1)Find date for “ tenthFeb2017” using LacalDate

LocalDate tenthFeb2014 = LocalDate.of(2017, Month.FEBRUARY, 10);

2)Find sixtyfifth day of 2017 using LocalDate

LocalDate sixtyFifthDayOf2010 = LocalDate.ofYearDay(2017, 65);

3)Find 12345th second of the day using LocalTime

LocalTime fromSecondsOfDay = LocalTime.ofSecondOfDay(12345);

4)Find LocalDateTime object for given date and time 2014-02-20 12:00

LocalDateTime dateTime = LocalDateTime.of(2014, 02, 20, 12, 0);

5)Show that 5th Feb 2017 is before current date or not( Boolean value)

LocalDate today = LocalDate.*now*();

LocalDate fifthfeb2017 = LocalDate.*of*(2017, Month.***FEBRUARY***, 5);

Period p=Period.*between*(today,fifthfeb2017);

**if**(p.isNegative())

System.***out***.println("true");

**else**

System.***out***.println("false");

**A Deeper Look into the Java 8 Date and Time API**

Within this post we will have a deeper look into the new Date/Time API we get with Java 8 ([JSR 310](https://java.net/projects/jsr-310/)). Please note that this post is mainly driven by code examples that show the new API functionality. I think the examples are self-explanatory so I did not spent much time writing text around them :-)

**Working with Date and Time Objects**

All classes of the Java 8 Date/Time API are located within the java.time package. The first class we want to look at is java.time.LocalDate. A [LocalDate](http://download.java.net/jdk8/docs/api/java/time/LocalDate.html) represents a year-month-day date without time. We start with creating new LocalDate instances:

// the current date

LocalDate currentDate = LocalDate.now();

// 2014-02-10

LocalDate tenthFeb2014 = LocalDate.of(2014, Month.FEBRUARY, 10);

// months values start at 1 (2014-08-01)

LocalDate firstAug2014 = LocalDate.of(2014, 8, 1);

// the 65th day of 2010 (2010-03-06)

LocalDate sixtyFifthDayOf2010 = LocalDate.ofYearDay(2010, 65);

[LocalTime](http://download.java.net/jdk8/docs/api/java/time/LocalTime.html) and [LocalDateTime](http://download.java.net/jdk8/docs/api/java/time/LocalDateTime.html) are the next classes we look at. Both work similar to LocalDate. ALocalTime works with time (without dates) while LocalDateTime combines date and time in one class:

LocalTime currentTime = LocalTime.now(); // current time

LocalTime midday = LocalTime.of(12, 0); // 12:00

LocalTime afterMidday = LocalTime.of(13, 30, 15); // 13:30:15

// 12345th second of day (03:25:45)

LocalTime fromSecondsOfDay = LocalTime.ofSecondOfDay(12345);

// dates with times, e.g. 2014-02-18 19:08:37.950

LocalDateTime currentDateTime = LocalDateTime.now();

// 2014-10-02 12:30

LocalDateTime secondAug2014 = LocalDateTime.of(2014, 10, 2, 12, 30);

// 2014-12-24 12:00

LocalDateTime christmas2014 = LocalDateTime.of(2014, Month.DECEMBER, 24, 12, 0);

By default LocalDate/Time classes will use the system clock in the default time zone. We can change this by providing a time zone or an alternative [Clock](http://download.java.net/jdk8/docs/api/java/time/Clock.html) implementation:

// current (local) time in Los Angeles

LocalTime currentTimeInLosAngeles = LocalTime.now(ZoneId.of("America/Los\_Angeles"));

// current time in UTC time zone

LocalTime nowInUtc = LocalTime.now(Clock.systemUTC());

From LocalDate/Time objects we can get all sorts of useful information we might need. Some examples:

LocalDate date = LocalDate.of(2014, 2, 15); // 2014-06-15

boolean isBefore = LocalDate.now().isBefore(date); // false

// information about the month

Month february = date.getMonth(); // FEBRUARY

int februaryIntValue = february.getValue(); // 2

int minLength = february.minLength(); // 28

int maxLength = february.maxLength(); // 29

Month firstMonthOfQuarter = february.firstMonthOfQuarter(); // JANUARY

// information about the year

int year = date.getYear(); // 2014

int dayOfYear = date.getDayOfYear(); // 46

int lengthOfYear = date.lengthOfYear(); // 365

boolean isLeapYear = date.isLeapYear(); // false

DayOfWeek dayOfWeek = date.getDayOfWeek();

int dayOfWeekIntValue = dayOfWeek.getValue(); // 6

String dayOfWeekName = dayOfWeek.name(); // SATURDAY

int dayOfMonth = date.getDayOfMonth(); // 15

LocalDateTime startOfDay = date.atStartOfDay(); // 2014-02-15 00:00

// time information

LocalTime time = LocalTime.of(15, 30); // 15:30:00

int hour = time.getHour(); // 15

int second = time.getSecond(); // 0

int minute = time.getMinute(); // 30

int secondOfDay = time.toSecondOfDay(); // 55800

Some information can be obtained without providing a specific date. For example, we can use the [Year](http://download.java.net/jdk8/docs/api/java/time/Year.html) class if we need information about a specific year:

Year currentYear = Year.now();

Year twoThousand = Year.of(2000);

boolean isLeap = currentYear.isLeap(); // false

int length = currentYear.length(); // 365

// sixtyFourth day of 2014 (2014-03-05)

LocalDate date = Year.of(2014).atDay(64);

We can use the plus and minus methods to add or subtract specific amounts of time. Note that these methods always return a new instance (Java 8 date/time classes are immutable).

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

// before 5 houres and 30 minutes

LocalDateTime dateTime = LocalDateTime.now().minusHours(5).minusMinutes(30);

TemporalAdjusters are another nice way for date manipulation. [TemporalAdjuster](http://download.java.net/jdk8/docs/api/java/time/temporal/TemporalAdjuster.html) is a single method interface that is used to separate the process of adjustment from actual date/time objects. A set of common TemporalAdjusters can be accessed using static methods of the [TemporalAdjusters](http://download.java.net/jdk8/docs/api/java/time/temporal/TemporalAdjusters.html) class.

LocalDate date = LocalDate.of(2014, Month.FEBRUARY, 25); // 2014-02-25

// first day of february 2014 (2014-02-01)

LocalDate firstDayOfMonth = date.with(TemporalAdjusters.firstDayOfMonth());

// last day of february 2014 (2014-02-28)

LocalDate lastDayOfMonth = date.with(TemporalAdjusters.lastDayOfMonth());

Static imports make this more fluent to read:

import static java.time.temporal.TemporalAdjusters.\*;

...

// last day of 2014 (2014-12-31)

LocalDate lastDayOfYear = date.with(lastDayOfYear());

// first day of next month (2014-03-01)

LocalDate firstDayOfNextMonth = date.with(firstDayOfNextMonth());

// next sunday (2014-03-02)

LocalDate nextSunday = date.with(next(DayOfWeek.SUNDAY));

**Time Zones**

Working with time zones is another big topic that is simplified by the new API. The LocalDate/Time classes we have seen so far do not contain information about a time zone. If we want to work with a date/time in a certain time zone we can use [ZonedDateTime](http://download.java.net/jdk8/docs/api/java/time/ZonedDateTime.html) or [OffsetDateTime](http://download.java.net/jdk8/docs/api/java/time/OffsetDateTime.html):

ZoneId losAngeles = ZoneId.of("America/Los\_Angeles");

ZoneId berlin = ZoneId.of("Europe/Berlin");

// 2014-02-20 12:00

LocalDateTime dateTime = LocalDateTime.of(2014, 02, 20, 12, 0);

// 2014-02-20 12:00, Europe/Berlin (+01:00)

ZonedDateTime berlinDateTime = ZonedDateTime.of(dateTime, berlin);

// 2014-02-20 03:00, America/Los\_Angeles (-08:00)

ZonedDateTime losAngelesDateTime = berlinDateTime.withZoneSameInstant(losAngeles);

int offsetInSeconds = losAngelesDateTime.getOffset().getTotalSeconds(); // -28800

// a collection of all available zones

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

// using offsets

LocalDateTime date = LocalDateTime.of(2013, Month.JULY, 20, 3, 30);

ZoneOffset offset = ZoneOffset.of("+05:00");

// 2013-07-20 03:30 +05:00

OffsetDateTime plusFive = OffsetDateTime.of(date, offset);

// 2013-07-19 20:30 -02:00

OffsetDateTime minusTwo = plusFive.withOffsetSameInstant(ZoneOffset.ofHours(-2));

**Timestamps**

Classes like LocalDate and ZonedDateTime provide a human view on time. However, often we need to work with time viewed from a machine perspective. For this we can use the [Instant](http://download.java.net/jdk8/docs/api/java/time/Instant.html) class which represents timestamps. An Instant counts the time beginning from the first second of January 1, 1970 (1970-01-01 00:00:00) also called the [EPOCH](http://download.java.net/jdk8/docs/api/java/time/Instant.html#EPOCH). Instant values can be negative if they occured before the epoch. They follow[ISO 8601](http://en.wikipedia.org/wiki/ISO_8601) the standard for representing date and time.

// current time

Instant now = Instant.now();

// from unix timestamp, 2010-01-01 12:00:00

Instant fromUnixTimestamp = Instant.ofEpochSecond(1262347200);

// same time in millis

Instant fromEpochMilli = Instant.ofEpochMilli(1262347200000l);

// parsing from ISO 8601

Instant fromIso8601 = Instant.parse("2010-01-01T12:00:00Z");

// toString() returns ISO 8601 format, e.g. 2014-02-15T01:02:03Z

String toIso8601 = now.toString();

// as unix timestamp

long toUnixTimestamp = now.getEpochSecond();

// in millis

long toEpochMillis = now.toEpochMilli();

// plus/minus methods are available too

Instant nowPlusTenSeconds = now.plusSeconds(10);

**Periods and Durations**

[Period](http://download.java.net/jdk8/docs/api/java/time/Period.html) and [Duration](http://download.java.net/jdk8/docs/api/java/time/Duration.html) are two other important classes. Like the names suggest they represent a quantity or amount of time. A Period uses date based values (years, months, days) while a Duration uses seconds or nanoseconds to define an amount of time. Duration is most suitable when working with Instants and machine time. Periods and Durations can contain negative values if the end point occurs before the starting point.

// periods

LocalDate firstDate = LocalDate.of(2010, 5, 17); // 2010-05-17

LocalDate secondDate = LocalDate.of(2015, 3, 7); // 2015-03-07

Period period = Period.between(firstDate, secondDate);

int days = period.getDays(); // 18

int months = period.getMonths(); // 9

int years = period.getYears(); // 4

boolean isNegative = period.isNegative(); // false

Period twoMonthsAndFiveDays = Period.ofMonths(2).plusDays(5);

LocalDate sixthOfJanuary = LocalDate.of(2014, 1, 6);

// add two months and five days to 2014-01-06, result is 2014-03-11

LocalDate eleventhOfMarch = sixthOfJanuary.plus(twoMonthsAndFiveDays);

// durations

Instant firstInstant= Instant.ofEpochSecond( 1294881180 ); // 2011-01-13 01:13

Instant secondInstant = Instant.ofEpochSecond(1294708260); // 2011-01-11 01:11

Duration between = Duration.between(firstInstant, secondInstant);

// negative because firstInstant is after secondInstant (-172920)

long seconds = between.getSeconds();

// get absolute result in minutes (2882)

long absoluteResult = between.abs().toMinutes();

// two hours in seconds (7200)

long twoHoursInSeconds = Duration.ofHours(2).getSeconds();

**Formatting and Parsing**

Formatting and parsing is another big topic when working with dates and times. In Java 8 this can be accomplished by using the format() and parse() methods:

// 2014-04-01 10:45

LocalDateTime dateTime = LocalDateTime.of(2014, Month.APRIL, 1, 10, 45);

// format as basic ISO date format (20140220)

String asBasicIsoDate = dateTime.format(DateTimeFormatter.BASIC\_ISO\_DATE);

// format as ISO week date (2014-W08-4)

String asIsoWeekDate = dateTime.format(DateTimeFormatter.ISO\_WEEK\_DATE);

// format ISO date time (2014-02-20T20:04:05.867)

String asIsoDateTime = dateTime.format(DateTimeFormatter.ISO\_DATE\_TIME);

// using a custom pattern (01/04/2014)

String asCustomPattern = dateTime.format(DateTimeFormatter.ofPattern("dd/MM/yyyy"));

// french date formatting (1. avril 2014)

String frenchDate = dateTime.format(DateTimeFormatter.ofPattern("d. MMMM yyyy", new Locale("fr")));

// using short german date/time formatting (01.04.14 10:45)

DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT)

.withLocale(new Locale("de"));

String germanDateTime = dateTime.format(formatter);

// parsing date strings

LocalDate fromIsoDate = LocalDate.parse("2014-01-20");

LocalDate fromIsoWeekDate = LocalDate.parse("2014-W14-2", DateTimeFormatter.ISO\_WEEK\_DATE);

LocalDate fromCustomPattern = LocalDate.parse("20.01.2014", DateTimeFormatter.ofPattern("dd.MM.yyyy"));

**Conversion**

Of course we do not always have objects of the type we need. Therefore, we need an option to convert different date/time related objects between each other. The following examples show some of the possible conversion options:

// LocalDate/LocalTime <-> LocalDateTime

LocalDate date = LocalDate.now();

LocalTime time = LocalTime.now();

LocalDateTime dateTimeFromDateAndTime = LocalDateTime.of(date, time);

LocalDate dateFromDateTime = LocalDateTime.now().toLocalDate();

LocalTime timeFromDateTime = LocalDateTime.now().toLocalTime();

// Instant <-> LocalDateTime

Instant instant = Instant.now();

LocalDateTime dateTimeFromInstant = LocalDateTime.ofInstant(instant, ZoneId.of("America/Los\_Angeles"));

Instant instantFromDateTime = LocalDateTime.now().toInstant(ZoneOffset.ofHours(-2));

// convert old date/calendar/timezone classes

Instant instantFromDate = new Date().toInstant();

Instant instantFromCalendar = Calendar.getInstance().toInstant();

ZoneId zoneId = TimeZone.getDefault().toZoneId();

ZonedDateTime zonedDateTimeFromGregorianCalendar = new GregorianCalendar().toZonedDateTime();

// convert to old classes

Date dateFromInstant = Date.from(Instant.now());

TimeZone timeZone = TimeZone.getTimeZone(ZoneId.of("America/Los\_Angeles"));

GregorianCalendar gregorianCalendar = GregorianCalendar.from(ZonedDateTime.now());

**Conclusion**