Date and time handling in Java

Jeszenszky, Péter
University of Debrecen, Faculty of Informatics
jeszenszky.peter@inf.unideb.hu

Kocsis, Gergely (English version)
University of Debrecen, Faculty of Informatics
kocsis.gergely@inf.unideb.hu

Last modified: 09.03.2017

History

- JDK 1.0 (1996): a `java.lang.System.currentTimeMillis()` method and the `java.util.Date` class
- JDK 1.1 (1997):
  - The `java.util.Calendar`, `java.util.GregorianCalendar`, `java.util.TimeZone` and `java.util.SimpleTimeZone` classes
  - Formatting, elementing: the `java.text.DateFormat` and the `java.text.SimpleDateFormat` classes
  - JDBC: the `java.sql.Date`, `java.sql.Time` and the `java.sql.Timestamp` classes
- Java SE 5 (2004): `javax.xml.datatype` package
- Java SE 8 (2014): *JSR 310: Date and Time API* (the `java.time` package and its subpackages)

java.util.Date (1)

- Represents a point of time with millisecond accuracy https://docs.oracle.com/javase/8/docs/api/java/util/Date.html
- Problems:
  - The name is misleading
  - Most constructors are deprecated
  - Indexing of months starts from 0
  - Uses local time zone
  - *Mutability*
    - Whenever an object is to be returned it its better to return only a copy in order not to let the object be changed through the reference
  - The class does not provide methods to manipulate objects (e.g. to get the date that is one day after a given date)

java.util.Date (2)

- Example use:

```
Date date = new Date(1848 - 1900, 3 - 1, 15);
System.out.println(date);   // Wed Mar 15 00:00:00 CET 1848
System.out.println(date.getTimezoneOffset());  // -60

DateFormat format = new SimpleDateFormat("yyy MMMM dd.", new Locale("HU"));
System.out.println(format.format(date));   // 1848 march 15.

date = format.parse("2016 március 1.");
System.out.println(date);   // Tue Mar 01 00:00:00 CET 2016

date = new Date();
System.out.println(date);   // Mon Feb 29 16:52:51 CET 2016

date.setTime(0);
System.out.println(date);   // Thu Jan 01 01:00:00 CET 1970
```

JDBC data types (1)

- Data types matching to SQL data types
  - `java.sql.Date`: represents SQL `DATE`
    https://docs.oracle.com/javase/8/docs/api/java/sql/Date.html
  - `java.sql.Time`: represents SQL `TIME`
    https://docs.oracle.com/javase/8/docs/api/java/sql/Time.html
  - `java.sql.Timestamp`: represents SQL `TIMESTAMP`
    https://docs.oracle.com/javase/8/docs/api/java/sql/Timestamp.html
    - Nanosecond accuracy
- All three extends the `java.util.Date` class
- None of them stores the time zone

JDBC data types (2)

- Example use:

```
Date date = Date.valueOf("1848-03-15");
System.out.println(date); // 1848-03-15

Time time = Time.valueOf("17:49:03");
System.out.println(time); // 17:49:03

Timestamp timestamp = new Timestamp(System.currentTimeMillis());
System.out.println(timestamp);   // 2016-02-29 17:51:13.886
```

java.util.Calendar (1)

- Represents a point of time with millisecond accuracy https://docs.oracle.com/javase/8/docs/api/java/util/Calendar.
  - Provides access to year, month, day etc. fields
  - Stores time zone
  - Provides methods to manipulate objects
    - Example: `add(...)`, `roll(...)`
- Abstract class, the only one exact class version is `java.util.GregorianCalendar`
  - Supports Gregorian and Julian calendar

java.util.Calendar (2)

- Still a problem:
  - Misleading name
  - Indexing months starts from 0
  - *Mutability*

java.util.Calendar (3)

- Example use:

```
Calendar cal = new GregorianCalendar(1848, 2, 15);
System.out.println(cal);
// java.util.GregorianCalendar[time=?,areFieldsSet=false,
// areAllFieldsSet=false, ...,YEAR=1848,MONTH=2,WEEK_OF_YEAR=?,
// WEEK_OF_MONTH=?,DAY_OF_MONTH=15, ...]

int year = cal.get(Calendar.YEAR);                // 1848
int month = cal.get(Calendar.MONTH);              // 2 -> March
int dayOfMonth = cal.get(Calendar.DAY_OF_MONTH);// 15
int dayOfWeek = cal.get(Calendar.DAY_OF_WEEK);  // 4 -> Wdnesday
int dayOfYear = cal.get(Calendar.DAY_OF_YEAR);  // 75

TimeZone tz = cal.getTimeZone();
System.out.println(tz.getDisplayName()); // Central European Time
System.out.println(tz.getID());          // Europe/Budapest

DateFormat format = SimpleDateFormat.getDateInstance(
    DateFormat.FULL, Locale.GERMAN);

Date date = cal.getTime();
System.out.println(format.format(date)); // Mittwoch, 15. März 1848
```

java.util.Calendar (4)

- Example use:

```
Date date = cal.getTime();
System.out.println(format.format(date)); // Mittwoch, 15. März 1848

cal.add(Calendar.DAY_OF_MONTH, 16);
date = cal.getTime();
System.out.println(format.format(date)); // Freitag, 31. März 1848

cal.roll(Calendar.MONTH, -1);
date = cal.getTime();
System.out.println(format.format(date)); // Dienstag, 29. Februar 1848
```

java.util.Calendar (5)

- Example use:

```
Calendar now = Calendar.getInstance();
System.out.println(now.getClass().getName());
// java.util.GregorianCalendar

DateFormat format = SimpleDateFormat.getDateTimeInstance(
    DateFormat.FULL, DateFormat.FULL, new Locale("hu"));

Date date = now.getTime();
System.out.println(format.format(date));
// 2016. február 29. 20:03:05 CET

format.setTimeZone(TimeZone.getTimeZone("PST"));
System.out.println(format.format(date));
// 2016. február 29. 11:03:05 PST
```

java.util.Calendar (6)

- What is wrong with mutability?

```
public static long countDays(Calendar start, Calendar end) {
    long count = 0;
    while (start.before(end)) {
        start.add(Calendar.DAY_OF_MONTH, 1);
        ++count;
    }
    return count;
}

Calendar then = new GregorianCalendar(1848, 2, 15);
Calendar now = Calendar.getInstance();
System.out.println(countDays(then, now));  // 61668
System.out.println(countDays(then, now));  // What is the output?
```

java.util.Calendar (6)

- What is wrong with mutability?

```
public static long countDays(Calendar start, Calendar end) {
    long count = 0;
    while (start.before(end)) {
        start.add(Calendar.DAY_OF_MONTH, 1);
        ++count;
    }
    return count;
}

Calendar then = new GregorianCalendar(1848, 2, 15);
Calendar now = Calendar.getInstance();
System.out.println(countDays(then, now));  // 61668
System.out.println(countDays(then, now));  // 0
```

java.util.Calendar (7)

- The correct use:

```
public static long countDays(Calendar start, Calendar end) {
    long count = 0;
    start = (Calendar) start.clone();
    while (start.before(end)) {
        start.add(Calendar.DAY_OF_MONTH, 1);
        ++count;
    }
    return count;
}

Calendar then = new GregorianCalendar(1848, 2, 15);
Calendar now = Calendar.getInstance();
System.out.println(countDays(then, now));  // 61668
System.out.println(countDays(then, now));  // 61668
```

W3C XML Schema (1)

- Paul V. Biron, Ashok Malhotra (ed). *XML Schema Part 2: Datatypes Second Edition*. W3C Recommendation. 28 October 2004. https://www.w3.org/TR/xmlschema-2/
  - Definition of standard data types for date and time handling
- Anders Berglund, Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, Norman Walsh (ed). *XQuery 1.0 and XPath 2.0 Data Model (XDM)*. W3C Recommendation. 23 January 2007. https://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/
  - Two more data types: `xsd:dayTimeDuration`, `xsd:yearMonthDuration`

W3C XML Schema (2)

- Representation of values of data types by character sequences (literals) and the definition of the syntax

W3C XML Schema (3)

- Bulit-in data types for handling date and time:
  - **`xsd:date`**: representing calendar dates
    - Literals: `1848-03-15`, `2004-10-28+13:00`, `2016-02-29Z`
  - **`xsd:time`**: Representing points of time repeating every day
    - Lierals: `12:15:00` (no time zone), `23:05:30Z` (world time), `00:45:00+01:00` (Central European time)
  - **`xsd:dateTime`**: data type representing time points
    - Literals: `1969-07-16T13:32:00Z`
  - **`xsd:duration`**: Data type representing time intervals
    - Literals: `P1Y2DT5H30M` (1 year, 2 days, 5 hours and 30 minutes), `P120Y6M` (120 years and 6 months), `PT9.58S` (9,58 seconds)

W3C XML Schema (4)

- Bulit-in data types for handling date and time:
  - **`xsd:gDay`**: Data type representing monthly repeating days of the Gregorian calendar
    - Literals: `---05` (the 5th day of the month)
  - **`xsd:gMonth`**: Data type representing the 12 months of the Gregorian Calendar
    - Literals: `--03` (March)
  - **`xsd:gYear`**: Data type representing the years of the Gregorian Calendar
    - Literals: `-0753, 0476, 1984`
  - **`xsd:gMonthDay`**: Data type representing yearly repeating dates of the Gregorian Calendar
    - Literals: `--05-25` (25 May)
  - **`xsd:gYearMonth`**: Data type representing the months of years of the Gregorian Calendar
    - Literals: `1848-03` (March 1848)

W3C XML Schema (5)

- Java support: `javax.xml.datatype` package https://docs.oracle.com/javase/8/docs/api/javax/xm
  - From J2SE 5 (2004)
- Example use:

```
DatatypeFactory factory = DatatypeFactory.newInstance();

Duration duration = factory.newDuration(true, 1, 0, 2, 5, 30, 30);
System.out.println(duration);   // P1Y0M2DT5H30M30S

XMLGregorianCalendar date = factory.newXMLGregorianCalendarDate(1848, 3,
    15, DatatypeConstants.FIELD_UNDEFINED);
System.out.println(date);   // 1848-03-15

date.add(duration);
System.out.println(date);   // 1849-03-17
```

Joda-Time

- Java library for handling date and time
  http://www.joda.org/joda-time/
  - License: Apache License v2
- *De-facto* standard before Java SE 8 in industry
- The `java.time package of Java SE 8` made it unnecessary

JSR 130

- *JSR 310: Date and Time API (Final Release)*. 4 March 2014. https://jcp.org/en/jsr/detail?id=310
  - Packages: `java.time`, `java.time.chrono`, `java.time.format`, `java.time.temporal`, `java.time.zone`
- Java SE 8
- See more:
  - *The Java™ Tutorials – Trail: Date Time* https://docs.oracle.com/javase/tutorial/datetime/

The basic standard

- ISO 8601:2004: *Data elements and interchange formats – Information interchange – Representation of dates and times*
  http://www.iso.org/iso/home/standards/iso8601.htm

Planning goals

- Clearness:
  - Well defined deterministically working API
- Immutable classes:
  - An object cannot be changed after creation
- Fluent API:
  - The goal is good readability by the use of method chaining
- Extensibility:
  - Personalized calendar systems are possible to be implemented

Properties

- More date and time handling classes for different purposes
- Nanosecond accuracy
- All classes support the instantiation and formatting of objects from strings
- I18n and L10n support (internationalization):
  - The use of *Unicode Common Locale Data Repository* (CLDR) http://cldr.unicode.org/
- Time zones:
  - The use of IANA *Time Zone Database*

    http://www.iana.org/time-zones

Packages

- `java.time`:
  - Core API that contains classes to handle dates, times, periods, time zones etc...
- `java.time.chrono`:
  - API to handlee different calendar sysems
- `java.time.format`:
  - API to format date and time and to create objects from strings
- `java.time.temporal`:
  - Extended API mostly for those who develop program packages
- `java.time.zone`:
  - Time zone handling

Naming convention of methods

Prefix	Type	Meaning
`from`	Static object factory	Create object by converting the parameters
`of`	Static object factory	Create object by using the parameters
`parse`	Static object factory	Create object from String
`at`	Entity level	Combine an object with another one e.g.: `Date.atTime(Time)`
`format`	Entity level	Method to format the object
`get`	Entity level	Method to get a given value
`is`	Entity level	Method to test thee logical value of a attribute of an object
`minus`	Entity level	Subtract a given quantity from the object
`plus`	Entity level	Add a given quantity to the object
`to`	Entity level	Convert an object to another (different) object
`with`	Entity level	Returns a copy of the object where one element is changed. This is the immutable version of setter methods.

Date and Time classes

Class or Enum	Year	Month	Day	Hours	Minutes	Seconds*	Zone Offset	Zone ID	toString Output
Instant						✓			2013-08-20T15:16:26.355Z
LocalDate	✓	✓	✓						2013-08-20
LocalDateTime	✓	✓	✓	✓	✓	✓			2013-08-20T08:16:26.937
ZonedDateTime	✓	✓	✓	✓	✓	✓	✓	✓	2013-08-21T00:16:26.941+09:00[Asia/Tokyo]
LocalTime				✓	✓	✓			08:16:26.943
MonthDay		✓	✓						--08-20
Year	✓								2013
YearMonth	✓	✓							2013-08
Month		✓							AUGUST
OffsetDateTime	✓	✓	✓	✓	✓	✓	✓		2013-08-20T08:16:26.954-07:00
OffsetTime				✓	✓	✓	✓		08:16:26.957-07:00
Duration			**	**	**	✓			PT20H (20 hours)
Period	✓	✓	✓				***	***	P10D (10 days)

* Seconds are captured to nanosecond precision.

** This class does not store this information, but has methods to provide time in these units.

Source: https://docs.oracle.com/javase/tutorial/datetime/iso/overview.html

Example of basic use of the API

```
LocalDate date = LocalDate.now();
System.out.println(date); // 2017-03-09

date = LocalDate.of(1848, 3, 15);
System.out.println(date); // 1848-03-15

date = LocalDate.of(1848, Month.MARCH, 15);
System.out.println(date); // 1848-03-15

LocalTime time = LocalTime.of(14, 52, 26);
System.out.println(time); // 14:52:26

LocalDateTime dateTime = date.atTime(time);
System.out.println(dateTime); // 1848-03-15T14:52:26

ZonedDateTime zonedDateTime = dateTime.atZone(ZoneId.of("CET"));
System.out.println(zonedDateTime);
// 1848-03-15T14:52:26+01:00[CET]
```

Rich and clean API

- Example:

```
LocalDate today = LocalDate.now();
System.out.println(today);                    // 2017-03-09
System.out.println(today.getEra());           // CE
System.out.println(today.getYear());          // 2017
System.out.println(today.getMonth());         // MARCH
System.out.println(today.getMonthValue()); // 3
System.out.println(today.getDayOfMonth()); // 9
System.out.println(today.getDayOfWeek());  // TUESDAY
System.out.println(today.lengthOfMonth()); // 31
System.out.println(today.getDayOfYear());  // 61
System.out.println(today.isLeapYear());    // true

LocalDate date = Year.of(1848).atMonth(Month.MARCH).atDay(15);
System.out.println(date);                     // 1848-03-15
```

Fluent API

- Example:

```
LocalDate today = LocalDate.now();
LocalDate yesterday = today.minusDays(1);
LocalDate tomorrow = today.plusDays(1);
LocalDate nextWeek = today.plusWeeks(1);
LocalDate nextYear = today.plusYears(1);

LocalTime time = LocalTime.now().plusHours(1).plusMinutes(45);
```

Solution of the exercise shown before

- It can be solved by one method call:

```
public static long countDays(LocalDate start, LocalDate end) {
    return start.until(end, ChronoUnit.DAYS);
}

LocalDate then = LocalDate.of(1848, Month.MARCH, 15);
System.out.println(then); // 1848-03-15

LocalDate now = LocalDate.now(); // 2017-03-09
System.out.println(now);

System.out.println(countDays(then, now));  // 61347
```

Parts of dates

- Partial dates can be handled by the `MonthDay`, `Year` and `YearMonth` classes
- Example:

```
MonthDay monthDay = MonthDay.of(Month.JUNE, 22);
System.out.println(monthDay); // --06-22
LocalDate date = monthDay.atYear(1975);
System.out.println(date);     // 1975-06-22

LocalDate   dateOfBirth = LocalDate.of(1809, 1, 19);
MonthDay    birthday = MonthDay.from(dateOfBirth);
System.out.println(birthday); // --01-19
```

Periods and Durations (1)

- `java.time.Period`:
  - Representing a time period in years, months, and days
- `java.time.Duration`:
  - Representing a duration in seconds and nanoseconds.

Periods and Durations (2)

- Példa a használatukra:

```
LocalDate dateOfBirth = LocalDate.of(2012, Month.APRIL, 4);
LocalDate today = LocalDate.now();
System.out.println(today);// 2017-01-15

Period age = dateOfBirth.until(today);
System.out.println(age);                    // P4Y9M11D
System.out.println(age.getYears());         // 4
System.out.println(age.getMonths());        // 9
System.out.println(age.getDays());          // 11
System.out.println(age.isNegative());       // false

Instant t1 = Instant.now();
System.out.println(t1);              // 2017-01-15T10:23:18.678Z

Duration duration = Duration.ofMinutes(50).plusSeconds(48);
System.out.println(duration);     // PT50M48S

Instant t2 = t1.plus(duration);
System.out.println(t2);              // 2017-01-15T11:14:06.678Z
```

Changing date and time (1)

- The `withXXX(...)` methods are used to modify dates and times:

```
LocalDate date = LocalDate.now();
System.out.println(date);  // 2017-03-09

date = date.withMonth(6);
System.out.println(date);  // 2017-06-09

date = date.withDayOfMonth(22);
System.out.println(date);  // 2017-06-22

date = date.withYear(1975);
System.out.println(date);  // 1975-06-22
```

Changing date and time (2)

- `java.time.temporal.TemporalAdjuster` is a functional innterface to change date and time. It can be used by the use of `with(TemporalAdjuster)` method of the date and time classes.
  - Built-in implementations are in the `java.time.temporal.TemporalAdjusters` class

Changing date and time (3)

- Example:

```
LocalDate date = LocalDate.of(1848, Month.MARCH, 15);
System.out.println(date); // 1848-03-15

System.out.println(date
    .with(TemporalAdjusters.lastDayOfMonth()));
// 1848-03-31

System.out.println(date
    .with(TemporalAdjusters.firstDayOfNextMonth()));
// 1848-04-01

System.out.println(date
    .with(TemporalAdjusters.firstInMonth(DayOfWeek.SUNDAY)));
// 1848-03-05

System.out.println(date
    .with(TemporalAdjusters.nextOrSame(DayOfWeek.WEDNESDAY)));
// 1848-03-15
```

Queries (1)

- The `java.time.temporal.TemporalQuery` is a functional interface to get information.
- It can be used by the `query(TemporalQuery)` method of date and time classes.
  - Built-in implementations are in the `java.time.temporal.TemporalQueries` class
- Example:

```
LocalTime now = LocalTime.now();
System.out.println(now.query(TemporalQueries.precision()));
// Nanos
```

Queries (2)

- Example (1):

```
public static Boolean isClassBreak(TemporalAccessor temporal) {
    int hour = temporal.get(ChronoField.HOUR_OF_DAY);
    int minute = temporal.get(ChronoField.MINUTE_OF_HOUR);
    if (hour < 8 || hour > 18) return false;
    return (hour % 2 == 1) && (minute >= 40 && minute < 60);
}

LocalTime time = LocalTime.of(14, 15);
System.out.println(time.query(QueryExample::isClassBreak));
// false

time = LocalTime.of(15, 42);
System.out.println(time.query(QueryExample::isClassBreak));
// true
```

Queries (3)

- Example (2):

```
Stream.of("08:45", "09:40", "09:59", "10:00", "11:50",
       "17:45", "19:30")
    .map(LocalTime::parse)
    .map(QueryExample::isClassBreak)
    .forEach(System.out::println);
// false
// true
// true
// false
// true
// true
// false
```

Formatting and parsing

- Example:

```
LocalDate date = LocalDate.parse("04 Feb 2016",
    DateTimeFormatter.ofPattern("dd MMM yyyy")
        .withLocale(Locale.ENGLISH));
System.out.println(date); // 2016-02-04

DateTimeFormatter formatter = new DateTimeFormatterBuilder()
    .appendValue(ChronoField.DAY_OF_MONTH, 2)
    .appendLiteral(' ')
    .appendText(ChronoField.MONTH_OF_YEAR, TextStyle.SHORT)
    .appendLiteral(' ')
    .appendValue(ChronoField.YEAR).toFormatter(Locale.ENGLISH);

date = LocalDate.parse("04 Feb 2016", formatter);
System.out.println(date); // 2016-02-04

System.out.println(LocalDate.now().format(formatter)); // 01 Mar 2016
```

Dates b.C.

- Example use:

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("y G",
    Locale.ENGLISH);

Year year = Year.parse("753 BC", formatter);// Funding of Rome
System.out.println(year);                   // -752
System.out.println(formatter.format(year)); // 753 BC
```

Instant

- A class representing a given point of time
  - Can be applied e.g. to register timestamps in applications
- Example use:

```
Instant now = Instant.now();
System.out.println(now);      // 2017-03-09T14:10:31.646Z

LocalDateTime dateTime =
   LocalDateTime.from(now);   // java.time.DateTimeException

LocalDateTime dateTime = LocalDateTime.ofInstant(now,
   ZoneId.systemDefault());
System.out.println(dateTime); // 2016-03-02T15:10:31.646
```

Clock (1)

- Provides access to the actual instant, time, date ina given time zone
  - See the `now(Clock)` methods of date and time classes. It uses the Clock given as a parameter when creating the object.
  - This is useful for testing purposes.

Clock (2)

- Example use:

```
Clock clock1 = Clock.systemDefaultZone();
Clock clock2 = Clock.systemUTC();
Clock clock3 = Clock.system(ZoneId.of("Asia/Vladivostok"));

System.out.println(clock1);   // SystemClock[Europe/Budapest]
System.out.println(clock2);   // SystemClock[Z]
System.out.println(clock3);   // SystemClock[Asia/Vladivostok]

System.out.println(LocalTime.now(clock1));   // 14:01:26.204
System.out.println(LocalTime.now(clock2));   // 13:01:26.209
System.out.println(LocalTime.now(clock3));   // 23:01:26.209
```

Clock (3)

- Example use: stop the time (Useful for testing.)

```
Instant instant = Instant.parse("2016-03-01T00:00:00Z");
Clock clock = Clock.fixed(instant, ZoneId.of("UTC"));

System.out.println(ZonedDateTime.now(clock));
// 2016-03-01T00:00Z[UTC]

Thread.sleep(5000);   // 5 second delay

System.out.println(ZonedDateTime.now(clock));
// 2016-03-01T00:00Z[UTC]

Instant now = Instant.now(clock);
System.out.println(now.equals(instant)); // true
```

Example

- If an air plane departures at 01.04 11:45 (local time) from New York to Tokyo and the travel time is 14 hours 10 minutes, when will it arrive to Tokyo (local time)?

```
ZonedDateTime departure = LocalDate.of(2016, Month.APRIL, 1)
    .atTime(11, 45)
    .atZone(ZoneId.of("America/New_York"));
System.out.println(departure);
// 2016-04-01T11:45-04:00[America/New_York]

Duration duration = Duration.ofHours(14).plusMinutes(10);
System.out.println(duration);
// PT14H10M

ZonedDateTime arrival =
departure.withZoneSameInstant(ZoneId.of("Asia/Tokyo"))
    .plus(duration);
System.out.println(arrival);
// 2016-04-02T14:55+09:00[Asia/Tokyo]
```

Example

- What days of a year are on 13. Friday?

```
public static List<LocalDate> fridayThe13th(int year) {
    return IntStream.range(1, 13)
        .mapToObj(i -> LocalDate.of(year, i, 13))
        .filter(d -> d.getDayOfWeek().equals(DayOfWeek.FRIDAY))
        .collect(Collectors.toList());
}

System.out.println(fridayThe13th(2015));
// [2015-02-13, 2015-03-13, 2015-11-13]

System.out.println(fridayThe13th(2016));
// [2016-05-13]
```

Example (1)

- What day will be the next 13. Friday after a given day?
  - Naive solution for `LocalDate` objects
  - Generalized solution: as an implementation of `TemporalAdjuster`

Example (2)

- Naive solution for `LocalDate` objects:

```
public static LocalDate nextFriday13th(LocalDate date) {
    int dayOfMonth = date.getDayOfMonth();
    if (dayOfMonth != 13) {
        if (dayOfMonth > 13)
            date = date.plusMonths(1);
        date = date.withDayOfMonth(13);
    }
    while (date.getDayOfWeek() != DayOfWeek.FRIDAY)
        date = date.plusMonths(1);
    return date;
}

LocalDate date = LocalDate.of(2016, Month.MARCH, 1);
System.out.println(nextFriday13th(date));  // 2017-10-15
```

Example (3)

- Generalized solution:

```
public class NextFriday13th implements TemporalAdjuster {

    public Temporal adjustInto(Temporal temporal) {
        int dayOfMonth = temporal.get(ChronoField.DAY_OF_MONTH);
        if (dayOfMonth != 13) {
            if (dayOfMonth > 13)
                temporal = temporal.plus(1, ChronoUnit.MONTHS);
            temporal = temporal.with(ChronoField.DAY_OF_MONTH, 13);
        }
        while (temporal.get(ChronoField.DAY_OF_WEEK) != 5)
            temporal = temporal.plus(1, ChronoUnit.MONTHS);
        return temporal;
    }

}
```

Example (4)

- Generalized solution (continue): by using the `TemporalAdjuster` we can modify `LocalDate`, `LocalDateTime`, `ZonedDateTime` and `OffsetDateTime` objects as well

```
LocalDate date = LocalDate.of(2016, Month.MARCH, 1);
System.out.println(date.with(new NextFriday13th()));
// 2016-05-13

LocalDateTime dateTime = date.atStartOfDay();
System.out.println(dateTime.with(new NextFriday13th()));
// 2016-05-13T00:00
```

Further reading

- `java.time` (Java Platform SE 8)
  https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html
- *The Java™ Tutorials – Trail: Date Time*
  https://docs.oracle.com/javase/tutorial/datetime/