChronoUnit` gerekir (bkz. "Duration ve Period").

## Özet, Cheat Sheet ve Terimler Sözlüğü

`java.time`, Java 8'den beri değişmez, thread-safe ve net sorumluluk ayrımına sahip tarih/saat API'sidir. Öne çıkan noktalar:

- `LocalDate` / `LocalTime` / `LocalDateTime`, hiçbir saat dilimi bilgisi taşımaz — "yerel" ifadesi bunu vurgular
- `Instant`, UTC zaman çizelgesinde belirsizliksiz tek bir andır — makineler arası iletişim ve veritabanı zaman damgaları için idealdir
- `ZonedDateTime`, bir `ZoneId` (DST kuralları dahil bölge bilgisi) taşır; `OffsetDateTime` ise yalnızca sabit bir `ZoneOffset` taşır
- `Duration` saat bazlı, `Period` takvim bazlı, `ChronoUnit` ise tek bir ham sayı olarak iki an arasındaki farkı ifade eder
- `DateTimeFormatter` hem formatlama hem parse etme için kullanılır, thread-safe'dir
- Her `plus` / `minus` çağrısı yeni bir nesne döndürür, orijinali değiştirmez
- `TemporalAdjusters`, "bir sonraki Pazartesi" gibi kural-bazlı hesaplamalar için hazır adjuster'lar sunar
- `Date` / `Calendar` / `SimpleDateFormat` yalnızca eski API'lerle köprü kurmak için kullanılmalı — `SimpleDateFormat` thread-safe değildir
- Veritabanında `Instant` / `OffsetDateTime` sakla, kullanıcıya `ZonedDateTime` göster

Hızlı referans:

```
// Temel sınıflar
LocalDate date = LocalDate.now();               // yalnızca tarih
LocalTime time = LocalTime.now();                // yalnızca saat
LocalDateTime dateTime = LocalDateTime.now();    // tarih + saat, saat dilimi yok
Instant instant = Instant.now();                 // UTC zaman çizelgesinde tek an

// Saat dilimi
ZonedDateTime zoned = ZonedDateTime.now(ZoneId.of("Europe/Istanbul"));
OffsetDateTime offset = OffsetDateTime.now(ZoneOffset.of("+03:00"));

// Fark hesaplama
Duration duration = Duration.between(instant1, instant2); // saat/dakika/saniye
Period period = Period.between(date1, date2);              // yıl/ay/gün
long days = ChronoUnit.DAYS.between(date1, date2);          // ham sayı

// Formatlama / parse
DateTimeFormatter fmt = DateTimeFormatter.ofPattern("dd/MM/yyyy");
String text = date.format(fmt);
LocalDate parsed = LocalDate.parse(text, fmt);

// Hesaplama ve TemporalAdjusters
LocalDate nextMonday = date.with(TemporalAdjusters.next(DayOfWeek.MONDAY));
LocalDate endOfMonth = date.with(TemporalAdjusters.lastDayOfMonth());

// Legacy köprüsü
Instant fromLegacy = legacyDate.toInstant();
Date toLegacy = Date.from(instant);
```

### Terimler Sözlüğü

`LocalDate` / `LocalTime` / `LocalDateTime` — Saat dilimi bilgisi taşımayan, sırasıyla tarih, saat ve ikisinin birleşimini temsil eden değişmez sınıflar.

`Instant` — UTC zaman çizelgesinde, saat dilimi belirsizliği olmadan tek bir anı temsil eden değişmez sınıf.

`ZonedDateTime` — Bir yerel tarih/saati, DST kuralları dahil bir bölgenin tüm kurallarını taşıyan bir `ZoneId` ile birleştiren sınıf.

`OffsetDateTime` — Bir yerel tarih/saati, DST kuralı taşımayan sabit bir `ZoneOffset` ile birleştiren sınıf.

`Duration` — İki zaman noktası arasındaki farkı saat/dakika/saniye cinsinden ifade eden, zaman bazlı bir miktar.

`Period` — İki tarih arasındaki farkı yıl/ay/gün cinsinden ifade eden, takvim bazlı bir miktar.

`ChronoUnit` — `TemporalUnit`'i implement eden enum; iki an arasındaki farkı tek bir ham sayı olarak vermek için kullanılır.

`DateTimeFormatter` — Tarih/saat nesnelerini metne çevirmek (formatlama) ya da metni tarih /saat nesnesine çevirmek (parse etme) için kullanılan, thread-safe sınıf.

`TemporalAdjusters` — "Bir sonraki Pazartesi", "ayın son günü" gibi kural-bazlı tarih hesaplamaları için hazır adjuster'lar sunan yardımcı sınıf.

**DST (Daylight Saving Time / Yaz Saati Uygulaması)** — Bazı bölgelerde saatlerin yılın belirli dönemlerinde ileri/geri alınması; `ZoneId` kurallarının zamanla değişebilmesinin başlıca nedeni.

## Ek: Mini Proje — Çoklu Saat Dilimli Toplantı Planlayıcı

Bu mini projede, "Neden Var?" bölümünde tarif ettiğimiz senaryoyu gerçekleştiriyoruz: bir toplantıyı tek bir mutlak an (`Instant`) olarak saklayıp, istediğin herhangi bir saat diliminde görüntüleyebilen bir planlayıcı:

```
import java.time.Instant;
import java.time.ZoneId;
import java.time.ZonedDateTime;

class MeetingScheduler {
    private final String title;
    private final Instant scheduledAt; // single source of truth -- an unambiguous instant

    MeetingScheduler(String title, Instant scheduledAt) {
        this.title = title;
        this.scheduledAt = scheduledAt;
    }

    ZonedDateTime viewIn(ZoneId zone) {
        return scheduledAt.atZone(zone);
    }

    String getTitle() {
        return title;
    }
}
```

```
import java.time.Instant;
import java.time.ZoneId;
import java.time.ZonedDateTime;

class MeetingSchedulerDemo {
    public static void main(String[] args) {
        MeetingScheduler meeting = new MeetingScheduler(
            "Sprint Planning",
            Instant.parse("2026-07-15T12:00:00Z")
        );

        ZonedDateTime inIstanbul = meeting.viewIn(ZoneId.of("Europe/Istanbul"));
        ZonedDateTime inNewYork = meeting.viewIn(ZoneId.of("America/New_York"));
        ZonedDateTime inTokyo = meeting.viewIn(ZoneId.of("Asia/Tokyo"));

        System.out.println(meeting.getTitle() + " local times:");
        System.out.println("  Istanbul: " + inIstanbul);
        System.out.println("  New York: " + inNewYork);
        System.out.println("  Tokyo: " + inTokyo);

        // All three represent the exact same instant, just expressed differently.
        boolean sameInstant = inIstanbul.toInstant().equals(inNewYork.toInstant())
            && inNewYork.toInstant().equals(inTokyo.toInstant());
        System.out.println("Same instant everywhere? " + sameInstant);
    }
}
```

**Beklenen Çıktı**

```
Sprint Planning local times:
  Istanbul: 2026-07-15T15:00+03:00[Europe/Istanbul]
  New York: 2026-07-15T08:00-04:00[America/New_York]
  Tokyo: 2026-07-15T21:00+09:00[Asia/Tokyo]
Same instant everywhere? true
```

`MeetingScheduler`, toplantı anını yalnızca bir `Instant` olarak tutuyor — "Instant" bölümünde vurguladığımız gibi, bu tek doğruluk kaynağı hiçbir saat dilimi belirsizliği taşımıyor. `viewIn(ZoneId)` metodu, bu `Instant`'ı istenen bölgenin `ZonedDateTime`'ına çeviriyor; `MeetingSchedulerDemo`, aynı toplantının İstanbul, New York ve Tokyo'da **aynı anda** hangi yerel saate denk geldiğini gösteriyor — üçü de farklı saatler yazdırıyor, ama hepsi aynı `Instant`'a işaret ediyor.

> `MeetingSchedulerDemo`'daki üç şehrin de farklı UTC ofsetlerine sahip olması tesadüf değil — İstanbul'un artık DST uygulamadığını ("Time Zone'lar ve Daylight Saving Time" bölümünü hatırla), New York'un ise uyguladığını görmek, `ZoneId` kurallarının neden bir bölgeye özel olduğunu somutlaştırıyor.

## Ek: Mini Proje — Etkinlik Süre Takibi

Son mini proje, "Legacy API'den java.time'a Geçiş" ve "Duration ve Period" bölümlerindeki fikirleri birleştiriyor: eski bir sistemden gelen `Date` tabanlı etkinlik kayıtlarını `Instant`'a çeviren ve süresini hesaplayan bir takip aracı:

```
import java.time.Duration;
import java.time.Instant;
import java.util.Date;

class EventDurationTracker {
    private final String name;
    private final Instant start;
    private final Instant end;

    EventDurationTracker(String name, Instant start, Instant end) {
        this.name = name;
        this.start = start;
        this.end = end;
    }

    static EventDurationTracker fromLegacyDates(String name, Date legacyStart, Date
legacyEnd) {
        // Bridging old java.util.Date records into the modern API -- lossless,
        // because a Date is already just an epoch timestamp underneath.
        return new EventDurationTracker(name, legacyStart.toInstant(), legacyEnd.
toInstant());
    }

    Duration duration() {
        return Duration.between(start, end);
    }

    String getName() {
        return name;
    }
}
```

```
import java.time.Duration;
import java.util.Date;

class EventDurationTrackerDemo {
    public static void main(String[] args) {
        // A modern record, created directly with Instant.
        EventDurationTracker modern = new EventDurationTracker(
            "Modern Conference",
            java.time.Instant.parse("2026-05-01T09:00:00Z"),
            java.time.Instant.parse("2026-05-01T17:00:00Z")
        );

        // A "legacy" record, as if it came from an old system using java.util.Date.
        Date legacyStart = new Date(1_700_000_000_000L);
        Date legacyEnd = new Date(1_700_010_800_000L);
        EventDurationTracker legacy = EventDurationTracker.fromLegacyDates("Legacy
Workshop", legacyStart, legacyEnd);

        for (EventDurationTracker tracker : new EventDurationTracker[] { modern, legacy }) {
            Duration d = tracker.duration();
            System.out.println(tracker.getName() + ": " + d.toHours() + "h " + (d.
toMinutes() % 60) + "m");
        }
    }
}
```

**Beklenen Çıktı**

```
Modern Conference: 8h 0m
Legacy Workshop: 3h 0m
```

`EventDurationTracker.fromLegacyDates(...)`, eski API'den gelen iki `java.util.Date`'i `toInstant()` ile modern `Instant`'a çeviriyor — bu, "Legacy API'den java.time'a Geçiş" bölümünde gördüğümüz köprünün gerçek bir kullanım örneği. `duration()` metodu ise `Duration.between(...)` ile etkinliğin toplam süresini saat/dakika olarak hesaplıyor; `EventDurationTrackerDemo`, hem eski hem yeni API'den gelen kayıtların aynı `EventDurationTracker` ile sorunsuz işlenebildiğini gösteriyor.

> `fromLegacyDates(...)` içinde `Date.toInstant()` kullanmamızın nedeni, `Date`'in kendisinin **zaten** yalnızca bir epoch zaman damgası tutması — bu yüzden dönüşüm her zaman güvenli ve kayıpsızdır. Ama `Calendar`'dan dönüşüm yaparken bölge bilgisini de hesaba katman gerekebilir; bu, "Legacy API'den java.time'a Geçiş" bölümünde değinmediğimiz, daha ileri bir konu.