4.3 Datumsarithmetik

Die bislang vorgestellten Klassen und Interfaces aus dem neuen Date and Time API erleichtern die Datumsverarbeitung – jedoch habe ich bisher (zumindest komplexere) Berechnungen noch weitestgehend außen vor gelassen. Allerdings findet man gerade in der Praxis bei der Verarbeitung von Datumswerten oftmals auch die Notwendigkeit für Berechnungen, beispielsweise um einige Tage, Wochen oder gar Monate in die Zukunft oder die Vergangenheit zu springen. Praktischerweise sind diverse gebräuchliche Operationen der Datumsarithmetik in der Utility-Klasse `TemporalAdjusters` gebündelt und basieren auf dem Functional Interface `TemporalAdjuster` – beide aus dem Package `java.time.temporal`. Im Beispiel für die Klassen `LocalDate`, `LocalTime` und `LocalDateTime` haben wir bereits einen ersten Eindruck von den Möglichkeiten gewinnen können. Dieses Wissen wollen wir nun ausbauen.

Das Interface `TemporalAdjuster`

Ein `TemporalAdjuster` definiert die Methode `adjustInto(Temporal)`. In deren Implementierungen kann man eine flexible Anpassung sowohl für Datums- als auch Zeit-Objekte, also genauer für alle Objekte mit dem Basistyp `Temporal`, vornehmen:

```
@FunctionalInterface
public interface TemporalAdjuster
{
    Temporal adjustInto(Temporal temporal);
}
```

Realisierungen dieses Interface helfen dabei, Berechnungen auszuführen, wie etwa den ersten oder letzten Tag im Monat zu ermitteln. Relativ klar wird der Sinn, wenn man sich die vordefinierten Varianten anschaut.

Vordefinierte `TemporalAdjusters`

Wie eingangs erwähnt, findet man in der Utility-Klasse `TemporalAdjusters` eine Menge gebräuchlicher Operationen der Datumsarithmetik. Das sind unter anderem:

- `firstDayOfMonth()`, `firstDayOfNextMonth()` und `lastDayOfMonth()` – Diese Methoden berechnen den ersten oder letzten Tag im (nächsten) Monat.
- `firstDayOfYear()`, `firstDayOfNextYear()` und `lastDayOfYear()` – Damit ermittelt man den ersten oder letzten Tag im (nächsten) Jahr.
- `firstInMonth(DayOfWeek)`, `lastInMonth(DayOfWeek)` – Springe zum ersten oder letzten als Parameter übergebenen Wochentag im Monat.
- `next(DayOfWeek)`, `nextOrSame(DayOfWeek)`, `previous(DayOfWeek)` und `previousOrSame(DayOfWeek)` – Mit diesen Methoden kann man den nächsten oder vorherigen Wochentag im Monat berechnen, etwa den nächsten Freitag. Dabei wird gegebenenfalls auch berücksichtigt, ob man sich bereits an diesem Wochentag befindet. In diesem Fall findet selbstverständlich keine Anpassung statt.

Zum besseren Verständnis der aufgelisteten `TemporalAdjuster` schauen wir, wie einfach und gut lesbar sich damit Datumsarithmetik programmieren lässt – insbesondere wenn man dabei statische Imports nutzt. Im folgenden Listing habe ich bewusst einige Dinge nicht statisch importiert, um einen Vergleich zu ermöglichen, der den Gewinn an Lesbarkeit beim Einsatz der statischen Imports für die Datumsberechnungen zeigt.[6] Im Beispiel werden ausgehend vom 7. Februar 2015 einige »Zeitsprünge« vorgenommen und das Ergebnis ausgegeben:

```
import static java.time.temporal.TemporalAdjusters.firstDayOfNextMonth;
import static java.time.temporal.TemporalAdjusters.firstInMonth;
import static java.time.temporal.TemporalAdjusters.lastInMonth;
import static java.time.DayOfWeek.SATURDAY;

// ...

public static void main(final String[] args)
{
    final LocalDate date = LocalDate.of(2015, Month.FEBRUARY, 7);

    LocalDate firstOfMonth = date.with(TemporalAdjusters.firstDayOfMonth());
    LocalDate lastofMonth = date.with(TemporalAdjusters.lastDayOfMonth());
    LocalDate firstOfNextMonth = date.with(firstDayOfNextMonth());
    LocalDate firstMondayInMonth = date.with(firstInMonth(DayOfWeek.MONDAY));
    LocalDate lastSaturdayInMonth = date.with(lastInMonth(SATURDAY));

    System.out.println("date:                " + date);
    System.out.println("lastofMonth:         " + lastofMonth);
    System.out.println("firstOfMonth:        " + firstOfMonth);
    System.out.println("firstOfNextMonth:    " + firstOfNextMonth);
    System.out.println("firstMondayInMonth:  " + firstMondayInMonth);
    System.out.println("lastSaturdayInMonth: " + lastSaturdayInMonth);
}
```

Listing 4.19 *Ausführbar als* **'PREDEFINEDTEMPORALADJUSTERSEXAMPLE'**

Starten wir das Programm PREDEFINEDTEMPORALADJUSTERSEXAMPLE, so erhalten wir folgende Resultate der Datumsberechnungen:

```
date:                2015-02-07
lastofMonth:         2015-02-28
firstOfMonth:        2015-02-01
firstOfNextMonth:    2015-03-01
firstMondayInMonth:  2015-02-02
lastSaturdayInMonth: 2015-02-28
```

Aufrufe von `TemporalAdjuster` Wir wissen bereits, dass ein `TemporalAdjuster` die Methode `adjustInto(Temporal)` besitzt und durch deren Aufruf ein passend modifiziertes `Temporal`-Objekt zurückgibt.

Wenn wir das obige Beispiel anschauen, sehen wir aber nirgendwo den Aufruf der `adjustInto(Temporal)`-Methode. Stattdessen wird die Methode `with(Temporal-`

[6] Sie sollten statische Imports nur ganz bewusst für solche Fälle nutzen, wo sich dadurch die Lesbarkeit enorm steigert. Unüberlegt eingesetzt leidet oftmals die Verständlichkeit eines Programms, weil Abhängigkeiten und tatsächlich genutzte Klassen schwieriger ersichtlich sind.

`Adjuster)` aufgerufen. Das liegt daran, dass diese eine besser lesbare und zudem semantisch äquivalente Alternative ist. Folgende Aufrufe bewirken demnach dasselbe:

```
temporal = thisAdjuster.adjustInto(temporal);
temporal = temporal.with(thisAdjuster);
```

Speziellere vordefinierte `TemporalAdjusters`

Die zuvor genannten `TemporalAdjusters` sind für viele Anwendungsfälle schon ausreichend. Für mehr Flexibilität gibt es noch folgende zwei Methoden:

- `dayOfWeekInMonth(int, DayOfWeek)` – Errechnet den n-ten Wochentag im Monat. Dabei wird auch über Monatsgrenzen hinweg gesprungen, etwa wenn man den (nicht existierenden) 7. Dienstag eines Monats ermitteln wollte. Zudem sind negative Werte erlaubt, wobei die Werte 0 und -1 eine spezielle Bedeutung tragen: 0 ermittelt den letzten gewünschten Wochentag im Vormonat, -1 den letzten Wochentag in diesem Monat. Die negativen Werte rechnen ausgehend vom letzten Wochentag des Monats die übergebene Anzahl an Wochen zurück. Der -3. Sonntag wäre also 3 Wochen vor dem letzten Sonntag des Monats und man müsste die Werte -4 sowie `SUNDAY` an `dayOfWeekInMonth(int, DayOfWeek)` übergeben.
- `ofDateAdjuster(UnaryOperator<LocalDate>)` – Mithilfe dieser Methode lassen sich `TemporalAdjusters` erstellen. Die auszuführenden Berechnungen werden mithilfe eines `UnaryOperator<LocalDate>` beschrieben. Das ermöglicht viel Flexibilität: Mit dem Lambda `date -> date.plusDays(5)` springt man fünf Tage in die Zukunft. Andere Berechnungen sind ähnlich leicht.

Im nachfolgenden Listing definieren wir jeweils eine Hilfsmethode, die die Arbeitsweise der beiden Methoden aus der Aufzählung verdeutlicht:

```
public static void main(final String[] args)
{
    ofDateAdjusterCalculations();
    dayOfWeekInMonthCalculations();
}

private static void ofDateAdjusterCalculations()
{
    System.out.println("Adjusting with ofDateAdjuster()");
    System.out.println("-------------------------------");

    final Map<String, TemporalAdjuster> adjusters = new LinkedHashMap<>();
    adjusters.put("ONE_WEEK_LATER", ofDateAdjuster(localDate ->
                                                   localDate.plusDays(7)));
    adjusters.put("FOUR_WEEKS_LATER", ofDateAdjuster(localDate ->
                                                    localDate.plusDays(28)));
    adjusters.put("ONE_MONTH_LATER", ofDateAdjuster(localDate ->
                                                   localDate.plusMonths(1)));

    final LocalDate base = LocalDate.of(2012, FEBRUARY, 27);
    applyAdjusters(base, adjusters);
}
```

```
private static void dayOfWeekInMonthCalculations()
{
    System.out.println("\nAdjusting with dayOfWeekInMonth()");
    System.out.println("--------------------------------");

    final Map<String, TemporalAdjuster> adjusters = new LinkedHashMap<>();
    adjusters.put("THIRD_FRIDAY", dayOfWeekInMonth(3, FRIDAY));
    adjusters.put("LAST_FRIDAY", dayOfWeekInMonth(-1, FRIDAY));
    adjusters.put("LAST_FRIDAY_LAST_MONTH", dayOfWeekInMonth(0, FRIDAY));
    adjusters.put("FRIDAY_3_WEEKS_BEFORE_LAST", dayOfWeekInMonth(-4, FRIDAY));
    adjusters.put("FRIDAY_4_WEEKS_BEFORE_LAST", dayOfWeekInMonth(-5, FRIDAY));

    final LocalDate base = LocalDate.of(2015, FEBRUARY, 7);
    applyAdjusters(base, adjusters);
}

private static void applyAdjusters(final LocalDate localDate,
                        final Map<String, TemporalAdjuster> adjusters)
{
    System.out.println("Starting on: " + localDate);

    adjusters.forEach( (name, adjuster) ->
    {
       System.out.println("adjusting to " + name + ": " + base.with(adjuster));
    });
}
```

Listing 4.20 *Ausführbar als* **'MOREPREDEFINEDTEMPORALADJUSTERSEXAMPLE'**

Das obige Programm produziert folgende Ausgaben:

```
Adjusting with ofDateAdjuster()
-------------------------------
Starting on: 2012-02-27
adjusting to ONE_WEEK_LATER: 2012-03-05
adjusting to FOUR_WEEKS_LATER: 2012-03-26
adjusting to ONE_MONTH_LATER: 2012-03-27

Adjusting with dayOfWeekInMonth()
---------------------------------
Starting on: 2015-02-07
adjusting to THIRD_FRIDAY: 2015-02-20
adjusting to LAST_FRIDAY: 2015-02-27
adjusting to LAST_FRIDAY_LAST_MONTH: 2015-01-30
adjusting to FRIDAY_3_WEEKS_BEFORE_LAST: 2015-02-06
adjusting to FRIDAY_4_WEEKS_BEFORE_LAST: 2015-01-30
```

4.4 Das neue Date and Time API im Einsatz

Nach der Vorstellung zentraler Klassen und Interfaces des JSR-310 in den vorherigen Abschnitten wollen wir damit nochmals die Berechnung einer Zeitdifferenz und deren Ausgabe implementieren. Danach schauen wir auf Modifikationen der Zeit, etwa für Simulationen mit Zeitlupen- oder Zeitraffereffekten. Anschließend nutzen wir selbst definierte `TemporalAdjuster`s für spezielle Datunsarithmetiken. Abgerundet wird dieses Teilkapitel durch eine Betrachtung von Interoperabilität mit dem alten Datums-API.