Handling Date and Time with the Date-Time API

Esteban Herrera

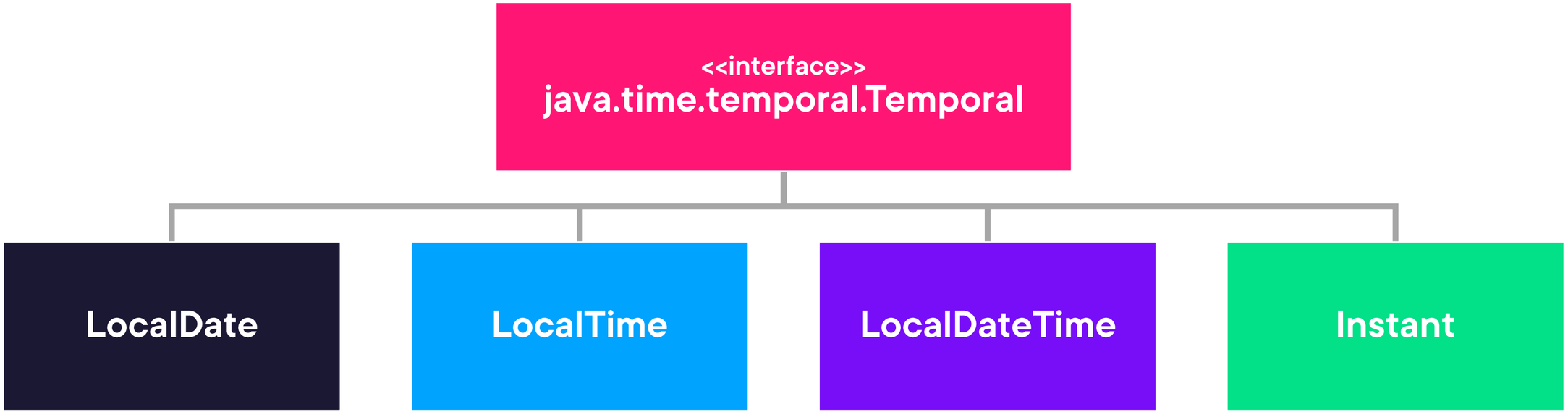
Author

@eh3rrera | eherrera.net

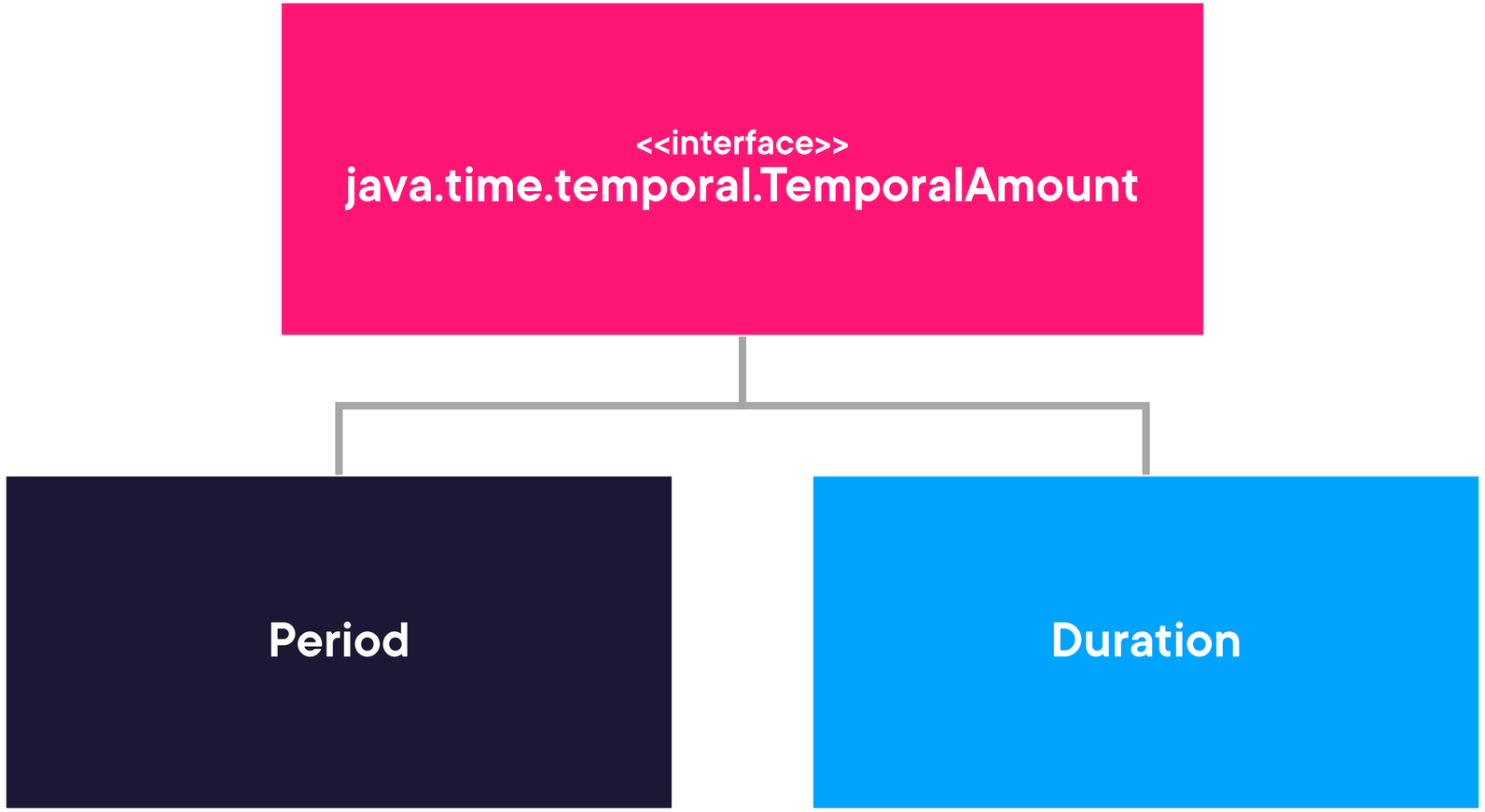
java.time Package

- LocalDate
- LocalTime
- LocalDateTime
- Instant
- Period
- Duration

Temporal Interface



TemporalAmount Interface



Time Zone Information

ZoneId

- ZoneOffset

ZonedDateTime

OffsetDateTime

OffsetTime

All the classes of the Date-Time API are immutable.

Working with LocalDate

A LocalDate Object

Creating a LocalDate Object

```
// With year (-999999999-999999999), month (1-12), day of the month (1-31)
LocalDate newYear2001 = LocalDate.of(2001, 1, 1);

// This version uses the enum java.time.Month
LocalDate newYear2002 = LocalDate.of(2002, Month.JANUARY, 1);
```

LocalDate Methods

```
LocalDate today = LocalDate.now();


int year = today.getYear();
int month = today.getMonthValue();
Month monthAsEnum = today.getMonth(); // As enum: JANUARY, FEBRUARY, etc.
int dayYear = today.getDayOfYear();
int dayMonth = today.getDayOfMonth();
DayOfWeek dayWeekEnum = today.getDayOfWeek(); // As enum: MONDAY, TUESDAY, etc.
```

```
int get(java.time.temporal.TemporalField field); // value as int
long getLong(java.time.temporal.TemporalField field); // value as long
```

Get Methods

LocalDate Methods

```
LocalDate today = LocalDate.now();


int year = today.get(ChronoField.YEAR);
int month = today.get(ChronoField.MONTH_OF_YEAR);
int dayYear = today.get(ChronoField.DAY_OF_YEAR);
int dayMonth = today.get(ChronoField.DAY_OF_MONTH);
int dayWeek = today.get(ChronoField.DAY_OF_WEEK);
long dayEpoch = today.getLong(ChronoField.EPOCH_DAY);
```

ChronoField Values for LocalDate

- DAY_OF_WEEK
- ALIGNED_DAY_OF_WEEK_IN_MONTH
- ALIGNED_DAY_OF_WEEK_IN_YEAR
- DAY_OF_MONTH
- DAY_OF_YEAR
- EPOCH_DAY
- ALIGNED_WEEK_OF_MONTH
- ALIGNED_WEEK_OF_YEAR
- MONTH_OF_YEAR
- PROLEPTIC_MONTH
- YEAR_OF_ERA
- YEAR
- ERA

LocalDate Methods

```
LocalDate newYear2001 = LocalDate.of(2001, 1, 1);
LocalDate newYear2002 = LocalDate.of(2002, 1, 1);

boolean after = newYear2001.isAfter(newYear2002); // false
boolean before = newYear2001.isBefore(newYear2002); // true
boolean equal = newYear2001.equals(newYear2002); // false
boolean leapYear = newYear2001.isLeapYear(); // false
```

LocalDate Methods

```
LocalDate newYear2001 = LocalDate.of(2001, 1, 1);

LocalDate newYear2003 = newYear2001.with(ChronoField.YEAR, 2003);
LocalDate newYear2004 = newYear2001.withYear(2004);
LocalDate december2001 = newYear2001.withMonth(12);
LocalDate february2001 = newYear2001.withDayOfYear(32);

// Since these methods return a new instance, you can chain them
LocalDate xmas2001 = newYear2001.withMonth(12).withDayOfMonth(25);
```

LocalDate Methods

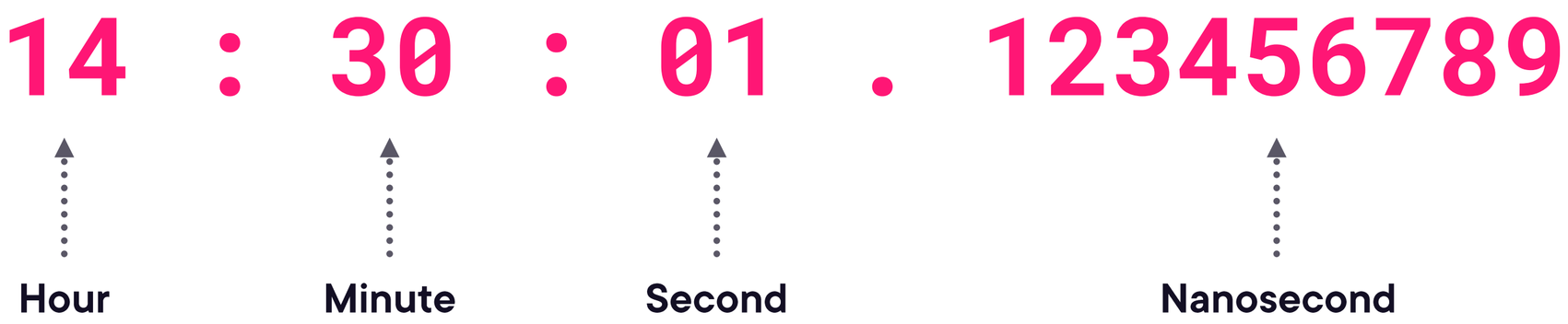
```
LocalDate newYear2001 = LocalDate.of(2001, 1, 1);

// Adding
LocalDate newYear2005 = newYear2001.plusYears(4);
LocalDate march2001 = newYear2001.plusMonths(2);
LocalDate january15_2001 = newYear2001.plusDays(14);
LocalDate lastWeekJanuary2001 = newYear2001.plusWeeks(3);
LocalDate newYear2006 = newYear2001.plus(5, ChronoUnit.YEARS);
// Subtracting
LocalDate newYear2000 = newYear2001.minusYears(1);
LocalDate nov2000 = newYear2001.minusMonths(2);
LocalDate dec30_2000 = newYear2001.minusDays(2);
LocalDate lastWeekDec2000 = newYear2001.minusWeeks(1);
LocalDate newYear1999 = newYear2001.minus(2, ChronoUnit.YEARS);
```

ChronoUnit Values for LocalDate

- DAYS
- WEEKS
- MONTHS
- YEARS
- DECADES
- CENTURIES
- MILLENNIA
- ERAS

Working with LocalTime

A LocalTime Object



LocalTime Methods Similar to LocalDate's

```
// Creating an instance with hours, minutes, and seconds
LocalTime time = LocalTime.of(14, 30, 45); // 14:30:45 (2:30:45 PM)

// Modifying Time with with(), but for time values
LocalTime newHour = time.withHour(16); // 16:30:45 (4:30:45 PM)
LocalTime newMinute = time.withMinute(15); // 14:15:45
LocalTime newSecond = time.withSecond(10); // 14:30:10

// Adding/Subtracting Time with methods equivalent to plusDays() or minusMonths()
LocalTime plusHours = time.plusHours(2); // 16:30:45 (adds 2 hours)
LocalTime minusMinutes = time.minusMinutes(10); // 14:20:45 (subtracts 10 minutes)
```

LocalTime Methods Similar to LocalDate's

```
LocalTime time = LocalTime.of(14, 30, 45); // 14:30:45 (2:30:45 PM)

// Retrieving Values similar to getYear(), getMonthValue(), etc.
int hour = time.getHour(); // 14
int minute = time.getMinute(); // 30
int second = time.getSecond(); // 45

// Comparing times
boolean isBefore = time.isBefore(LocalTime.of(16, 0)); // true
boolean isAfter = time.isAfter(LocalTime.of(12, 0)); // true

// toString() returns time in ISO-8601 format, hour minutes seconds
System.out.println(time); // Output: 14:30:45
```

Creating a LocalTime Object

```
// With hour (0-23) and minutes (0-59)
LocalTime fiveThirty = LocalTime.of(5, 30);

// With hour, minutes, and seconds (0-59)
LocalTime noon = LocalTime.of(12, 0, 0);

// With hour, minutes, seconds, and nanoseconds (0-999_999_999)
LocalTime almostMidnight = LocalTime.of(23, 59, 59, 999_999_999);
```

LocalTime Methods

```
LocalTime now = LocalTime.now();


int hour = now.getHour();
int minute = now.getMinute();
int second = now.getSecond();
int nanosecond = now.getNano();
```

```
int get(java.time.temporal.TemporalField field); // value as int
long getLong(java.time.temporal.TemporalField field); // value as long
```

Get Methods

LocalTime Methods

```
LocalTime now = LocalTime.now();


int hourAMPM = now.get(ChronoField.HOUR_OF_AMPM); // 0 - 11
int hourDay = now.get(ChronoField.HOUR_OF_DAY); // 0 - 23
int minuteDay = now.get(ChronoField.MINUTE_OF_DAY); // 0 - 1,439
int minuteHour = now.get(ChronoField.MINUTE_OF_HOUR); // 0 - 59
int secondDay = now.get(ChronoField.SECOND_OF_DAY); // 0 - 86,399
int secondMinute = now.get(ChronoField.SECOND_OF_MINUTE); // 0 - 59
long nanoDay = now.getLong(ChronoField.NANO_OF_DAY); // 0-86_399_999_999
int nanoSecond = now.get(ChronoField.NANO_OF_SECOND); // 0-999_999_999
```

ChronoField Values for LocalTime

- NANO_OF_SECOND
- NANO_OF_DAY
- MICRO_OF_SECOND
- MICRO_OF_DAY
- MILLI_OF_SECOND
- MILLI_OF_DAY
- SECOND_OF_MINUTE
- SECOND_OF_DAY
- MINUTE_OF_HOUR
- MINUTE_OF_DAY
- HOUR_OF_AMPM
- CLOCK_HOUR_OF_AMPM
- HOUR_OF_DAY
- CLOCK_HOUR_OF_DAY
- AMPM_OF_DAY

LocalTime Methods

```
LocalTime fiveThirty = LocalTime.of(5, 30);
LocalTime noon = LocalTime.of(12, 0, 0);

boolean after = fiveThirty.isAfter(noon); // false
boolean before = fiveThirty.isBefore(noon); // true
boolean equal = fiveThirty.equals(noon); // false
```

LocalTime Methods

```
LocalTime noon = LocalTime.of(12, 0, 0);

LocalTime ten = noon.with(ChronoField.HOUR_OF_DAY, 10);
LocalTime eight = noon.withHour(8);
LocalTime twelveThirty = noon.withMinute(30);
LocalTime thirtyTwoSeconds = noon.withSecond(32);

// Since these methods return a new instance, you can chain them
LocalTime secondsNano = noon.withSecond(20).withNano(999_999);
```

LocalTime Methods

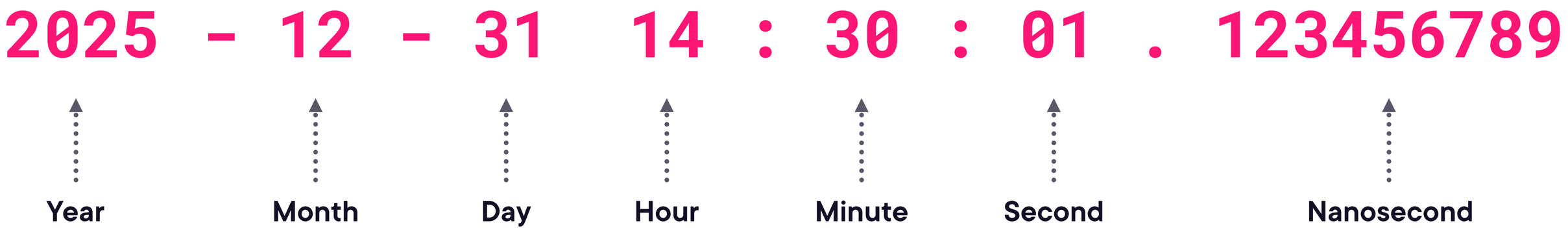
```
LocalTime fiveThirty = LocalTime.of(5, 30);

// Adding
LocalTime sixThirty = fiveThirty.plusHours(1);
LocalTime fiveForty = fiveThirty.plusMinutes(10);
LocalTime plusSeconds = fiveThirty.plusSeconds(14);
LocalTime plusNanos = fiveThirty.plusNanos(99_999_999);
LocalTime sevenThirty = fiveThirty.plus(2, ChronoUnit.HOURS);
// Subtracting
LocalTime fourThirty = fiveThirty.minusHours(1);
LocalTime fiveTen = fiveThirty.minusMinutes(20);
LocalTime minusSeconds = fiveThirty.minusSeconds(2);
LocalTime minusNanos = fiveThirty.minusNanos(1);
LocalTime fiveTwenty = fiveThirty.minus(10, ChronoUnit.MINUTES);
```

ChronoUnit Values for LocalTime

- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS

Working with LocalDateTime

A LocalDateTime Object



Creating a LocalDateTime Object

```
// Setting seconds and nanoseconds to zero
LocalDateTime dt1 = LocalDateTime.of(2024, 9, 19, 14, 5);

// Setting nanoseconds to zero
LocalDateTime dt2 = LocalDateTime.of(2024, 9, 19, 14, 5, 20);

// Setting all fields
LocalDateTime dt3 = LocalDateTime.of(2024, 9, 19, 14, 5, 20, 9);
```

Creating a LocalDateTime Object

```
// Assuming this date
LocalDate date = LocalDate.now();
// And this time
LocalTime time = LocalTime.now();

// Combine the date with the time
LocalDateTime dt4 = date.atTime(14, 30, 59, 999999);
LocalDateTime dt5 = date.atTime(time);
// Combine the time with the date
LocalDateTime dt6 = time.atDate(date);
```

LocalDateTime Methods

```
LocalDateTime now = LocalDateTime.now();


int year = now.getYear();
int dayYear = now.getDayOfYear();
int hour = now.getHour();
int minute = now.getMinute();
```

```
int get(java.time.temporal.TemporalField field); // value as int
long getLong(java.time.temporal.TemporalField field); // value as long
```

Get Methods

LocalDateTime Methods

```
LocalTime now = LocalDateTime.now();


int month = now.get(ChronoField.MONTH_OF_YEAR);
int minuteHour = now.get(ChronoField.MINUTE_OF_HOUR);
```

ChronoField Values for LocalDateTime

- NANO_OF_SECOND
- NANO_OF_DAY
- MICRO_OF_SECOND
- MICRO_OF_DAY
- MILLI_OF_SECOND
- MILLI_OF_DAY
- SECOND_OF_MINUTE
- SECOND_OF_DAY
- MINUTE_OF_HOUR
- MINUTE_OF_DAY
- HOUR_OF_AMPM
- CLOCK_HOUR_OF_AMPM
- HOUR_OF_DAY
- CLOCK_HOUR_OF_DAY
- AMPM_OF_DAY
- DAY_OF_WEEK
- ALIGNED_DAY_OF_WEEK_IN_MONTH
- ALIGNED_DAY_OF_WEEK_IN_YEAR
- DAY_OF_MONTH
- DAY_OF_YEAR
- EPOCH_DAY
- ALIGNED_WEEK_OF_MONTH
- ALIGNED_WEEK_OF_YEAR
- MONTH_OF_YEAR
- PROLEPTIC_MONTH
- YEAR_OF_ERA
- YEAR
- ERA

LocalDateTime Methods

```
LocalTime now = LocalDateTime.now();
LocalDateTime dt1 = LocalDateTime.of(2024, 9, 19, 14, 5);

boolean after = now.isAfter(dt1); // false
boolean before = now.isBefore(dt1); // true
boolean equal = now.equals(dt1); // false
```

LocalDateTime Methods

```
LocalTime now = LocalDateTime.now();

LocalDateTime dt1 = now.with(ChronoField.HOUR_OF_DAY, 10);
LocalDateTime dt2 = now.withMonth(8);
// Since these methods return a new instance, you can chain them
LocalDateTime dt3 = now.withYear(2013).withMinute(0);
```

LocalDateTime Methods

```
LocalTime now = LocalDateTime.now();


// Adding
LocalDateTime dt1 = now.plusYears(4);
LocalDateTime dt2 = now.plusWeeks(3);
LocalDateTime dt3 = now.plus(2, ChronoUnit.HOURS);
// Subtracting
LocalDateTime dt4 = now.minusMonths(2);
LocalDateTime dt5 = now.minusNanos(1);
LocalDateTime dt6 = now.minus(10, ChronoUnit.SECONDS);
```

ChronoUnit Values for LocalDateTime

- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS
- DAYS
- WEEKS
- MONTHS
- YEARS
- DECADES
- CENTURIES
- MILLENNIA
- ERAS

Working with Instant

The Instant Class

Instant Representation

A long represents epoch-seconds

An int for nanosecond-of-second (always between 0 and 999,999,999)

Positive values after the epoch

Negative values before the epoch

Creating an Instant Object

```
// Setting seconds
Instant fiveSecondsAfterEpoch = Instant.ofEpochSecond(5);

// Setting seconds and nanoseconds (negative for before epoch)
Instant sixSecTwoNanBeforeEpoch = Instant.ofEpochSecond(-6, -2);

// Setting milliseconds since epoch (negative for before epoch)
Instant fiftyMillisecondsAfterEpoch = Instant.ofEpochMilli(50);
```

Instant Methods

```
Instant now = Instant.now();


long seconds = now.getEpochSecond(); // Gets the seconds
int nanos1 = now.getNano(); // Gets the nanoseconds


// Gets the value as an int
int millis = now.get(ChronoField.MILLI_OF_SECOND);
// Gets the value as a long
long nanos2 = now.getLong(ChronoField.NANO_OF_SECOND);
```

ChronoField Values for Instant

- `NANO_OF_SECOND`
- `MICRO_OF_SECOND`
- `MILLI_OF_SECOND`
- `INSTANT_SECONDS`

Instant Methods

```
Instant now = Instant.now();
Instant fiveSecondsAfterEpoch = Instant.ofEpochSecond(5);

boolean after = now.isAfter(fiveSecondsAfterEpoch); // true
boolean before = now.isBefore(fiveSecondsAfterEpoch); // false
boolean equal = now.equals(fiveSecondsAfterEpoch); // false
```

Instant Methods

```
Instant now = Instant.now();


// Creating a new Instant with the nanosecond field set to 10
Instant nanoInstant = now.with(ChronoField.NANO_OF_SECOND, 10);


// Using an Instant as a TemporalAdjuster
Instant epochInstant = now.with(Instant.EPOCH);
// Creating a new Instant with a custom implementation of TemporalAdjuster
Instant zeroMillisecondInstant = now.with(
    temporal -> temporal.with(ChronoField.MILLI_OF_SECOND, 0)
);
```

Instant Methods

```
Instant now = Instant.now();

// Adding
Instant i1 = now.plusSeconds(400);
Instant i2 = now.plusMillis(98622200);
Instant i3 = now.plusNanos(300013890);
Instant i4 = now.plus(2, ChronoUnit.MINUTES);

// Subtracting
Instant i5 = now.minusSeconds(2);
Instant i6 = now.minusMillis(1);
Instant i7 = now.minusNanos(1);
Instant i8 = now.minus(10, ChronoUnit.SECONDS);
```

ChronoUnit Values for Instant

- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS
- DAYS

Working with Period

A Period Object

Creating a Period Object

```
// Setting years, months, days (all can be negative)
Period period5y4m3d = Period.of(5, 4, 3);
// Setting days (can be negative), years and months will be zero
Period period2d = Period.ofDays(2);
// Setting months (can be negative), years and days will be zero
Period period2m = Period.ofMonths(2);
// Setting weeks (can be negative). The resulting period will
// be in days (1 week = 7 days). Years and months will be zero
Period period14d = Period.ofWeeks(2);
// Setting years (can be negative), days and months will be zero
Period period2y = Period.ofYears(2);
```

```
public static Period between(LocalDate startDateInclusive,
                             LocalDate endDateExclusive)
```

Calculating the Difference Between Two Dates

Use the Period.between method

Rules for Calculating the Difference between Dates

Complete months are counted, and then the remaining number of days

The number of months is then split into years (1 year equals 12 months)

A month is counted if the end day is greater than or equal to the start day

The result can be negative if the end date is before the start date

Period Methods

```
Period period5y4m3d = Period.of(5, 4, 3);


int days = period5y4m3d.getDays();
int months = period5y4m3d.getMonths();
int year = period5y4m3d.getYears();
long days2 = period5y4m3d.get(ChronoUnit.DAYS);
```

ChronoUnit Values for Period

- DAYS
- MONTHS
- YEARS

Period Methods

```
Period period2d = Period.ofDays(2);

Period period8d = period2d.withDays(8);

// Since these methods return a new instance, you can chain them
Period period2y1m2d = period2d.withYears(2).withMonths(1);
```

Period Methods

```
Period period5y4m3d = Period.of(5, 4, 3);

// Adding
Period period9y4m3d = period5y4m3d.plusYears(4);
Period period5y7m3d = period5y4m3d.plusMonths(3);
Period period5y4m6d = period5y4m3d.plusDays(3);
Period period7y4m3d = period5y4m3d.plus(period5y4m6d);
// Subtracting
Period period5y4m1d = period5y4m3d.minusYears(2);
Period period5y3m3d = period5y4m3d.minusMonths(1);
Period period5y4m2d = period5y4m3d.minusDays(1);
Period period3y4m3d = period5y4m3d.minus(period5y4m2d);
```

```
String toString() // Returns the period in the format P<years>Y<months>M<days>D
```

The toString() Method

Working with Duration

A Duration Object

Creating a Duration Object

```
Duration oneDay = Duration.ofDays(1); // 1 day = 86400 seconds
Duration oneHour = Duration.ofHours(1); // 1 hour = 3600 seconds
Duration oneMin = Duration.ofMinutes(1); // 1 minute = 60 seconds
Duration tenSeconds = Duration.ofSeconds(10);
// Set seconds and nanoseconds
// Nanoseconds adjust to 0-999,999,999. Excess nanoseconds carry over to seconds
Duration twoSeconds = Duration.ofSeconds(1, 1000000000);
// Seconds and nanoseconds are extracted from the passed milliseconds
Duration twoMilliseconds = Duration.ofMillis(2);
// Seconds and nanoseconds are extracted from the passed nanos
Duration oneSecondFromNanos = Duration.ofNanos(1000000000);
Duration oneSecond = Duration.of(1, ChronoUnit.SECONDS);
```

ChronoUnit Values for Duration

- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS
- DAYS

Duration Methods

```java
Duration oneSecond = Duration.of(1, ChronoUnit.SECONDS);

// The nanoseconds part of the duration, from 0 to 999,999,999
int nanos = oneSecond.getNano();
// The seconds part of the duration, positive or negative
long seconds = oneSecond.getSeconds();
// It supports SECONDS and NANOS. Other units throw an exception
long oneSec = oneSecond.get(ChronoUnit.SECONDS);
```

```
// Temporal objects must support seconds, and for more accuracy, nanoseconds,
// like LocalTime, LocalDateTime, and Instant
public static Duration between(Temporal startInclusive, Temporal endExclusive)
```

Calculating the Difference Between Two Temporal Objects

Use the Duration.between method.
If the objects are of different types, then the duration is calculated based on the type of the first object.

Duration Methods

```
Duration oneSecond = Duration.of(1, ChronoUnit.SECONDS);

Duration duration1sec8nan = oneSecond.withNanos(8);
Duration duration2sec1nan = oneSecond.withSeconds(2).withNanos(1);
```

Duration Methods

```
// Adding
Duration plus4Days = oneSecond.plusDays(4);
Duration plus3Hours = oneSecond.plusHours(3);
Duration plus3Minutes = oneSecond.plusMinutes(3);
Duration plus3Seconds = oneSecond.plusSeconds(3);
Duration plus3Millis = oneSecond.plusMillis(3);
Duration plus3Nanos = oneSecond.plusNanos(3);
// Subtracting
Duration minus4Days = oneSecond.minusDays(4);
Duration minus3Hours = oneSecond.minusHours(3);
Duration minus3Minutes = oneSecond.minusMinutes(3);
Duration minus3Seconds = oneSecond.minusSeconds(3);
Duration minus3Millis = oneSecond.minusMillis(3);
Duration minus3Nanos = oneSecond.minusNanos(3);
```

Duration Methods

```
Duration oneSecond = Duration.of(1, ChronoUnit.SECONDS);
Duration twoSeconds = Duration.of(2, ChronoUnit.SECONDS);

// Adding
Duration plusAnotherDuration = oneSecond.plus(twoSeconds);
Duration plusChronoUnits = oneSecond.plus(1, ChronoUnit.DAYS);

// Subtracting
Duration minusAnotherDuration = oneSecond.minus(twoSeconds);
Duration minusChronoUnits = oneSecond.minus(1, ChronoUnit.DAYS);
```

ChronoUnit Values for Duration

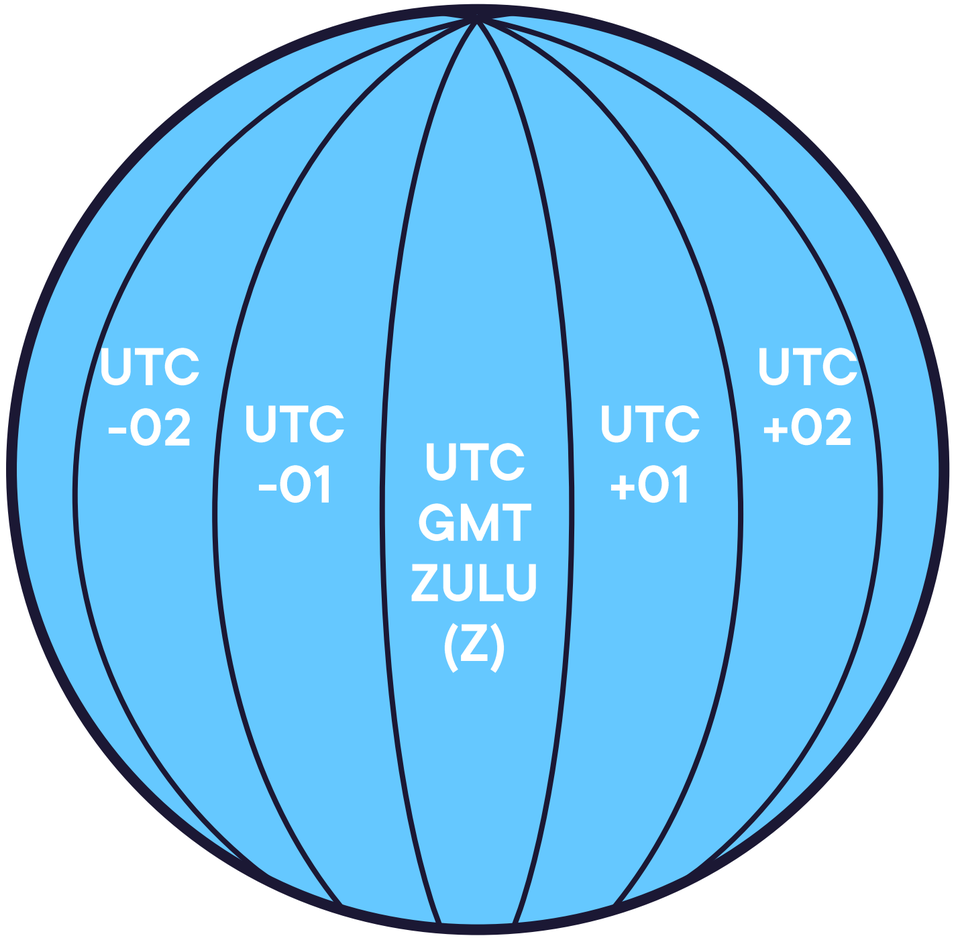
- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS
- DAYS

```
String toString() // Returns the duration in the format PTnHnMnS
```

The toString() Method

Working with ZoneId and ZoneOffset

Time Zones



IANA

Internet Assigned Numbers Authority (IANA)
Time Zone Database (TZDB)

java.time.ZoneId

Each time zone has an ID represented by java.time.ZoneId:

- Region-based, using the format area/city (Europe/London)
- Offset from UTC/GMT time (+02:00), represented by the ZoneOffset class
- Offset with UTC, GMT, or UT prefix (UTC+11:00), represented by the ZoneOffset class

```
Set<String> getAvailableZoneIds()
```

Get All Available Time Zones

```
public static ZoneId systemDefault() // ZoneId.systemDefault()
```

Get Default Time Zone

It calls java.util.TimeZone.getDefault() to find and convert the default time zone to a ZoneId

Creating a ZoneId Object

```
ZoneId singaporeZoneId = ZoneId.of("Asia/Singapore");

ZoneId zoneId = ZoneId.of("Z"); // Z represents the UTC ID (as ZoneOffset)
ZoneId zoneId2 = ZoneId.of("-2"); // -02:00 (as ZoneOffset)
```

Creating a ZoneOffset Object

```
// The offset must be in the range of -18 to +18
ZoneOffset offsetHours = ZoneOffset.ofHours(1);
// The range is -18 to +18 for hours and 0 to ± 59 for minutes
// If the hours are negative, the minutes must be negative or zero
ZoneOffset offsetHrMin = ZoneOffset.ofHoursMinutes(1, 30);
// The range is -18 to +18 for hours and 0 to ± 59 for mins and secs
// If the hours are negative, mins and secs must be negative or zero
ZoneOffset offsetHrMinSe = ZoneOffset.ofHoursMinutesSeconds(1, 30, 0);
// The offset must be in the range -18:00 to +18:00
// Which corresponds to -64800 to +64800
ZoneOffset offsetTotalSeconds = ZoneOffset.ofTotalSeconds(3600);
// The range must be from -18:00 to +18:00
ZoneOffset offset = ZoneOffset.of("+01:30:00");
```

Formats Accepted by the ZoneOffset.of() Method

- `Z`
- `+h`
- `+hh`
- `+hh:mm`
- `-hh:mm`
- `+hhmm`
- `-hhmm`
- `+hh:mm:ss`
- `-hh:mm:ss`
- `+hhmmss`
- `-hhmmss`

ZoneOffset Methods

```java
ZoneOffset offset = ZoneOffset.of("+01:30:00");

// Gets the offset as int
int offsetInt = offset.get(ChronoField.OFFSET_SECONDS);

// Gets the offset as long
long offsetLong= offset.getLong(ChronoField.OFFSET_SECONDS);

// Gets the offset in seconds
int offsetSeconds = offset.getTotalSeconds();
```

Working with ZonedDateTime, OffsetDateTime and OffsetTime

A ZonedDateTime Object

Creating a ZonedDateTime Object

```
ZoneId australiaZone = ZoneId.of("Australia/Victoria");

LocalDate date = LocalDate.of(2020, 7, 3);
ZonedDateTime zonedDate = date.atStartOfDay(australiaZone);

LocalDateTime dateTime = LocalDateTime.of(2020, 7, 3, 9, 0);
ZonedDateTime zonedDateTime = dateTime.atZone(australiaZone);

Instant instant = Instant.now();
ZonedDateTime zonedInstant = instant.atZone(australiaZone);
```

Creating a ZonedDateTime Object

```
ZoneId australiaZone = ZoneId.of("Australia/Victoria");

ZonedDateTime zonedDateTime =
    ZonedDateTime.of(LocalDate.now(), LocalTime.now(), australiaZone);

ZonedDateTime zonedDateTime2 =
    ZonedDateTime.of(LocalDateTime.now(), australiaZone);

// year, month, day, hours, minutes, seconds, nanoseconds, zoneId
ZonedDateTime zonedDateTime3 =
    ZonedDateTime.of(2025, 1, 30, 13, 59, 59, 999, australiaZone);
```

Creating a ZonedDateTime Object

```
ZoneId australiaZone = ZoneId.of("Australia/Victoria");

ZonedDateTime zonedDateTimeInstant1 = ZonedDateTime.ofInstant(
        Instant.now(), australiaZone
);

ZonedDateTime zonedDateTimeInstant2 = ZonedDateTime.ofInstant(
        LocalDateTime.now(), ZoneOffset.of("+01:00:00"), australiaZone
);
```

ZonedDateTime Methods

```
ZonedDateTime now = ZonedDateTime.now();

LocalDate currentDate = now.toLocalDate();
LocalTime currentTime = now.toLocalTime();
LocalDateTime currentDateTime = now.toLocalDateTime();
```

ZonedDateTime Methods

```
ZonedDateTime now = ZonedDateTime.now();

// To get the value of a specified field
int day = now.getDayOfMonth();
int dayYear = now.getDayOfYear();
int nanos = now.getNano();
Month monthEnum = now.getMonth();
ZoneOffset offset = now.getOffset(); // Gets the zone offset, such as "-03:00"

int year = now.get(ChronoField.YEAR);
long micro = now.getLong(ChronoField.MICRO_OF_DAY);
```

ZonedDateTime Methods

```
ZonedDateTime now = ZonedDateTime.now();

// To create another instance
ZonedDateTime zdt1 = now.with(ChronoField.HOUR_OF_DAY, 10);
ZonedDateTime zdt2 = now.withSecond(49);

// Since these methods return a new instance, you can chain them
ZonedDateTime zdt3 = now.withYear(2023).withMonth(12);
```

ZonedDateTime Methods

```
ZonedDateTime now = ZonedDateTime.now();

// Adding
ZonedDateTime zdt1 = now.plusDays(4);
ZonedDateTime zdt2 = now.plusWeeks(3);
ZonedDateTime zdt3 = now.plus(2, ChronoUnit.HOURS);

// Subtracting
ZonedDateTime zdt4 = now.minusMinutes(20);
ZonedDateTime zdt5 = now.minusNanos(99999);
ZonedDateTime zdt6 = now.minus(10, ChronoUnit.SECONDS);
```

ZonedDateTime Methods

```
ZonedDateTime now = ZonedDateTime.now();
ZoneId australiaZone = ZoneId.of("Australia/Victoria");

// Returns a copy of the date/time with a different zone, retaining the instant
ZonedDateTime zdt1 = now.withZoneSameInstant(australiaZone);

// Returns a copy of this date/time with a different time zone,
// retaining the local date/time if it's valid for the new time zone
ZonedDateTime zdt2 = now.withZoneSameLocal(australiaZone);
```

```
String toString() // For example: 2001-11-30T16:20:10.123456789+09:00[Asia/Tokyo]
```

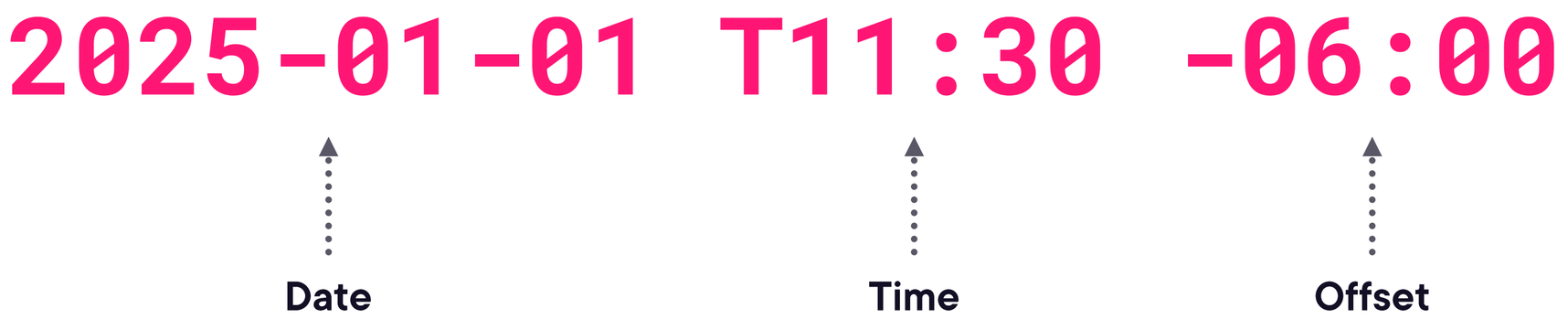
ZonedDateTime toString() Method

Offset Classes

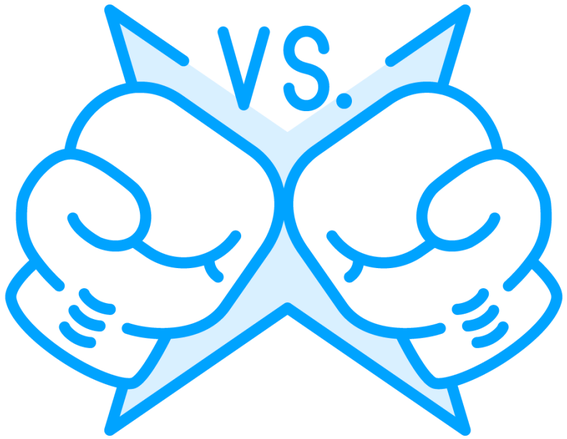
OffsetDateTime

OffsetTime

An OffsetDateTime Object



Instant vs. OffsetDateTime vs. ZonedDateTime



Instant is a point in time in UTC

OffsetDateTime is a point in time with any offset

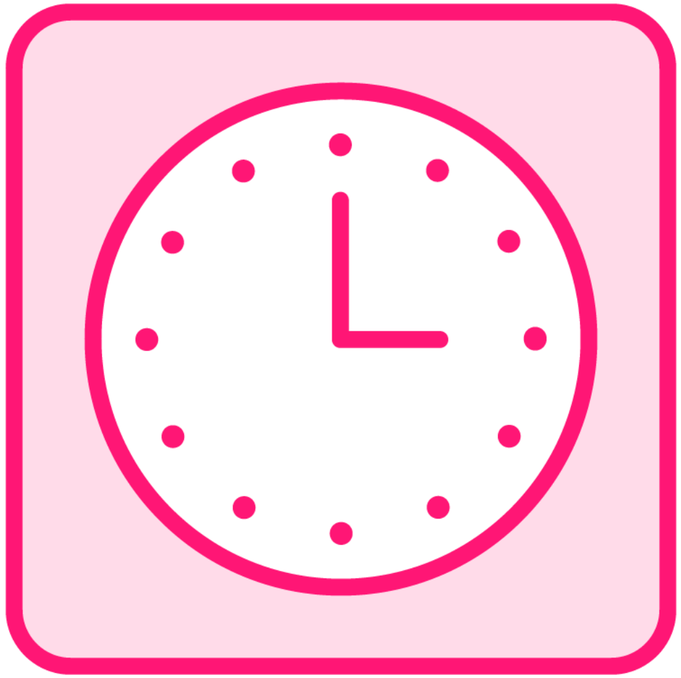
ZonedDateTime is a point in time in any time zone with full time zone rules

An OffsetTime Object

Daylight Savings

Daylight Saving Time (DST)

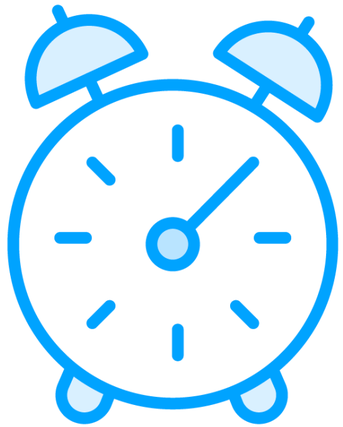
Daylight Saving Time (DST)



Daylight Saving Time (DST)

ZonedDateTime is completely aware of DST.

Daylight Saving Time in Italy



UTC+1 in standard time

UTC+2 in DST

In 2024:

- DST start: March 31
- DST end: October 27
- March 31, 2:00 A.M. → 3:00 A.M. (UTC+2)
- October 27, 3:00 A.M. → 2:00 A.M. (UTC+1)

Parsing and Formatting

```
// Formats the date/time object using the specified formatter
String format(DateTimeFormatter formatter)

// Gets an instance of a date/time object (of type T)
// from a string with default format
static T parse(CharSequence text)

// Gets an instance of a date/time object (of type T)
// from a string using a specific formatter
static T parse(CharSequence text, DateTimeFormatter formatter)
```

Format and Parse Methods of the Date and Time Classes

T represents LocalDate, LocalTime, LocalDateTime, ZonedDateTime, OffsetTime, and OffsetDateTime

```
// Formats a date/time object using the formatter instance
String format(TemporalAccessor temporal)

// Parses the text producing a temporal object
TemporalAccessor parse(CharSequence text)
<T> T parse(CharSequence text, TemporalQuery<T> query)
```

Format and Parse Methods of the DateTimeFormatter Class

Three Ways to Format Date and Time Objects

Predefined formatters

Locale-specific formatters

Formatters with custom patterns

Format Methods

```
LocalDateTime now = LocalDateTime.now();

// Predefined formatter
DateTimeFormatter isoFormatter = DateTimeFormatter.ISO_LOCAL_DATE_TIME;
String isoUsingLocalDateTime = now.format(isoFormatter);
String isoUsingFormatter = isoFormatter.format(now);

// Locale-specific formatter with MEDIUM style
DateTimeFormatter mediumFormatter =  DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
String mediumUsingLocalDateTime = now.format(mediumFormatter);
String mediumUsingFormatter = mediumFormatter.format(now);

// Custom pattern
DateTimeFormatter customFormatter =  DateTimeFormatter.ofPattern("yyyy.MM.dd HH:mm");
String customUsingLocalDateTime = now.format(customFormatter);
String customUsingFormatter = customFormatter.format(now);
```

Parsing Methods

```
String iso8601String = "2025-06-29T14:45:30"; // Format according to ISO-8601
String customFormatString = "2025/06/29 14:45:30"; // Custom format

// Automatically recognizes ISO-8601 standard
LocalDateTime ldt1 = LocalDateTime.parse(iso8601String);

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");

// DateTimeFormatter's parse method returns a TemporalAccessor instance
TemporalAccessor ta = formatter.parse(customFormatString);
// Specify the type of object, typically using a method reference to a from() method
LocalDateTime ldt2 = formatter.parse(customFormatString, LocalDateTime::from);

// LocalDateTime's parse method returns an instance of the same type
LocalDateTime ldt3 = LocalDateTime.parse(customFormatString, formatter);
```

Default Format for Parse Methods

Class	Format	Example
LocalDate	uuuu-MM-dd	2024-12-03
LocalTime	HH:mm:ss	10:15
LocalDateTime	uuuu-MM-dd'T'HH:mm:ss	2024-12-03T10:15:30
ZonedDateTime	uuuu-MM-dd'T'HH:mm:ssXXXXX[VV]	2023-12-03T10:15:30+01:00[Europe/Paris]
OffsetDateTime	uuuu-MM-dd'T'HH:mm:ssXXXXX	2023-12-03T10:15:30+01:00
OffsetTime	HH:mm:ssXXXXX	10:15:30+01:00

Review

Overview of Core Date-Time API Classes

Class	Represents	Key Features
LocalDate	Date (year, month, day)	Immutable, thread-safe
LocalTime	Time (hour, minute, second, nanosecond)	Ideal for time-specific operations
LocalDateTime	Combination of date and time (no timezone)	Combines LocalDate and LocalTime
Instant	A moment on the timeline in UTC	High precision timestamp for machine time
Period	Date-based amount (years, months, days)	Useful for calendar arithmetic (e.g., birthdays)
Duration	Time-based amount (seconds, nanoseconds)	Suitable for elapsed time measurements
ZoneId	Represents a region-based time zone	Used for identifying time zone rules
ZoneOffset	Represents a fixed offset from UTC	Simplified time zone offset information
ZonedDateTime	Date and time with timezone information	Combines LocalDateTime with ZoneId and handles DST
OffsetDateTime	Date and time with a fixed offset from UTC	Combines LocalDateTime with a ZoneOffset
OffsetTime	Time with a fixed offset from UTC	Similar to OffsetDateTime but only for time

Period vs. Duration

Aspect	Period	Duration
Measurement Unit	Years, months, days	Seconds, nanoseconds
Use Case	Calendar-based arithmetic (e.g., birthdays)	Time-based arithmetic (e.g., elapsed time)
Example Methods	Period.ofDays(), Period.between()	Duration.ofSeconds(), Duration.between()
Handling Ambiguities	Accounts for variable month lengths	Fixed length; no ambiguity in seconds

Instant vs. OffsetDateTime vs. ZonedDateTime

Aspect	Instant	OffsetDateTime	ZonedDateTime
Representation	Absolute timestamp in UTC	Local date-time with a fixed offset from UTC	Local date-time with full time zone information (ZoneId)
Time Zone/Offset	No time zone or offset information beyond UTC	Includes a ZoneOffset (e.g., +02:00)	Includes both a ZoneId (e.g., Europe/Paris) and an offset
Use Case	High-precision timestamps and machine time	When you need a human-readable date-time along with an offset	When full time zone rules (including DST) are required
Conversion Behavior	Always in UTC, conversion required for local representations	Fixed offset may differ from local time zone rules	Automatically adjusts for DST and other zone-specific rules

Practice Questions

Test your knowledge.

Question 1

What will be the output of the following program?

```
public class DateTimeTest {
    public static void main(String[] args) {
        Period period = Period.of(0, 1, 15);
        Duration duration = Duration.ofDays(1);
        LocalDate date = LocalDate.of(2024, 2, 14);
        LocalTime time = LocalTime.of(12, 0);
        LocalDateTime dateTime = LocalDateTime.of(date, time);

        dateTime = dateTime.plus(period).plus(duration);
        System.out.println(dateTime);
    }
}
```

A) 2024-02-29T12:00 ✗

B) 2024-03-30T12:00 ✓

C) 2024-03-02T12:00 ✗

D) 2024-03-16T12:00 ✗

```
February 14, 2024 at 12:00
March 14, 2024 at 12:00
March 29, 2024 at 12:00
March 30, 2024 at 12:00
```

Question 2

What will be the output of this program, given that March 10, 2024 at 2:00 AM, is the start of DST in America/New_York (UTC-5 in standard time, UTC-4 in DST)?

```java
public class DSTTest {
    public static void main(String[] args) {
        ZoneId zone = ZoneId.of("America/New_York");
        LocalDate date = LocalDate.of(2024, 3, 10);
        LocalTime time = LocalTime.of(2, 30);
        ZonedDateTime zdt = ZonedDateTime.of(date, time, zone);

        System.out.println(zdt);
    }
}
```

A) 2024-03-10T02:30-05:00[America/New_York] ✗

B) 2024-03-10T02:30-04:00[America/New_York] ✗

C) 2024-03-10T03:30-05:00[America/New_York] ✗

D) 2024-03-10T03:30-04:00[America/New_York] ✓

E) It throws an exception at runtime ✗

March 10, 2024 at 2:00 AM

March 10, 2024 at 3:00 AM

~~March 10, 2024 at 2:30 AM~~

March 10, 2024 at 3:30 AM UTC-4

Thank you for your time!

@eh3rrera | eherrera.net