Introduction to the Java 8 Date/Time API

Last modified: August 31, 2017

https://www.mkyong.com/java8/java-8-how-to-format-localdatetime/
https://www.mkyong.com/java/how-to-convert-string-to-date-java/

by baeldung

Java +

```
//DateTimeFormatter formatter = DateTimeFormatter.ofPattern("d/MM/yyyy");
//String date = "16/08/2016";
//convert String to LocalDate
//LocalDate localDate = LocalDate.parse(date, formatter);
```

1. Overview

Java 8 introduced new APIs for *Date* and *Time* to address the shortcomings of the older *java.util.Date* and *java.util.Calendar*.

As part of this article, let's start with the issues in the existing *Date* and *Calendar* APIs and let's discuss how the new Java 8 *Date* and *Time* APIs address them.

We will also look at some of the core classes of the new Java 8 project that are part of the *java.time* package like *LocalDate*, *LocalTime*, *LocalDateTime*, *ZonedDateTime*, *Period*, *Duration* and their supported APIs.

2. Issues with the Existing *Date/ Time* APIs

- **Thread Safety** – The *Date* and *Calendar* classes are not thread safe, leaving developers to deal with the headache of hard to debug concurrency issues and to write additional code to handle thread safety. On the contrary the new *Date* and *Time* APIs introduced in Java 8 are immutable and thread safe, thus taking that concurrency headache away from developers.
- **APIs Design and Ease of Understanding** – The *Date* and *Calendar* APIs are poorly designed with inadequate methods to perform day-to-day operations. The new *Date/Time* APIs is ISO centric and follows consistent

domain models for date, time, duration and periods. There are a wide variety of utility methods that support the commonest operations.
- ***ZonedDate* and *Time*** – Developers had to write additional logic to handle timezone logic with the old APIs, whereas with the new APIs, handling of timezone can be done with *Local* and *ZonedDate/ Time* APIs.

3. Using *LocalDate*, *LocalTime* and *LocalDateTime*

The most commonly used classes are *LocalDate*, *LocalTime* and *LocalDateTime*. As their names indicate, they represent the local Date/Time from the context of the observer.

These classes are mainly used when timezone are not required to be explicitly specified in the context. As part of this section, we will cover the most commonly used APIs.

3.1. Working with *LocalDate*

The *LocalDate* represents **a date in ISO format (yyyy-MM-dd) without time**.

It can be used to store dates like birthdays and paydays.

An instance of current date can be created from the system clock as below:

```
1 LocalDate localDate = LocalDate.now();
```

The *LocalDate* representing a specific day, month and year can be obtained using the "*of*" method or by using the "*parse*" method. For example the below code snippets represents the *LocalDate* for 20 February 2015:

```
LocalDate.of(2015, 02, 20);

LocalDate.parse("2015-02-20");
```

The *LocalDate* provides various utility methods to obtain a variety of information. Let's have a quick peek at some of these APIs methods.

The following code snippet gets the current local date and adds one day:

```
LocalDate tomorrow = LocalDate.now().plusDays(1);
```

This example obtains the current date and subtracts one month. Note how it accepts an *enum* as the time unit:

```
LocalDate previousMonthSameDay = LocalDate.now().minus(1, ChronoUnit.MONTHS);
```

In the following two code examples we parse the date "2016-06-12" and get the day of the week and the day of the month respectively. Note the return values, the first is an object representing the *DayOfWeek* while the second in an *int* representing the ordinal value of the month:

```
DayOfWeek sunday = LocalDate.parse("2016-06-12").getDayOfWeek();

int twelve = LocalDate.parse("2016-06-12").getDayOfMonth();
```

We can test if a date occurs in a leap year. In this example we test if the current date occurs is a leap year:

```
boolean leapYear = LocalDate.now().isLeapYear();
```

The relationship of a date to another can be determined to occur before or after another date:

```
boolean notBefore = LocalDate.parse("2016-06-12")
  .isBefore(LocalDate.parse("2016-06-11"));

boolean isAfter = LocalDate.parse("2016-06-12").isAfter(LocalDate.parse("2016
```

Date boundaries can be obtained from a given date. In the following two examples we get the *LocalDateTime* that represents the beginning of the day (2016-06-12T00:00) of the given date and the *LocalDate* that represents the beginning of the month (2016-06-01) respectively:

```
LocalDateTime beginningOfDay = LocalDate.parse("2016-06-12").atStartOfDay();
LocalDate firstDayOfMonth = LocalDate.parse("2016-06-12")
  .with(TemporalAdjusters.firstDayOfMonth());
```

Now let's have a look at how we work with local time.

3.2. Working with *LocalTime*

The *LocalTime* represents **time without a date**.

Similar to *LocalDate* an instance of *LocalTime* can be created from system clock or by using "parse" and "of" method. Quick look at some of the commonly used APIs below.

An instance of current *LocalTime* can be created from the system clock as below:

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

In the below code sample, we create a *LocalTime* representing 06:30 AM by parsing a string representation:

```
1 LocalTime sixThirty = LocalTime.parse("06:30");
```

The Factory method "of" can be used to create a *LocalTime*. For example the below code creates *LocalTime* representing 06:30 AM using the factory method:

```
1 LocalTime sixThirty = LocalTime.of(6, 30);
```

The below example creates a *LocalTime* by parsing a string and adds an hour to it by using the "plus" API. The result would be *LocalTime* representing 07:30 AM:

```
1 LocalTime sevenThirty = LocalTime.parse("06:30").plus(1, ChronoUnit.HOURS);
```

Various getter methods are available which can be used to get specific units of time like hour, min and secs like below:

```
int six = LocalTime.parse("06:30").getHour();
```

We can also check if a specific time is before or after another specific time. The below code sample compares two *LocalTime* for which the result would be true:

```
boolean isbefore = LocalTime.parse("06:30").isBefore(LocalTime.parse("07:30")
```

The max, min and noon time of a day can be obtained by constants in *LocalTime* class. This is very useful when performing database queries to find records within a given span of time. For example, the below code represents 23:59:59.99:

```
LocalTime maxTime = LocalTime.MAX
```

Now let's dive into *LocalDateTime*.

3.3. Working with *LocalDateTime*

The *LocalDateTime* is used to represent **a combination of date and time**.

This is the most commonly used class when we need a combination of date and time. The class offers a variety of APIs and we will look at some of the most commonly used ones.

An instance of *LocalDateTime* can be obtained from the system clock similar to *LocalDate* and *LocalTime:*

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

The below code samples explain how to create an instance using the factory "of" and "parse" methods. The result would be a *LocalDateTime* instance representing 20 February 2015, 06:30 AM:

```
LocalDateTime.of(2015, Month.FEBRUARY, 20, 06, 30);
```

```
LocalDateTime.parse("2015-02-20T06:30:00");
```

There are utility APIs to support addition and subtraction of specific units of time like days, months, year and minutes are available. The below code samples demonstrates the usage of "plus" and "minus" methods. These APIs behave exactly like their counterparts in *LocalDate* and *LocalTime:*

```
localDateTime.plusDays(1);
```

```
localDateTime.minusHours(2);
```

Getter methods are available to extract specific units similar to the date and time classes. Given the above instance of *LocalDateTime*, the below code sample will return the month February:

```
localDateTime.getMonth();
```

4. Using *ZonedDateTime* API

Java 8 provides *ZonedDateTime* when we need to deal with time zone specific date and time. The *ZoneId* is an identifier used to represent different zones. There are about 40 different time zones and the *ZoneId* are used to represent them as follows.

In this code snippet we create a *Zone* for Paris:

```
1 ZoneId zoneId = ZoneId.of("Europe/Paris");
```

A set of all zone ids can be obtained as below:

```
1 Set<String> allZoneIds = ZoneId.getAvailableZoneIds();
```

The *LocalDateTime* can be converted to a specific zone:

```
1 ZonedDateTime zonedDateTime = ZonedDateTime.of(localDateTime, zoneId);
```

The *ZonedDateTime* provides *parse* method to get time zone specific date time:

```
1 ZonedDateTime.parse("2015-05-03T10:15:30+01:00[Europe/Paris]");
```

Another way to work with time zone is by using *OffsetDateTime*.
The *OffsetDateTime* is an immutable representation of a date-time with an offset. This class stores all date and time fields, to a precision of nanoseconds, as well as the offset from UTC/Greenwich.

The *OffSetDateTime* instance can be created as below using *ZoneOffset*. Here we create a *LocalDateTime* representing 6:30 am on 20th February

```
LocalDateTime localDateTime = LocalDateTime.of(2015, Month.FEBRUARY, 20, 06,
```

Then we add two hours to the time by creating a *ZoneOffset* and setting for the *localDateTime* instance:

```
ZoneOffset offset = ZoneOffset.of("+02:00");

OffsetDateTime offSetByTwo = OffsetDateTime
  .of(localDateTime, offset);
```

We now have a *localDateTime* of 2015-02-20 06:30 +02:00. Now let's move on to how to modify date and time values using the *Period* and *Duration* classes.

5. Using *Period* and *Duration*

The *Period* class represents a quantity of time in terms of years, months and days and the *Duration* class represents a quantity of time in terms of seconds and nano seconds.

5.1. Working with *Period*

The *Period* class is widely used to modify values of given a date or to obtain the difference between two dates:

```
LocalDate initialDate = LocalDate.parse("2007-05-10");
```

The *Date* can be manipulated using *Period* as shown in the following code snippet:

```
LocalDate finalDate = initialDate.plus(Period.ofDays(5));
```

The *Period* class has various getter methods such as *getYears*, *getMonths* and *getDays* to get values from a *Period* object. The below code example returns an *int* value of 5 as we try to get difference in terms of days:

```
int five = Period.between(finalDate, initialDate).getDays();
```

The *Period* between two dates can be obtained in a specific unit such as days or month or years, using *ChronoUnit.between:*

```
int five = ChronoUnit.DAYS.between(initialDate , initialDate);
```

This code example returns five days. Let's continue by taking a look at the *Duration* class.

5.2. Working with *Duration*

Similar to *Period*, the *Duration class* is use to deal with *Time*. In the following code we create a *LocalTime* of 6:30 am and then add a duration of 30 seconds to make a *LocalTime* of 06:30:30am:

```
LocalTime initialTime = LocalTime.of(6, 30, 0);

LocalTime finalTime = initialTime.plus(Duration.ofSeconds(30));
```

The *Duration* between two instants can be obtained either as a *Duration* or as a specific unit. In the first code snippet we use the *between()* method of the *Duration* class to find the time difference between *finalTime* and *initialTime* and return the difference in seconds:

```
int thirty = Duration.between(finalTime, initialTime).getSeconds();
```

In the second example we use the *between()* method of the *ChronoUnit* class to perform the same operation:

```
int thirty = ChronoUnit.SECONDS.between(finalTime, initialTime);
```

Now we will look at how to convert existing *Date* and *Calendar* to new *Date/Time*.

6. Compatibility with *Date* and *Calendar*

Java 8 has added the *toInstant()* method which helps to convert existing *Date* and *Calendar* instance to new Date Time API as in the following code snippet:

```
LocalDateTime.ofInstant(date.toInstant(), ZoneId.systemDefault());
LocalDateTime.ofInstant(calendar.toInstant(), ZoneId.systemDefault());
```

The *LocalDateTime* can be constructed from epoch seconds as below. The result of the below code would be a *LocalDateTime* representing 2016-06-13T11:34:50:

```
LocalDateTime.ofEpochSecond(1465817690, 0, ZoneOffset.UTC);
```

Now let's move on to *Date* and *Time* formatting.

7. *Date* and *Time* Formatting

Java 8 provides APIs for the easy formatting of *Date* and *Time*:

```
LocalDateTime localDateTime = LocalDateTime.of(2015, Month.JANUARY, 25, 6, 30
```

The below code passes an ISO date format to format the local date. The result would be 2015-01-25 :

```
LocalDate localDate = localDateTime.format(DateTimeFormatter.ISO_DATE);
```

The *DateTimeFormatter* provides various standard formatting options. Custom patterns can be provided to format method as well, like below, which would return a *LocalDate* as 2015/01/25:

```
localDateTime.format(DateTimeFormatter.ofPattern("yyyy/MM/dd"));
```

We can pass in formatting style either as *SHORT*, *LONG* or *MEDIUM* as part of the formatting option. The below code sample would give an output representing *LocalDateTime* in 25-Jan-2015 06:30:00:

```
localDateTime
  .format(DateTimeFormatter.ofLocalizedDateTime(FormatSt
  .withLocale(Locale.UK);
```

Let us take a look at alternatives available to Java 8 Core *Date/ Time* APIs.

8. Backport and Alternate Options

8.1. Using Project Threeten

For organization that are on the path of moving to Java 8 from Java 7 or Java 6 and want to use date and time API, project threeten provides the backport capability. Developers can use classes available in this project to achieve the same functionality as that of new Java 8 *Date* and *Time* API and once they move to Java 8, the packages can be switched. Artifact for the project threeten can be found in the maven central repository:

```
<dependency>
    <groupId>org.threeten</groupId>
    <artifactId>threetenbp</artifactId>
    <version>1.3.1</version>
</dependency>
```

8.2. Joda-Time Library

Another alternative for Java 8 *Date* and *Time* library is Joda-Time library. In fact Java 8 *Date Time* APIs has been led jointly by the author of Joc
(Stephen Colebourne) and Oracle. This library provides pretty much all capabilities

that is supported in Java 8 *Date Time* project. The Artifact can be found in the maven central by including the below pom dependency in your project:

```
<dependency>
    <groupId>joda-time</groupId>
    <artifactId>joda-time</artifactId>
    <version>2.9.4</version>
</dependency>
```

9. Conclusion

Java 8 provides a rich set of APIs with consistent API design for easier development.

The code samples for the above article can be found in the Java 8 Date/Time git repository.

CATEGORIES

- SPRING
- REST
- JAVA
- SECURITY
- PERSISTENCE
- JACKSON
- HTTPCLIENT
- KOTLIN

SERIES

- JAVA "BACK TO BASICS" TUTORIAL
- JACKSON JSON TUTORIAL
- HTTPCLIENT 4 TUTORIAL
- REST WITH SPRING TUTORIAL
- SPRING PERSISTENCE TUTORIAL
- SECURITY WITH SPRING

ABOUT

- ABOUT BAELDUNG
- THE COURSES
- CONSULTING WORK
- META BAELDUNG
- THE FULL ARCHIVE
- WRITE FOR BAELDUNG
- CONTACT
- COMPANY INFO
- TERMS OF SERVICE
- PRIVACY POLICY
- EDITORS
- MEDIA KIT (PDF)