# 7-date-time-essentials

## Contents

  - Zones
    * ZoneId
    * ZoneOffset
    * ZonedDateTime
    * Daylight savings
  - Formatting
- Summary

# Preface

`UTC (Coordinated Universal Time)is closely aligned with Greenwich Mean Time (GMT)in terms of timekeeping. Both are based on the time at the Prime Meridian (0ø longitude), which runs through Greenwich, London.`

The new Java date and time API is provided in the java.time package. This new API in Java 8 replaces the older classes supporting date and time related functionality such as the `Date`, `Calendar` and `TimeZone` classes provided as part of the `java.util.package`

Why did Java 8 introduce a new date and time API when it already had classes such as Date and Calendar from the early days of Java 8 ? The main reason was inconvenient API design. For example the Date class has both date and time components; if one wants to use only time information and not date-related information, then the date-related values have to be set to zero. Some aspects of the classes are unintuitive as well. For example in the Date constructor, the range of date values is 1 to 31 but the range of month values is 0 to 11 (not 1 to 12) Further, there are many concurrency-related issues with `java.util.Date` and `SimpleDateFormatter` because they are not thread-safe.

Java 8 provides very good support for date and time related functionality in the newly introduced java.time package, Most of the classes in this package are immutable and thread-safe. This chapter explains how to use important classes in this package and interfaces. Including the `LocalDate` and `LocalTime`, `LocalDateTime`, `Instant`, `Period` and `Duration` as well as `TemporalUnit`. The java.time API incorporates the concept of fluent interfaces it is designed in such a way that the code is more readable and easier to use. For this reason classes in this package have numerous static methods (many of them factory methods). In addition the methods in the classes follow a common naming convention (for example they use the prefixes plus and minus to add or subtract date or time values).

There are several important interfaces in the new java.time package, which are base interfaces to pretty much all the other date and time related implementations and classes.

## TemporalAccessor

A read only interface for accessing temporal information such as fields or units from a temporal object. Focuses solely on querying with no modification capabilities, acts as a base interface for Temporal.

## Temporal

This interface is meant to represent a point in time or an object in the date-time API that can be queries and adjusted, it does provide API methods such as plus,minus,with and get. The implementations of this class include `Instant, LocalDate, LocalTime, ZonedDateTime`

## TemporalAmount

Represents a relative amount of time such as a duration or a period, which ca nbe added to or subtracted from a Temporal, some implementation classes include `Duration and Period`.

## TemporalAdjuster

Represents a strategy for adjusting a Temporal object, allows performing custom or pre defined adjustments such as setting the date to the next Monday or the first day of the month. Built-in examples the `TemporalAdjusters` utility class provides common implementations like next(), firstDayOfMonth() etc

## TemporalUnit

The `TemporalUnit` interface is part of the temporal package. It represents date or time units such as seconds, hours, days, months, years, and so on. The enumeration `ChronoUnit` implements this interface. Instead of using constant values it is better to sue their equivalent enumeration values, this is due to the fact that the enumeration values in `ChronoUnit` results in more readable code, further it is less likely to make some logical mistakes in the implementation

| ChronoUnit | DateBased | TimeBased | Duration |
|---|---|---|---|
| Nanos | false | true | PT0.000000001S |
| Micros | false | true | PT0.000001S |
| Millis | false | true | PT0.001S |
| Seconds | false | true | PT1S |
| Minutes | false | true | PT1M |
| Hours | false | true | PT1H |
| HalfDays | false | true | PT12H |
| Days | true | false | PT24H |
| Weeks | true | false | PT168H |
| Months | true | false | PT730H29M6S |
| Years | true | false | PT8765H49M12S |
| Decades | true | false | PT87658H12M |
| Centuries | true | false | PT876582H |
| Millennia | true | false | PT8765820H |
| Eras | true | false | PT8765820000000H |
| Forever | false | false | PT2562047788015215H30M7.999999999S |

| Interface | Method | Description |
|---|---|---|
| **Temporal** | plus(long amountToAdd, TemporalUnit unit) | Adds an amount of time to this temporal object. |
| | minus(long amountToSubtract, TemporalUnit unit) | Subtracts an amount of time from this temporal object. |
| | with(TemporalField field, long newValue) | Returns a copy with the specified field set to a new value. |
| | get(TemporalField field) | Retrieves the value of the specified field as an int. |
| | isSupported(TemporalField field) | Checks if a field is supported by this temporal object. |

| Interface | Method | Description |
| --- | --- | --- |
| | until(Temporal endExclusive, TemporalUnit unit) | Calculates the amount of time until another temporal object. |
| **TemporalAccessor** | get(TemporalField field) | Gets the value of the specified field as an int. |
| | getLong(TemporalField field) | Gets the value of the specified field as a long. |
| | isSupported(TemporalField field) | Checks if the specified field is supported. |
| | query(TemporalQuery query) | Queries this temporal object using a specified query strategy. |
| **TemporalAdjuster** | firstDayOfMonth() | |
| | lastDayOfMonth() | |
| | firstDayOfNextMonth() | |
| | firstDayOfYear() | |
| | lastDayOfYear() | |
| | firstDayOfNextYear() | |
| | firstInMonth(DayOfWeek) | |
| **TemporalAmount** | addTo(Temporal temporal) | Adds this amount of time to the specified temporal object. |
| | subtractFrom(Temporal temporal) | Subtracts this amount of time from the specified temporal object. |
| | getUnits() | Returns the units of this temporal amount. |
| | get(TemporalUnit unit) | Gets the value of the specified unit. |
| **TemporalField** | getFrom(TemporalAccessor temporal) | Retrieves the value of this field from the specified temporal object. |
| | isSupportedBy(TemporalAccessor temporal) | Checks if the field is supported by the specified temporal object. |
| | rangeRefinedBy(TemporalAccessor temporal) | Gets the range of valid values for this field in the specified temporal. |
| | adjustInto(Temporal temporal, long newValue) | Adjusts the specified temporal object with a new value for this field. |
| **TemporalUnit** | addTo(Temporal temporal, long amount) | Adds the specified amount of this unit to the given temporal object. |
| | between(Temporal temporal1Inclusive, Temporal temporal2Exclusive) | Calculates the amount of time between two temporal objects in this unit. |
| | isSupportedBy(Temporal temporal) | Checks if this unit is supported by the specified temporal object. |
| | getDuration() | Gets the duration of this unit as a Duration. |

```
// because the ChronoUnit enumeration implements the TemporalUnit interface,
   you can pass ChronoUnit enumaration value
// as the second argument in this constructor
```

```
System.out.println(Duration.of(1, ChronoUnit.MINUTES).getSeconds());
System.out.println(Duration.of(1, ChronoUnit.HOURS).getSeconds());
System.out.println(Duration.of(1, ChronoUnit.DAYS).getSeconds());
```

## LocaDate

The `LocalDate` class represents a date without a time component. `LocalDate` is represented in the ISO-86011 calendar system in a year-month-day format. The java 8 and time API uses the ISO 8601 standard as the default calendar format. In this internationally accepted format the date and time values are sorted from the largest to the smallest unit of time - year, month day, hour, minute second and millisecond

```
LocalDate today = LocalDate.now();
System.out.println("Today's date is: " + today);
```

The `LocalDate.now()` method gets the current date using the system clock, based on the default time zone of the JVM. You can get a `LocalDate` object by explicitly specifying the day, month and year components separately too

```
LocalDate newYear2016 = LocalDate.of(2016, 1, 1);
System.out.plrintln("New year 2016: " + newYear2016);
```

The input arguments to the local date of method are also properly validated as well, one can not simply pass just any value to those methods, for example passing an invalid year, month, or day would throw an exception, in this case a `DateTimeException`

```
// this however will throw an exception, due to the fact that the month and
   day are actually inverted as input arguments
// to this method, the 14, which is supposed to be the day, is actually
   passed to the second argument which is the month
// not the day, and the month is passed in as the third argument, which is
   actually the day
LocalDate valentinesDay = LocalDate.of(2016, 14, 2 );
System.out.println("Valentine's day is on: ", valentinesDay);
```

To avoid making this mistake, you can use the overloaded version of `LocalDate.of(int year, Month month , int day)`. The second argument being of type Month, is an enumeration that represents the 12 months of the year. In that case it is not possible to interchange the arguments, otherwise the code will not compile.

```
LocalDate valentinesDay = LocalDate.of(2016, Month.FEBRUARY, 14);
System.out.println("Valentine's day is on: " + valentinesDay);
```

The `LocalDate` class has methods with which you can add or subtract days, weeks months or years to or from the current `LocalDate` object. For example suppose your visa expires 180 days from now. Here is code segment that shows the expiry date.

```
long visaValidityDays = 180L;
LocalDate currDate = LocalDate.now();
System.out.println("Expires on: " + currDate.plusDays(visaValidityDays));
```

In addition to the `plusDays()` method, `LocalDate` also provides `plusWeeks()`, `plusMonths()`, and `plusYears()` methods as well as methods for subtracting the same - `minusDays()`, `minusWeeks()`, `minusMonths()` and `minusYears()`.

## LocalTime

The `java.time.LocalTime` class is similar to the `LocalDate` except that LocalTime represents time without dates or any time zones. The time is in ISO-8601 as well, and the format is HH:MM:SS.nanoseconds. Both `LocalTime` and `LocalDate` use the system clock and the default time zone.

```
// the statement below can print something along the lines 12:23:05.072
LocalTime currTime = LocalTime.now();
System.out.println("Current time is: " + currTime);
```

As mentioned, `LocalTime` uses the system clock and its default time zone. To create different time objects based on the specific time values, you can use the overloaded of() method of the `LocalTime` class:

`LocalTime` provides many useful methods with which you can add or subtract hours, minutes, seconds and nanoseconds. For example, assume that you have a meeting in 6.5 hours from `now`, and you would like to find the exact meeting, or in other words the absolute meeting time. Here is how that can be achieved

```
long hours = 6;
long minutes = 30;
LocalTime currTime = LocalTime.now();
System.out.println("Current time is: " + currTime);
System.out.println("Meeting is at : " + currTime.plusHours(hours).plusMinutes
   (minutes));
```

In addition to the methods shown above, like `plusHours` and `plusMinutes`, the `LocalTime` supports, `plusSeconds`, and `plusNanos`. The equivalent for subtracting also exist, such as `minusHours`, `minusMinutes`, `minusSeconds` and `minusNanos`.

## LocalDateTime

The `LocalDateTime` represents both date and time without time zones. That class can be thought as a logical combination of the `LocalTime` and `LocalDate` classes. The date and time formats both are in the already mentioned ISO-8601 calendar format: `YYYY-MM-DD HH:MM:SS.nanoseconds`

The general `toString` format output for a given `LocalDateTime` can look something like the following: `2015-10-29T21:04:36.376`. In this output note that the character T stands for time, and it separates the date and time components of this `LocalDateTime` instance, it is also used to parse `LocalDateTime` instances from strings as well, that is part of the ISO-8601 format standard.

Similar to both the `LocalDate` and `LocalTime` classes the `LocalDateTime` also provides means of subtracting and adding temporal elements to a given `LocalDateTime` instance, it does support the same types of methods such as - like `plusHours` and `plusMinutes`, the `LocalTime` supports, `plusSeconds`, and `plusNanos`. The equivalent for subtracting also exist, such as `minusHours`, `minusMinutes`, `minusSeconds` and `minusNanos`

## Miscellaneous

There are several more methods which are shared, semantically between the different types of dates, methods such as

### Conversions

```
LocalDateTime dateTime = LocalDateTime.now();
System.out.println("Today's date and current time: " + dateTime);
System.out.println("The date component is " + dateTime.toLocalDate());
System.out.println("The date component is " + dateTime.toLocalTime());
```

The code above demonstrates how a `LocalDateTime` can be split into its constituent elements, such that the `LocalDate` and `LocalTime` can be extracted separately, from it.

The whole package including the `LocalDate` and `LocalTime` and `LocalDateTime` classes, along with Instant, Period, Duration and so on, provide means of converting from one type to another, where reasonable.

### isAfter & isBefore

These methods compare if a given instance of the class of the `Local*` type is after/before another one. What is important to note here is that those methods are non-inclusive, meaning that calling `isAfter` on two equal dates, or times, or date-time instances will yield false, for both `isAfter` or `isBefore`, they are not taking into account when the two instances are equal. To compare for equality one has to use the equals method instead

## Instant

This type of class deals with time and date values, and the instant values begin on `01.1970` at `00:00:00` hours, known as the UNIX epoch. The Instant class internally uses a long variable that holds the number of seconds since the start of the UNIX epoch, times that start before this epoch are treated as negative values. Instant uses an integer variable to store the number of nanoseconds elapsed for each second. The `Instant` class is meant to deal with both date and time components, it is semantically equivalent to the `LocalDateTime`. However internally the way the information is stored is not how `LocalDateTime` stores it, as already mentioned above, the instant stores absolute date and time in relation to UTC or the UNIX epoch.

```
public static void main(String args[]){
    // prints the current timestamp with UTC as time zone
    Instant currTimeStamp = Instant.now();
    System.out.println("Instant timestamp is: " + currTimeStamp);
    // prints the number of seconds as Unix timestamp from epoch time
    System.out.println("Number of seconds elapsed: " + currTimeStamp.
       getEpochSecond());
    // prints the Unix timestamp in milliseconds
    System.out.println("Number of milliseconds elapsed: " + currTimeStamp.
       toEpochMilli());
}
```

When executed the above can print something along the lines of what is shown below, now notice that the format of Instant is pretty much identical to the one for LocalDateTime

```
Instant timestamp is: 2015-11-02T03:16:04.502Z
Number of seconds elapsed: 1446434164
Number of milliseconds elapsed: 1446434164502
```

So what is the difference between `Instant and LocalDateTime`. Well the example below shows that the LocalDateTime is based on the current system or JVM the time zone, along with the actual time and date components, however the Instant is an absolute value, which starts as mentioned from the UNIX epoch.

```
LocalDateTime localDateTime = LocalDateTime.now();
Instant instant = Instant.now();
System.out.println("LocalDateTime is: " + localDateTime + " \nInstant is: " +
    instant);
```

The snippet above might print something along the lines of the example below, where it is clear that the Instant is not affected by the current system or user time zones. As mentioned the Instant is dealing with absolute date & time values as compared to the `LocalDateTime` class type, and that starts from the UNIX epoch, so no matter on which machine the Instant is generated it will always print the absolute value regardless of the system time zone.

```
LocalDateTime is: 2015-11-02T17:21:11.402
Instant is: 2015-11-02T11:51:11.404
```

## Period

The Period type is meant to deal with amounts of time in terms of years, months and days. It is semantically equivalent to the `LocalDate` class. However internally the way the information is stored about is very much different than the one in `LocalDate`, that is due to the requirements for what the Period is, even though when printed out with `toString` it might look like exactly `LocalDate toString` method format, the period stores absolute period of time from the UNIX epoch.

Similarly to the other classes in this package one can subtract years months and days using the methods `plusYears`, `plusMonths plusDays`, `minusYears minusMonths` and `minusDays`.

## Duration

As discussed the Period class earlier, represents time in terms of years, months, and days. Duration is the time equivalent of Period. The Duration class represents time in terms of hours, minutes, seconds and so on. It is suitable for measuring machine time or when working with Instance objects, Similar to the Instance class, the Duration class stores the seconds component as a long value and nanoseconds using an int value.

```
LocaslDateeTime comingMidnight = LocalDateTime.of(LocalDate.now().plusDays(1)
   , LocalTime.MIDNIGHT);
LocalDateTime now = LocalDateTime.now();
Duration between = Duration.between(now, comingMidnight);
System.out.println(between);
```

The snippet above might print something like `PT7H13M42.003S`. This example uses the overloaded version of the of method in the `LocalDateTime` class, which is building `LocalDateTime` from a `LocalDate.now` combined with the time component of MIDNIGHT, which produces the final `LocalDateTime`, on the next day, in midnight

```
The Java 8 date and time API differentiates how humans and computers use date and time
 related information. For example, the Instant class represents a Unix timestamp and
internally uses long and int variables. Instant values are not very readable or usable by
humans because the class does not support methods related to day, month, hours and so on,
in contrast the Period class supports such methods.
```

## Zones

There are three important classes related to time zones that one needs to be aware of, in order to work with dates and times across time zones: `ZoneId, ZoneOffset and ZonedDateTime`.

### ZoneId

This class represents time zones. Time zones are typically identified using and offset from Greenwich Mean Time (GMT, also known as UTC/Greenwich). For instance given the example of the time zone Europe/Helsinki, one can print out the current system time zone by simply doing

```
// that would print out the current system time zone, based on what is
   configured by the user, system or even the JVM
System.out.println("My zone id is: " + ZoneId.systemDefault());
```

To obtain the list of all time zones by calling the static method `getAvailableZoneIds` in `ZoneId`, which returns `Set<String>`. The snippet below shows how to obtain the list of all zones.

```
// the snippet below would print something like the following
// Number of available time zones is: 589
// Asia/Aden
// America/Cuiaba
// .............
Set<String> zones = ZoneId.getAvailableZoneIds();
System.out.println("Numer of avaiilable time zones is: " + zones.size());
zones.forEach(System.out::println)
```

To parse or re-build a `ZoneId` from a String zone representation, one can use the `of` method, which can construct a `ZoneId` instance from a String, the format of the String has to match the time zone format shown above (region)/(zone)

```
// this will construct an instance of ZoneId, from the string representation
   of the zone, this representation format is
// specified in the ISO-8601
ZoneId europe = ZoneId.of("Europe/Helsinki");
```

### ZoneOffset

`ZoneId` identifies a time zone, such as the example shown above, another companion class is the `ZoneOffset`, which represents the time-zone offset from UTC/Greenwich. Each different time zone has a different time offset compared to the UTC one.

### ZonedDateTime

In Java if one desires to deal with date and time or time zone, it is better to use the `LocalDate` or `LocalTime` or `ZoneId` respectively. What if one wants all three date, time and time-zone. For that this is where the `ZonedDateTime` comes in

```
LocalDate currentDate = LocalDate.now();
LocalTime currentTime = LocalTime.now();
ZoneId zone = ZoneId.systemDefault();
ZonedDateTime zonedDateTime = ZonedDateTime.of(currentDate, currentTime, zone
   );
System.out.println(zonedDateTime);
```

This code segment uses the overloaded static method of `ZonedDateTime - of(LocalDate, LocalTime, ZoneID)`. Given a `LocalDateTime` one can use a `ZoneId` to get a `ZonedDateTime`. But it is also possible to convert between the `LocalDateTime` and `ZonedDateTime`

```
LocalDateTime dateTime = LocalDateTime.now();
ZoneId myZone = ZoneId.systemDefault();
ZonedDateTime zonedDateTime = dateTime.atZone(myZone);
```

### Daylight savings

The amount of daylight does not remain the same throughout the year because the seasons change. There is more daylight in the summer than in the winter. With daylight savings time (DST) the clock is set one hour earlier or later to make the best use of the daylight. As the saying goes "Spring forward, fall back" - the clock is typically set on hour earlier when Spring begins and one hour later at the start of the Fall:

```
ZoneId kolkataZone = ZoneId.of("Asia/Kolkata");
Duration kolkataDST = kolkataZone.getRules().getDaylightSavings(Instant.now()
   );
System.out.printf("Kolkata zone DST is: %d hours %n", kolkataDST.toHours());

ZoneId aucklandZone = ZoneId.of("Pacific/Auckland");
Duration aucklandDST = aucklandZone.getRules().getDaylightSavings(Instant.now
   ());
System.out.printf("Auckland zone DST is: %d hours", aucklandDST.toHours());
```

The call to `getDaylightSavings` will make sure to print correctly if the given zone is within a daylight saving regime, so given the snippet above one can expect to see that since the First time zone is in the northern hemisphere where during November it is winter, and the other zone is in the southern hemisphere November is in the summer.

- if the duration is zero - DST it not in effect in that zone;
- if the duration is non-zero - DST is in effect in that zone;

```
Here is the result (when executed on November 5):
Kolkata zone DST is: 0 hours
Auckland zone DST is: 1 hours
```

## Formatting

When programming with dates and times one often desires to print them in a different formats that the ones specified by default from ISO-8601. Also one may have to read date time information given in different formats. To read or print date and time values in various formats one can use the `DateTimeFormatter` class from the java.time.format package. The `DateTimeFormatter` class provides many predefined constants for formatting date and time values. Here is a list of a few such predefined formatters

```
ISO_DATE (2015-11-05)
ISO_TIME (11:25:47.624)
RFC_1123_DATE_TIME (Thu, 5 Nov 2015 11:27:22 +0530)
ISO_ZONED_DATE_TIME (2015-11-05T11:30:33.49+05:30[Asia/Kolkata])
```

To use these formatting constants, which are really simply an instance of the `DateTimeFormatter` itself, simply call the format method of the `DateTimeFormatter` with an argument, the argument to format is of type `TemporalAccessor` which is the base most interface, from which pretty much all entries in java.time implement from

```
LocalTime wakeupTime = LocalTime.of(6, 0, 0);
System.out.println("Wake up time: " + DateTimeFormatter.ISO_TIME.format(
   wakeupTime));
```

```
Note that in the format string, the upper and lower case letters might have different
meanings, for example upper case M means month, however to express minutes one has to use
 lower case m, this is especially true, when the format is formatting a class type with
date and time components such as LocalDateTime, and or ZonedDateTime
```

Here is a list of important letters and their meanings for creating patterns for dates

- G (era: BC, AD)
- y (year of era: 2015, 15)
- Y (week-based year: 2015, 15)
- M (month: 11, Nov, November)
- w (week in year: 13)
- W (week in month: 2)
- E (day name in week: Sun, Sunday)
- D (day of year: 256)
- d (day of month: 13)
- a (marker for the text a.m./p.m. marker)
- H (hour: value range 0-23)
- k (hour: value range 1-24)
- K (hour in a.m./p.m.: value range 0-11)
- h (hour in a.m./p.m.: value range 1-12)
- m (minute
- s (second)
- S (fraction of a second)
- z (time zone: general time-zone format)

```
// the example simply shows how to create a new formatter for a given format
   string, note that the methods convention
// here is the same as for the other classes in java.time, meaning that it
   follows the `of` pattern of method names, which
// are static and produce an instance of the given class type, instead of
   relying on constructors, to be called by the
// client code, which is less robust to future changes and more prone.
String dateTimeFormat = "dd-mm-yyy '('E')'";
DateTimeFormatter.ofPattern(dateTimeFormat);
```

```
Note that if one wishes to print raw text inside the pattern, the text has to be surrounded
 by single quotes, for example having the following text inside the pattern '('E')', will
 print the day of the week which is the E, surrounded by plain brackets, which are not
going to be interpreted as part of the pattern symbols internally by the formatter, in the
 end the final formatted string will look like that - 01.01.2000 (Wed)
```

## Summary

Create and manage dates

- The Java 8 date and time API uses ISO 8601 as the default calendar format, deprecating the old Calendar API
- The `LocalDate` class represents a date without time or time zones; the `LocalTime` class represents time without dates and time zones; the `LocalDateTime` class represents both date and time without time zones.
- The Instant class represents a Unix timestamp.
- The Period is used to measure the amount of time in terms of years, months, and days.
- The Duration class represents time in terms of hours, minutes, seconds, and fraction of seconds.

- The enumeration temporal.`ChronoUnit` implements the `TemporalUnit` interface.
- Both `TemporalUnit` and `ChronoUnit` deal with time unit values such as seconds, minutes, and hours and date values such as days, months, and years.

Create and manage dates with zones

- `ZoneId` identifies a time zone; `ZoneOffset` represents time zone offset from UTC/Greenwich.
- `ZonedDateTime` provides support for all three aspects: date, time, and time zone.
- You have to account for daylight savings time (DST) when working with different time zones.

Format date and time components

- The `DateTimeFormatter` class provides support for reading or printing date and time values in different formats.
- The `DateTimeFormatter` class provides predefined constants (such as ISO_DATE and ISO_TIME) for formatting date and time values.
- You encode the format of the date or time using case-sensitive letters to form a date or time pattern string with the `DateTimeFormatter` class.