Tutorials Watch & Listen FAQ Oracle University

Home > Tutorials > The Date Time API > Time Zone and Offset

Date and Time ← Time Zone and Offset → Instant

Time Zone and Offset

A *time zone* is a region of the earth where the same standard time is used. Each time zone is described by an identifier and usually has the format *region/city* (`Asia/Tokyo`) and an offset from Greenwich/UTC time. For example, the offset for Tokyo is `+09:00`.

The ZoneId and ZoneOffset Classes

The Date-Time API provides two classes for specifying a time zone or an offset:

- `ZoneId` specifies a time zone identifier and provides rules for converting between an Instant and a `LocalDateTime`.
- `ZoneOffset` specifies a time zone offset from Greenwich/UTC time.

Offsets from Greenwich/UTC time are usually defined in whole hours, but there are exceptions. The following code prints a list of all time zones that use offsets from Greenwich/UTC that are not defined in whole hours.

```
Set<String> allZones = ZoneId.getAvailableZoneIds();
LocalDateTime dt = LocalDateTime.now();

// Create a List using the set of zones and sort it.
List<String> zoneList = new ArrayList<>(allZones).sort();

for (String zone : zoneList) {
    ZoneId zone = ZoneId.of(zone);
    ZonedDateTime zdt = dt.atZone(zone);
    ZoneOffset offset = zdt.getOffset();
    int secondsOfHour = offset.getTotalSeconds() % (60 * 60);
    String out = String.format("%35s %10s%n", zone, offset);

    // Write only time zones that do not have a whole hour offset
```

```
        // to standard out.
        if (secondsOfHour != 0) {
            System.out.printf(out);
        }
    }
```

This example prints the following list to standard out:

```
      America/Caracas     -04:30
     America/St_Johns     -02:30
        Asia/Calcutta     +05:30
         Asia/Colombo     +05:30
           Asia/Kabul     +04:30
       Asia/Kathmandu     +05:45
        Asia/Katmandu     +05:45
         Asia/Kolkata     +05:30
         Asia/Rangoon     +06:30
          Asia/Tehran     +04:30
   Australia/Adelaide     +09:30
Australia/Broken_Hill     +09:30
     Australia/Darwin     +09:30
      Australia/Eucla     +08:45
        Australia/LHI     +10:30
  Australia/Lord_Howe     +10:30
      Australia/North     +09:30
      Australia/South     +09:30
 Australia/Yancowinna     +09:30
  Canada/Newfoundland     -02:30
         Indian/Cocos     +06:30
                 Iran     +04:30
              NZ-CHAT     +12:45
      Pacific/Chatham     +12:45
    Pacific/Marquesas     -09:30
      Pacific/Norfolk     +11:30
```

The Date Time Classes

The Date-Time API provides three temporal-based classes that work with time zones:

- `ZonedDateTime` handles a date and time with a corresponding time zone with a time zone offset from Greenwich/UTC.
- `OffsetDateTime` handles a date and time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.

- OffsetTime handles time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.

When would you use OffsetDateTime instead of ZonedDateTime? If you are writing complex software that models its own rules for date and time calculations based on geographic locations, or if you are storing time-stamps in a database that track only absolute offsets from Greenwich/UTC time, then you might want to use OffsetDateTime. Also, XML and other network formats define date-time transfer as OffsetDateTime or OffsetTime.

Although all three classes maintain an offset from Greenwich/UTC time, only ZonedDateTime uses the ZoneRules, part of the java.time.zone package, to determine how an offset varies for a particular time zone. For example, most time zones experience a gap (typically of 1 hour) when moving the clock forward to daylight saving time, and a time overlap when moving the clock back to standard time and the last hour before the transition is repeated. The ZonedDateTime class accommodates this scenario, whereas the OffsetDateTime and OffsetTime classes, which do not have access to the ZoneRules, do not.

The ZonedDateTime Class

The ZonedDateTime class, in effect, combines the LocalDateTime class with the ZoneId class. It is used to represent a full date (year, month, day) and time (hour, minute, second, nanosecond) with a time zone (region/city, such as Europe/Paris).

The following code, efines the departure time for a flight from San Francisco to Tokyo as a ZonedDateTime in the America/Los Angeles time zone. The withZoneSameInstant() and plusMinutes() methods are used to create an instance of ZonedDateTime that represents the projected arrival time in Tokyo, after the 650 minute flight. The ZoneRules.isDaylightSavings() method determines whether it is daylight saving time when the flight arrives in Tokyo.

A DateTimeFormatter object is used to format the ZonedDateTime instances for printing:

```
DateTimeFormatter format = DateTimeFormatter.ofPattern("MMM d yyyy  hh:mm a");

// Leaving from San Francisco on July 20, 2013, at 7:30 p.m.
LocalDateTime leaving = LocalDateTime.of(2013, Month.JULY, 20, 19, 30);
ZoneId leavingZone = ZoneId.of("America/Los_Angeles");
ZonedDateTime departure = ZonedDateTime.of(leaving, leavingZone);

try {
    String out1 = departure.format(format);
    System.out.printf("LEAVING:  %s (%s)%n", out1, leavingZone);
} catch (DateTimeException exc) {
    System.out.printf("%s can't be formatted!%n", departure);
    throw exc;
}

// Flight is 10 hours and 50 minutes, or 650 minutes
ZoneId arrivingZone = ZoneId.of("Asia/Tokyo");
ZonedDateTime arrival = departure.withZoneSameInstant(arrivingZone)
                                 .plusMinutes(650);

try {
    String out2 = arrival.format(format);
    System.out.printf("ARRIVING: %s (%s)%n", out2, arrivingZone);
} catch (DateTimeException exc) {
    System.out.printf("%s can't be formatted!%n", arrival);
    throw exc;
}

if (arrivingZone.getRules().isDaylightSavings(arrival.toInstant())){
        System.out.printf("  (%s daylight saving time will be in effect.)%n",
        arrivingZone);
} else{
        System.out.printf("  (%s standard time will be in effect.)%n",
        arrivingZone);
}
```

This produces the following output:

```
LEAVING:  Jul 20 2013  07:30 PM (America/Los_Angeles)
ARRIVING: Jul 21 2013  10:20 PM (Asia/Tokyo)
  (Asia/Tokyo standard time will be in effect.)
```

The OffsetDateTime Class

The `OffsetDateTime` class, in effect, combines the `LocalDateTime` class with the `ZoneOffset` class. It is used to represent a full date (year, month, day) and time (hour, minute, second, nanosecond) with an offset from Greenwich/UTC time (+/-hours:minutes, such as +06:00 or -08:00).

The following example uses `OffsetDateTime` with the `TemporalAdjusters.lastInMonth()` method to find the last Thursday in July 2013.

```java
// Find the last Thursday in July 2013.
LocalDateTime localDate = LocalDateTime.of(2013, Month.JULY, 20, 19, 30);
ZoneOffset offset = ZoneOffset.of("-08:00");

OffsetDateTime offsetDate = OffsetDateTime.of(localDate, offset);
OffsetDateTime lastThursday =
        offsetDate.with(TemporalAdjusters.lastInMonth(DayOfWeek.THURSDAY));
System.out.printf("The last Thursday in July 2013 is the %sth.%n",
                   lastThursday.getDayOfMonth());
```

The output from running this code is:

```
The last Thursday in July 2013 is the 25th.
```

The OffsetTime Class

The `OffsetTime` class, in effect, combines the `LocalTime` class with the `ZoneOffset` class. It is used to represent time (hour, minute, second, nanosecond) with an offset from Greenwich/UTC time (+/-hours:minutes, such as +06:00 or -08:00).

The `OffsetTime` class is used in the same situations as the `OffsetDateTime` class, but when tracking the date is not needed.

***Last update:** January 27, 2022*

Date and Time ← **Time Zone and Offset** → Instant

Home > Tutorials > The Date Time API > Time Zone and Offset