11 Datumsverarbeitung seit JDK 8

In diesem Kapitel stelle ich das in JDK 8 integrierte Date and Time API vor. Einführend gebe ich in Abschnitt 11.1 einen Überblick über wichtige Aufzählungen, Klassen und Interfaces. Nachdem die Grundlagen für das Verständnis und den praktischen Einsatz gelegt wurden, beschäftigen wir uns in Abschnitt 11.2 mit dem Thema Datumsarithmetik. Schließlich stelle ich in Abschnitt 11.3 ein paar Informationen zur Migration bestehender Datumsverarbeitungsfunktionalitäten bereit.

Die im Rahmen von JDK 8 entwickelten Klassen adressieren die Probleme mit den bisherigen Datums-APIs des JDKs und nutzen vor allem Ideen aus der Bibliothek Joda-Time, deren Schöpfer Stephen Colebourne eine führende Rolle bei der Entwicklung des neuen Datums-APIs innehatte. Die Zielsetzung war, alles robuster und einfacher nutzbar zu machen und ein gelungenes, hilfreiches API zur Verwaltung und zur Manipulation von Datums- und Zeitwerten bereitzustellen. Diese Ergänzungen finden mit Java 8 endlich Einzug ins JDK. Schauen wir uns das Ganze mal an.

11.1 Überblick über die neu eingeführten Typen

Das mit JDK 8 realisierte Date and Time API fügt dem JDK einige Funktionalität in verschiedenen Packages unter `java.time` hinzu. Dabei unterscheidet man im Wesentlichen zwei Konzepte: Zum einen gibt es die kontinuierliche oder Maschinenzeit, die linear voranschreitet und für die durch die Klasse `java.time.Instant` ein spezieller Zeitpunkt repräsentiert wird. Das ist mit der Klasse `Date` vergleichbar, jedoch wird eine Auflösung im Bereich von Nanosekunden geboten. Zum anderen existieren Datumsklassen, die eher an menschlichen Denkweisen ausgerichtet sind: Die Klassen `LocalDate` und `LocalTime` aus dem Package `java.time` repräsentieren Datumswerte ohne Zeitzonen in Form eines Datums bzw. einer Zeit. Beide modellieren jeweils nur die durch den Klassennamen beschriebene Zeitkomponente, also Datum oder Zeit.

Nach dieser Einführung möchte ich nun einige neue Aufzählungen, Klassen und Interfaces kurz vorstellen, bevor ich dann in jeweils separaten Abschnitten auf die neuen Typen anhand von Beispielen ein wenig genauer eingehe.

11.1.1 Neue Aufzählungen, Klassen und Interfaces

Das Date and Time API definiert im Package `java.time.temporal` verschiedene Basisinterfaces, unter anderem folgende in Abbildung 11-1 gezeigte:

- `TemporalAccessor` – Dieses Interface bildet die Basis für viele Zeit- und Datumsrepräsentationen und definiert Lesezugriffe auf den jeweils modellierten Wert. Ein solcher besitzt eine Einheit, die durch das Interface `TemporalUnit` bestimmt wird. Diverse Aufzählungen implementieren das Interface `TemporalAccessor`, etwa `DayOfWeek` und `Month` zur Modellierung von Wochentagen bzw. Monaten.
- `TemporalAdjuster` – Dieses Interface definiert eine Basis für verschiedene Varianten der Anpassung von Datums- und Zeitwerten.

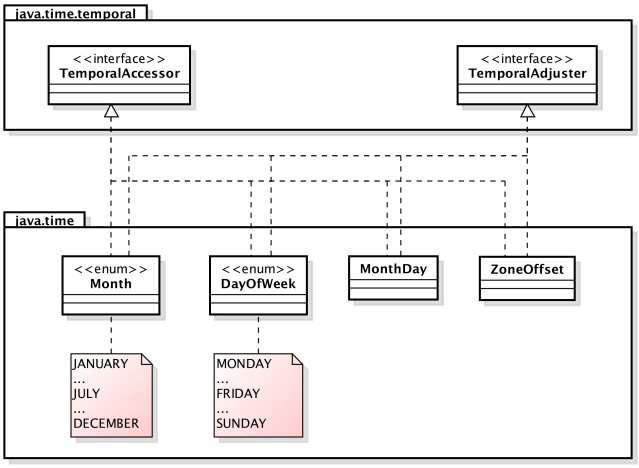

Abbildung 11-1 *Zentrale Interfaces im Überblick*

Ergänzend gibt es das Basisinterface `Temporal` sowie drei weitere Interfaces, die uns in den folgenden Abschnitten immer wieder einmal begegnen werden:

- `Temporal` – Dieses Interface ist eine Erweiterung von `TemporalAccessor` und ist die Basis für verschiedene Klassen aus dem neuen Date and Time API, die Zeitpunkte modellieren, wie `Instant` oder `LocalTime`. Das Interface `Temporal` bietet neben Lesezugriffen auch modifizierende Zugriffe, wobei hiermit gemeint ist, dass neue Instanzen mit veränderter Wertebelegung entstehen.
- `TemporalUnit` – Dieses Interface beschreibt eine Zeiteinheit. Konkrete Werte findet man in der Aufzählung `java.time.temporal.ChronoUnit`, etwa `MILLIS`.
- `TemporalAmount` – Dieses Interface modelliert eine Abstraktion für eine Zeitspanne und nutzt eine `TemporalUnit`.
- `TemporalField` – Dieses Interface bietet Zugriff auf Attribute von Datums- bzw. Zeitwerten, wie etwa die Attribute `DAY_OF_WEEK` oder `HOUR_OF_DAY`.

Diese drei sind zum Verständnis des Zusammenspiels in Abbildung 11-2 dargestellt.

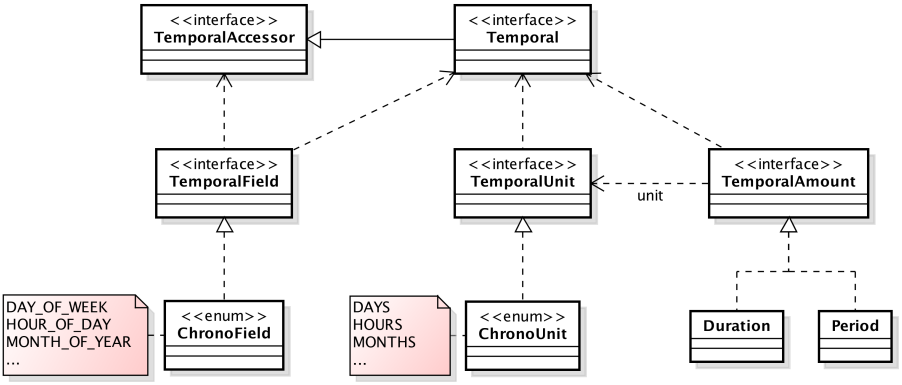

Abbildung 11-2 *Weitere Interface im Date and Time API*

Auch die in Abbildung 11-3 gezeigten und im Anschluss aufgelisteten Klassen machen die Arbeit mit dem neuen Date and Time API angenehm:

- `Instant` – Ein `Instant` repräsentiert ein eher technisches Konstrukt.
- `LocalDate`, `LocalTime` und `LocalDateTime` – Diese Klassen dienen der Verarbeitung von Datums- bzw. Zeitinformationen, aber auch zu deren Kombination.
- `ZoneId`, `ZoneOffset` und `ZoneDateTime` – Diese Klassen helfen bei der Verarbeitung von Daten, die sich auf Zeitzonen beziehen.

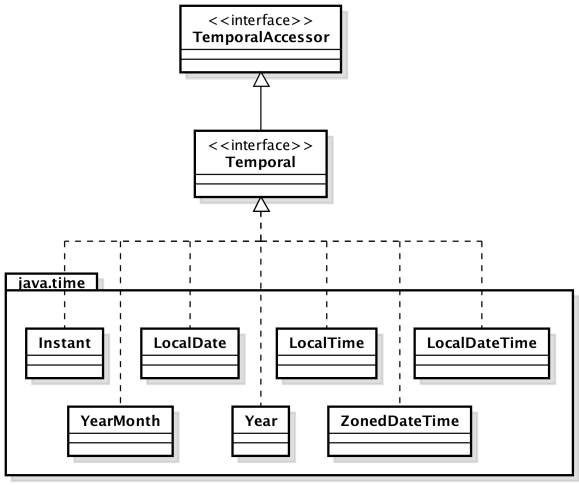

Abbildung 11-3 *Wichtige Klassen aus JSR-310 im Überblick*

11.1.2 Die Aufzählungen `DayOfWeek` und `Month`

Die Aufzählungen `java.time.DayOfWeek` und `java.time.Month` implementieren die Interfaces `TemporalAccessor` und `TemporalAdjuster`. Der Einsatz dieser Aufzählungen macht einerseits den Sourcecode lesbar und vermeidet andererseits auch einfache Fehler, weil man typsichere Konstanten anstelle von Magic Numbers verwenden kann. Bei Nutzung der alten APIs entstehen schnell Probleme durch Inkonsistenzen bei 0- und 1-basierten Angaben. Betrachten wir exemplarisch den Monat Februar. Im `Calendar`-API wusste man – ohne Blick in das Javadoc des APIs – nie so genau, ob Februar nun dem Wert 1 oder 2 entsprach.

Neben der Typsicherheit bieten die neuen Aufzählungstypen den Vorteil, dass man Berechnungen mit ihnen durchführen kann. Nachfolgend demonstriere ich dies, indem ich zu einem Sonntag 5 Tage hinzu addiere und zum Februar 10 Monate:

```
public static void main(final String[] args)
{
    final DayOfWeek sunday = DayOfWeek.SUNDAY;
    final Month february = Month.FEBRUARY;

    final DayOfWeek friday = sunday.plus(5);
    final Month december = february.plus(10);

    System.out.println(friday);
    System.out.println(december);
}
```

Listing 11.1 *Ausführbar als* **'MONTHANDDAYOFWEEKEXAMPLE'**

Wie erwartet, landet man an einem Freitag bzw. im Dezember, wenn man das Progamm MONTHANDDAYOFWEEKEXAMPLE ausführt:

```
FRIDAY
DECEMBER
```

11.1.3 Die Klassen `MonthDay`, `YearMonth` und `Year`

Für einige Anwendungsfälle benötigt man statt einer vollständigen Angabe aus Jahr, Monat und Tag lediglich eine Teilmenge der Informationen. Man kann sich Kombinationen aus Jahr und Monat, Monat und Tag sowie einfach nur Jahr vorstellen, um gewisse Datumsangaben zu modellieren. Für diese Zwecke wurden im Package `java.time` die Klassen `MonthDay`, `YearMonth` sowie `Year` eingeführt. Während `MonthDay` keine Veränderungen erlaubt, weil es die Interfaces `TemporalAccessor` und `TemporalAdjuster` implementiert, ermöglichen `YearMonth` und `Year` Modifikationen. Dazu realisieren diese neben `TemporalAdjuster` auch das Interface `Temporal`, das bei modifizierendem Zugriff jeweils entsprechend veränderte Instanzen zurückliefert. Im folgenden Listing nutzen wir die drei zuvor genannten Klassen:

```
public static void main(final String[] args)
{
    // MonthDay: Achtung, ISO-Format mit der Reihenfolge: Monat, Tag
    final MonthDay february7th = MonthDay.of(Month.FEBRUARY, 7);

    // YearMonth: Zur Lesbarkeit besser Month statisch importieren
    final YearMonth february2014 = YearMonth.of(2014, Month.FEBRUARY);

    // Year
    final Year year = Year.of(2012);
    final boolean isLeapYear = year.isLeap();

    System.out.println("MonthDay:  " + february7th);
    System.out.println("YearMonth: " + february2014);
    System.out.println("Year:      " + year + " / isLeap? " + isLeapYear);
}
```

Listing 11.2 *Ausführbar als '*YEARANDMOREEXAMPLE*'*

Das Programm YEARANDMOREEXAMPLE produziert folgende Konsolenausgaben:

```
MonthDay:  --02-07
YearMonth: 2014-02
Year:      2012 / isLeap? true
```

Anhand der Ausgaben sieht man verschiedene Notationsformen, wobei die Darstellung von `MonthDay` durch das Doppelminus etwas komisch anmutet. Man erkennt aber auch, dass man von der objektorientierten Umsetzung profitiert: Beispielsweise lässt sich per Aufruf von `isLeap()` prüfen, ob ein Jahr ein Schaltjahr ist.

11.1.4 Die Klasse `Instant`

Die Klasse `java.time.Instant` basiert auf dem Interface `Temporal`. Eine Instanz vom Typ `Instant` repräsentiert einen Zeitpunkt in Nanosekunden bezogen auf den Referenzzeitpunkt 1.1.1970 00:00:00 Uhr UTC. Die Zeit schreitet bei dieser Modellierung linear voran, wodurch sich die Verarbeitung durch Computer vereinfacht, da keine Spezialfälle zu betrachten sind.[1]

Im folgenden Beispiel modellieren wir Abfahrts- und Ankunftszeiten einer Reise mit der Dauer von 5 Stunden, die um 12:30 Uhr beginnt. Diesen Startzeitpunkt ermitteln wir aus textuellen Informationen durch den Aufruf von `parse(String)`. Zuvor zeige ich den Aufruf von `now()`, um den jetzigen Zeitpunkt zu bestimmen. Für die Berechnungen nutzen wir die Methode `plus(long, TemporalUnit)` bzw. die überladene Variante `plus(TemporalAmount)`, um auf einen Zeitpunkt vom Typ `Instant` eine Zeitspanne zu addieren. So erhalten wir die erwartete Ankunftszeit als `expectedArrivalTime`. Zudem nehmen wir eine Verspätung von 7 Minuten an, was der realen Ankunftszeit im `Instant`-Objekt `arrival` entspricht:

[1] Allerdings werden Schaltsekunden auf die letzten 1000 Sekunden des Tages verteilt, wodurch es möglicherweise zu Abweichungen von Zeitsystemen außerhalb von Java kommen kann.

```
public static void main(final String[] args)
{
    // Instant mit now() erzeugen
    final Instant now = Instant.now();

    // Abfahrt 12:30 und Reisedauer 5 Stunden sowie 7 Minuten Verspätung
    final Instant departureTime = Instant.parse("2015-02-07T12:30:00Z");
    final Instant expectedArrivalTime = departureTime.plus(5, ChronoUnit.HOURS);
    final Instant arrival = expectedArrivalTime.plus(Duration.ofMinutes(7));

    System.out.println("now:       " + now);
    System.out.println("departure: " + departureTime);
    System.out.println("expected:  " + expectedArrivalTime);
    System.out.println("arrival:   " + arrival);
}
```

Listing 11.3 *Ausführbar als* **'INSTANTEXAMPLE'**

Das Programm INSTANTEXAMPLE gibt in etwa Folgendes aus:

```
now:       2015-05-17T17:28:10.654Z
departure: 2015-02-07T12:30:00Z
expected:  2015-02-07T17:30:00Z
arrival:   2015-02-07T17:37:00Z
```

11.1.5 Die Klasse `Duration`

Die Klasse `java.time.Duration` implementiert das Interface `TemporalAmount` und erlaubt es, eine Zeitdauer in Nanosekunden festzulegen, etwa um Differenzen zwischen zwei `Instant`-Objekten auszudrücken. Instanzen der Klasse `Duration` können durch Aufruf verschiedener Methoden konstruiert werden, z. B. aus Werten verschiedener Zeiteinheiten[2] wie folgt:

```
public static void main(final String[] args)
{
    // Erzeugung mit ofXYZ()-Methoden
    final Duration durationFromNanos = Duration.ofNanos(7);
    final Duration durationFromSecs = Duration.ofSeconds(15);
    final Duration durationFromMinutes = Duration.ofMinutes(30);
    final Duration durationFromHours = Duration.ofHours(45);
    final Duration durationFromDays = Duration.ofDays(60);

    System.out.println("From Nanos:   " + durationFromNanos);
    System.out.println("From Secs:    " + durationFromSecs);
    System.out.println("From Minutes: " + durationFromMinutes);
    System.out.println("From Hours:   " + durationFromHours);
    System.out.println("From Days:    " + durationFromDays);
}
```

Listing 11.4 *Ausführbar als* **'DURATIONEXAMPLE'**

Führen wir das Programm DURATIONEXAMPLE aus, so kommt es zu folgenden Ausgaben, wobei im Speziellen folgende Dinge von Interesse sind: Zum einen werden Zeit-

[2] Zeiteinheiten mit variabler Länge, wie Monate, werden nicht unterstützt.

differenzen maximal in der Zeiteinheit von Stunden abgebildet, wodurch für 60 Tage der Wert 1440 Stunden zustande kommt:

```
From Nanos:   PT0.000000007S
From Secs:    PT15S
From Minutes: PT30M
From Hours:   PT45H
From Days:    PT1440H
```

Beim Betrachten dieser Ausgaben könnten wir durch die Stringrepräsentation von `Duration` irritiert sein. Diese mag zunächst etwas ungewöhnlich erscheinen, ist aber logisch, wenn man den Aufbau kennt. Dieser folgt der Norm ISO 8601 und die Ausgabe startet immer mit dem Kürzel `PT`.[3] Danach gibt es Sektionen für Stunden (`H`), Minuten (`M`) und Sekunden (`S`). Sofern nötig, werden Millisekunden bzw. gar Nanonsekunden von den Sekundenwerten durch einen Punkt abgetrennt.

Betrachten wir einfache Berechnungen mit der Methode `between()`, die eine `Duration` aus der Differenz zweier `Instant`-Objekte folgendermaßen errechnen kann:

```
public static void main(final String[] args)
{
    final Instant now = Instant.now();
    final Instant silvester2013 = Instant.parse("2013-12-31T00:00:00Z");
    final Instant myBirthday2015 = Instant.parse("2015-02-07T00:00:00Z");

    // Erzeugung mit between()
    final Duration duration1 = Duration.between(now, silvester2013);
    final Duration duration2 = Duration.between(silvester2013,
                                                myBirthday2015);

    System.out.println(now + " -- " + silvester2013 + ": " + duration1);
    System.out.println(silvester2013 + " -- " + myBirthday2015 + ": " +
                       duration2);
}
```

Listing 11.5 *Ausführbar als* **'DURATIONCREATIONEXAMPLE'**

Führen wir das Programm DURATIONCREATIONEXAMPLE aus, so kommt es in etwa zu folgenden Ausgaben:

```
2017-05-23T10:03:42.816Z -- 2013-12-31T00:00:00Z: PT-29746H-3M-42.816S
2013-12-31T00:00:00Z -- 2015-02-07T00:00:00Z: PT9672H
```

Anhand der Ausgaben sehen wir, dass nicht immer alle Einzelangaben vorhanden sein müssen und auch negative Zeitdauern möglich sind. Das entsteht im Beispiel dadurch, dass die Differenz eines vergangenen Zeitpunkts mit einem danach liegenden berechnet wird.

[3] Laut `http://en.wikipedia.org/wiki/ISO_8601#Durations` ergibt sich dies aus der historischen Benennung Period, also `P`, und das `T` steht für Time.

Wissenswertes zu Berechnungen

Zusätzlich zu der zuvor gezeigten Differenzberechnung mit `between(Temporal, Temporal)` ist auch eine Addition einer durch eine `Duration` definierten Zeitspanne zu einem `Instant`-Objekt möglich. Man erhält als Ergebnis wieder ein `Instant`-Objekt. Als Beispiel soll unter anderem ausgehend vom Heiligabend, dem 24.12.2013, eine Woche in die Zukunft zum 31.12.2013, also Silvester, gesprungen werden. Auch für das Releasedatum von JDK 8, den 18.3.2014 ermitteln wir die Zeitdifferenz zum Silverster-Tag. Schließlich zeige ich zwei Additionen einer Zeitspanne von einer Woche. Wir schreiben dazu folgendes Programm:

```
public static void main(final String[] args)
{
    // Erzeugung
    final Instant christmas2013 = Instant.parse("2013-12-24T00:00:00Z");
    final Instant silvester2013 = Instant.parse("2013-12-31T00:00:00Z");
    final Instant jdk8Release = Instant.parse("2014-03-18T00:00:00Z");

    // Vergleichswerte errechnen
    System.out.println("Christmas -> Silvester:     " +
                       Duration.between(christmas2013, silvester2013));
    System.out.println("Silvester -> JDK 8 Release: " +
                       Duration.between(silvester2013, jdk8Release));

    // Berechnungen
    final Instant calcSilvester_1 = christmas2013.plus(Duration.ofDays(7));
    final Instant calcSilvester_2 = christmas2013.plus(7, ChronoUnit.DAYS);

    System.out.println(calcSilvester_1);
    System.out.println(calcSilvester_2);
}
```

Listing 11.6 *Ausführbar als* **'DURATIONCALCULATIONEXAMPLE'**

Das Programm DURATIONCALCULATIONEXAMPLE erzeugt folgende Ausgaben:

```
Christmas -> Silvester:     PT168H
Silvester -> JDK 8 Release: PT1848H
2013-12-31T00:00:00Z
2013-12-31T00:00:00Z
```

Wenn wir einige Wochen oder Monate in die Vergangenheit oder Zukunft springen wollen, dann wären dazu Methoden wie `ofWeeks(long)` bzw. `ofMonths(long)` praktisch. Diese existieren jedoch für `Instants` nicht. Während die fehlende Bereitstellung einer Methode von `ofWeeks(long)` sich noch recht gut durch eigene Berechnungen unter Zuhilfenahme von `ofDays(long)` realisieren lässt, wird dies für Monate ohne die Existenz der Methode `ofMonths(long)` schwieriger. Ein Nachbau per `ofDays(long)` wirkt unter anderem wegen unterschiedlicher Monatslängen umständlich und man entdeckt möglicherweise die Methode `plus(long, TemporalUnit)`. Diese scheint für unsere Berechnungen prädestiniert zu sein, um Wochen oder Monate in die Zukunft zu springen. Setzen wir diese Methode einfach einmal ein:

```
public static void main(final String[] args)
{
    // Erzeugung
    final Instant christmas2013 = Instant.parse("2013-12-24T00:00:00Z");
    final Instant silvester2013 = Instant.parse("2013-12-31T00:00:00Z");

    // Achtung: Duration bietet nicht ofWeeks(long) oder ofMonths(long)
    final Instant silvester_OneWeek = christmas2013.plus(1, ChronoUnit.WEEKS);
    System.out.println(silvester_OneWeek);
}
```

Listing 11.7 *Ausführbar als* **'DURATIONSPECIALEXAMPLE'**

Wenn Sie das Programm DURATIONSPECIALEXAMPLE ausführen, werden jedoch statt der gewünschten Berechnungen Exceptions folgender Form ausgelöst:

```
Exception in thread "main" java.time.temporal.UnsupportedTemporalTypeException:
      Unsupported unit: Weeks
    at java.time.Instant.plus(Instant.java:861)
```

Die Ursache liegt darin, dass zur Definition einer `Duration` keine Zeiteinheiten genutzt werden können, die sich nicht präzise durch Stunden, Minuten usw. ausdrücken lassen. Allerdings könnte man sich fragen: Wir haben doch aber eine `Duration` für die gewünschten Zeiträume basierend auf `Instant`s berechnen können. Wieso war das möglich? Die Antwort ist ganz einfach: Weil wir hier fixe Werte vorliegen haben und somit die Differenz dazwischen eindeutig zu bestimmen war. Andersherum gilt das jedoch nicht: Die abstrakte Angabe von einer Woche oder einem Monat besitzt nämlich kein exaktes Äquivalent in Form einer fixen Zeitspanne in (Milli-)Sekunden, weil ein Tag in der Regel 24 Stunden lang ist – manchmal jedoch auch 23 bzw. 25 Stunden. Selten, aber ab und an gibt es auch Schaltsekunden, die einen Tag minimal verlängern. Hier steht die (vereinfachte) Modellierung in Maschinenzeit in Konflikt mit der komplexeren Wirklichkeit, wo Tage, Wochen und Monate unterschiedlich lang sein können. Später werden wir als Abhilfe die Klasse `Period` kennenlernen.

11.1.6 Die Aufzählung `ChronoUnit`

Teilweise haben wir in den bisher gezeigten Beispielen vorgreifend die Aufzählung `java.time.temporal.ChronoUnit` genutzt, um Einheiten von Zeitdauern zu spezifizieren. In der Aufzählung `ChronoUnit` sind alle Zeiteinheiten definiert, mit denen im Date and Time API gerechnet werden kann, unter anderem Minuten, Stunden, Wochen usw. Schauen wir uns folgenden, gekürzten Auszug aus dem JDK an:

```
public enum ChronoUnit implements TemporalUnit
{
    NANOS("Nanos", Duration.ofNanos(1)),
    MICROS("Micros", Duration.ofNanos(1000)),
    MILLIS("Millis", Duration.ofNanos(1000_000)),
    SECONDS("Seconds", Duration.ofSeconds(1)),
    MINUTES("Minutes", Duration.ofSeconds(60)),
    ...
```

Tatsächlich ist die Liste der Konstanten umfangreich und reicht von Nanosekunden bis hin zu Jahrtausenden sowie Äras und es gibt sogar eine `FOREVER`-Konstante.

Greifen wir die Idee aus dem für `Instant`s vorgestellten Beispiel der Berechnung von Ankunftszeiten auf: Wir nutzen hier Instanzen von `ChronoUnit`, um die Zeitdauer in verschiedenen Varianten (Stunden und Minuten) darzustellen. Ebenso wie für `Duration` gibt es auch für `ChronoUnit` eine Methode `between(Temporal, Temporal)`, mit der sich die Differenz zwischen zwei Zeitpunkten bestimmen lässt:

```
public static void main(final String[] args)
{
    // Abfahrt jetzt und Reisedauer 5 Stunden
    final Instant departureTime = Instant.now();
    final Instant arrivalTime = departureTime.plus(5, ChronoUnit.HOURS);

    System.out.println("departure now:    " + departureTime);
    System.out.println("arrival now + 5h: " + arrivalTime);

    // Berechnungen durchführen: Differenz bilden
    final long inBetweenHours = ChronoUnit.HOURS.between(departureTime,
                                                         arrivalTime);
    final long inBetweenMinutes = ChronoUnit.MINUTES.between(departureTime,
                                                             arrivalTime);

    System.out.println("inBetweenHours:   " + inBetweenHours);
    System.out.println("inBetweenMinutes: " + inBetweenMinutes);
}
```

Listing 11.8 *Ausführbar als '***CHRONOUNITEXAMPLE***'*

Führt man das Programm CHRONOUNITEXAMPLE aus, so erhält man in etwa folgende Ausgaben auf der Konsole, die sehr schön die Berechnungen von 5 Stunden in die Zukunft sowie die Differenzbildung zwischen zwei Zeitpunkten in verschiedenen Zeiteinheiten (Stunden und Minuten) zeigen:

```
departure now:    2014-02-19T22:13:50.691Z
arrival now + 5h: 2014-02-20T03:13:50.691Z
inBetweenHours:   5
inBetweenMinutes: 300
```

11.1.7 Die Klassen `LocalDate`, `LocalTime` und `LocalDateTime`

Wie eingangs erwähnt, hat die Darstellung von Zeitangaben in Millisekunden, die sehr hilfreich für die Verarbeitung mit Computern ist, recht wenig mit der menschlichen Denkweise und Orientierung im Zeitsystem zu tun. Menschen denken bevorzugt in Zeitabschnitten oder wiederkehrenden Datumsangaben, etwa 24.12. für Heiligabend, 31.12. für Silvester usw., also Datumsangaben ohne Uhrzeit und Jahr. Manchmal benötigt man »unvollständige Zeitangaben«, wie Uhrzeiten ohne Bezug zu einem Datum, etwa 18.00 Uhr Feierabend, oder als Kombination: dienstags und donnerstags 19 Uhr

Karate-Training.[4] Wollten wir so etwas mit dem bisher existierenden API ausdrücken, wäre das recht schwierig. Schauen wir uns nun die neuen Möglichkeiten an.

Die Klasse `java.time.LocalDate` repräsentiert eine reine Datumsangabe, also eine Kombination aus Jahr, Monat und Tag ohne Zeitinformationen. Mit der Klasse `java.time.LocalTime` wird eine Zeitangabe ohne Datumsangabe modelliert, z. B. 18:00 Uhr. Die Klasse `java.time.LocalDateTime` ist eine Kombination aus beiden. Folgendes Programm zeigt die Klassen und Berechnungen. Zur Konstruktion sehen wir jeweils `of`-Methoden und danach verschiedene `plusXYZ()`- sowie `minusXYZ()`- bzw. `getXYZ()`-Methoden:

```
public static void main(final String[] args)
{
    // LocalDate
    final LocalDate michasBirthday = LocalDate.of(1971, Month.FEBRUARY, 7);
    final LocalDate barbarasBirthday = michasBirthday.plusYears(2).
                                                      plusMonths(1).
                                                      plusDays(17);

    System.out.println("michasBirthday:    " + michasBirthday);
    System.out.println("barbarasBirthday:  " + barbarasBirthday);

    // LocalTime
    final LocalTime atTen = LocalTime.of(10,00,00);
    final LocalTime tenFifteen = atTen.plusMinutes(15);
    final LocalTime breakfastTime = tenFifteen.minusHours(2);
    System.out.println("\natTen:             " + atTen);
    System.out.println("tenFifteen:        " + tenFifteen);
    System.out.println("breakfastTime:     " + breakfastTime);

    // LocalDateTime
    final LocalDateTime jdk8Release = LocalDateTime.of(2014, 3, 18, 8, 30);
    System.out.println("\njdk8Release:       " + jdk8Release);
    System.out.printf("jdk8Release:  %s.%s.%s\n", jdk8Release.getDayOfMonth(),
                                                  jdk8Release.getMonthValue(),
                                                  jdk8Release.getYear());
}
```

Listing 11.9 *Ausführbar als* **'LOCALDATEANDTIMEEXAMPLE'**

Wir sehen, wie sich Berechnungen mithilfe von `plusXYZ()`- sowie `minusXYZ()`-Methoden einfach und sprechend ausdrücken lassen. Führt man das Programm LOCALDATEANDTIMEEXAMPLE aus, so erhält man folgende Ausgaben:

```
michasBirthday:    1971-02-07
barbarasBirthday:  1973-03-24

atTen:             10:00
tenFifteen:        10:15
breakfastTime:     08:15

jdk8Release:       2014-03-18T08:30
jdk8Release:  18.3.2014
```

[4] Insbesondere interessiert uns dabei in der Regel nicht die Zeitzone, in der die Termine stattfinden – mit Ausnahme von Telefonterminen etwa mit Geschäftspartnern in Übersee.

11.1.8 Die Klasse `Period`

Ähnlich wie die Klasse `Duration` implementiert auch die Klasse `java.time.Period` das Interface `TemporalAmount` und modelliert ebenfalls einen Zeitabschnitt. Beispiele sind etwa »2 Monate« oder »3 Tage«. Diese Art der Darstellung ist oftmals einfacher zu handhaben als eine korrespondierende Repräsentation in Nano- oder Millisekunden. Konstruieren wir ein paar Instanzen von `Period`:

```
public static void main(final String[] args)
{
    // Erzeuge ein Period-Objekt mit 1 Jahr, 6 Monaten und 3 Tagen
    final Period oneYear_sixMonths_ThreeDays = Period.ofYears(1).withMonths(6).
                                                      withDays(3);
    // Chaining von of() arbeitet anders, als man es eventuell erwartet!
    // Ergibt ein Period-Objekt mit 3 Tagen statt 2 Monate, 1 Woche und 3 Tagen
    final Period twoMonths_OneWeek_ThreeDays = Period.ofMonths(2).ofWeeks(1).
                                                       ofDays(3);

    final Period twoMonths_TenDays = Period.ofMonths(2).withDays(10);
    final Period sevenWeeks = Period.ofWeeks(7);
    final Period threeDays = Period.ofDays(3);

    System.out.println("1 year 6 months ...:  " + oneYear_sixMonths_ThreeDays);
    System.out.println("Surprise just 3 days: " + twoMonths_OneWeek_ThreeDays);
    System.out.println("2 months 10 days:     " + twoMonths_TenDays);
    System.out.println("sevenWeeks:           " + sevenWeeks);
    System.out.println("threeDays:            " + threeDays);
}
```

Listing 11.10 *Ausführbar als* **'PERIODEXAMPLE'**

Startet man das Programm PERIODEXAMPLE, so wird Folgendes ausgegeben:

```
1 year 6 months ...:  P1Y6M3D
Surprise just 3 days: P3D
2 month 10 days:      P2M10D
sevenWeeks:           P49D
threeDays:            P3D
```

Anhand des Beispiels und dessen Ausgaben lernen wir einiges über die Klasse `Period`: Zunächst ist da wieder die etwas kryptische Stringrepräsentation, die der ISO 8601 folgt. Dabei ist `P` das Startkürzel (für Period) und dann stehen `Y` für Jahre, `M` für Monate und `D` für Tage. Als Besonderheit gibt es zum einen noch die Umrechnung für Wochen: `P14D` steht für 2 Wochen und könnte durch `Period.ofWeeks(2)` erzeugt werden. Zum anderen sind auch negative Offsets erlaubt, etwa `P-2M4D`.

Neben diesen Details der Ausgabe sehen wir, dass sich Aufrufe von `of()` hintereinander ausführen lassen – es gewinnt aber der zuletzt aufgerufene.[5] Man kann also auf diese Weise keine Zeiträume kombinieren, sondern legt einen initialen Zeitraum fest. Sollen weitere Zeitabschnitte hinzugefügt werden, so muss man dafür verschiedene `with()`-Methoden nutzen. Dabei wird ein Implementierungsdetail sichtbar: Die Klasse `Period` verwaltet drei Einzelwerte, nämlich für Jahre, Monate und Tage, aber

[5] Dass dies problematisch ist, könnte man daran erkennen, dass diese Methoden statisch sind.

eben nicht für Wochen. Somit gibt es auch keine Methode `withWeeks()`, sondern nur eine `ofWeeks()`, die intern eine Umrechnung in Tage vornimmt. Das hätten wir schon anhand der Ausgabe vermuten können.

Wir schauen uns nun an, wie einfach und lesbar Berechnungen gestaltet werden können: Ausgehend vom 7.2.1971 10:11 springen wir 31 Tage und zum Vergleich vier Wochen sowie einen Monat in die Zukunft:

```
public static void main(final String[] args)
{
    final LocalDateTime start = LocalDateTime.of(1971, 2, 7, 10, 11);

    final Period thirtyOneDays = Period.ofDays(31);
    final Period fourWeeks = Period.ofWeeks(4);
    final Period oneMonth = Period.ofMonths(1);

    System.out.println("7.2.1971 + 31 Tage:  " + start.plus(thirtyOneDays));
    System.out.println("7.2.1971 + 4 Wochen: " + start.plus(fourWeeks));
    System.out.println("7.2.1971 + 1 Monat:  " + start.plus(oneMonth));
}
```

Listing 11.11 *Ausführbar als* **'PERIODCALCULATIONEXAMPLE'**

Das Programm PERIODCALCULATIONEXAMPLE erzeugt folgende Ausgaben:

```
7.2.1971 + 31 Tage:  1971-03-10T10:11
7.2.1971 + 4 Wochen: 1971-03-07T10:11
7.2.1971 + 1 Monat:  1971-03-07T10:11
```

11.1.9 Die Klasse `ZonedDateTime`

Neben der bereits vorgestellten Klasse `LocalDateTime` zur Repräsentation von Datum und Uhrzeit ohne Zeitzonenbezug existiert eine korrespondierende Klasse `java.time.ZonedDateTime`. Diese besitzt eine zugeordnete Zeitzone und berücksichtigt bei Berechnungen nicht nur die Zeitzone, sondern auch die Auswirkungen von Winter- und Sommerzeit. Um die aktuelle Zeit als `ZonedDateTime` zu ermitteln, kann man die Methode `now()` nutzen. Es existieren weitere Methoden, etwa um die Zeitzone und andere Werte abzufragen bzw. Instanzen von `ZonedDateTime` mit geänderter Wertebelegung durch Aufruf von `withXYZ()`-Methoden zu erzeugen. Interessant und etwas schade ist, dass man beim Aufruf von `withMonth(int)` einen `int`-Wert und keine Monatskonstante übergeben muss. Zur besseren Lesbarkeit empfiehlt sich, die Konstanten trotzdem zu verwenden und durch Aufruf der Methode `getValue()` auf deren `int`-Wert zuzugreifen. Das habe ich im Listing fett markiert.

Nachfolgend sind verschiedene Beispiele für Berechnungen mit der Klasse `ZonedDateTime` gezeigt, im Speziellen auch ein Wechsel der Zeitzone:

```
public static void main(final String[] args)
{
    // Aktuelle Zeit als ZonedDateTime-Objekt ermitteln
    final ZonedDateTime now = ZonedDateTime.now();

    // Die Uhrzeit ändern und in neuem Objekt speichern
    final ZonedDateTime nowButChangedTime = now.withHour(11).withMinute(44);

    // Neues Objekt mit verändertem Datum erzeugen
    final ZonedDateTime dateAndTime = nowButChangedTime.withYear(2008).
                                                        withMonth(9).
                                                        withDayOfMonth(29);

    // Einsatz einer Monatskonstanten und wechseln der Zeitzone
    final ZonedDateTime dateAndTime2 = nowButChangedTime.withYear(2008).
                                       withMonth(Month.SEPTEMBER.getValue()).
                                       withDayOfMonth(29).
                                       withZoneSameInstant(ZoneId.of("GMT"));

    System.out.println("now:          " + now);
    System.out.println("-> 11:44:     " + nowButChangedTime);
    System.out.println("-> 29.9.2008: " + dateAndTime);
    System.out.println("-> 29.9.2008: " + dateAndTime2);
}
```

Listing 11.12 *Ausführbar als '*ZONEDDATETIMEEXAMPLE*'*

Führt man das Programm ZONEDDATETIMEEXAMPLE aus, so kommt es zu den folgenden Ausgaben. Diese zeigen insbesondere den Einfluss von Winter- und Sommerzeit, wodurch im September 2008 die Abweichung von +02:00 angegeben wird. Worauf sich dies bezieht, erkennt man dann durch den Wechsel der Zeitzone auf GMT:

```
now:          2015-05-17T20:27:54.214+02:00[Europe/Zurich]
-> 11:44:     2015-05-17T11:44:54.214+02:00[Europe/Zurich]
-> 29.9.2008: 2008-09-29T11:44:54.214+02:00[Europe/Zurich]
-> 29.9.2008: 2008-09-29T09:44:54.214Z[GMT]
```

11.1.10 Zeitzonen und die Klassen `ZoneId` und `ZoneOffset`

Wir haben bereits einiges an Wissen über das neue Date and Time API erworben. Zeitzonen wurden bislang eher am Rande betrachtet. Wenn Sie allerdings bei der Datumsarithmetik Zeitzonen beachten müssen, dann helfen dabei seit JDK 8 die Klassen `ZoneId`, `ZoneOffset` sowie die gerade vorgestellte Klasse `ZonedDateTime` aus dem Package `java.time`.

Die Klasse `ZoneId`

Nun wollen wir die Verarbeitung von Zeitzonen anhand eines Beispiels kennenlernen. Zunächst ermitteln wir basierend auf einigen textuellen Zeitzonenkennungen durch Aufruf von `ZoneId.of(String)` die zugehörige `ZoneId`-Instanz und konstruieren

daraus jeweils ein `LocalTime`-Objekt. Dann rufen wir `ZoneId.getAvailableZoneIds()` auf und erhalten so alle verfügbaren Zeitzonen. Mithilfe von Streams und den beiden Methoden `filter()` und `limit()` finden wir drei Kandidaten aus Europa. Das Ganze realisieren wir wie folgt:

```
public static void main(final String[] args)
{
    final Stream<String> zoneIdNames = Stream.of("Asia/Chungking",
                                                 "Africa/Nairobi",
                                                 "America/Los_Angeles");

    zoneIdNames.forEach(zoneIdName ->
    {
        final ZoneId zoneId = ZoneId.of(zoneIdName);
        final LocalTime now = LocalTime.now(zoneId);

        System.out.println(zoneIdName + ": " + now);
    });

    final Set<String> allZones = ZoneId.getAvailableZoneIds();
    final Predicate<String> inEurope = name -> name.startsWith("Europe/");
    final List<String> threeFromEurope = allZones.stream().
                                                  filter(inEurope).limit(3).
                                                  collect(Collectors.toList());

    System.out.println("\nSome timezones in europe:");
    threeFromEurope.forEach(System.out::println);
}
```

Listing 11.13 *Ausführbar als* **'ZONEIDEXAMPLE'**

Das Programm ZONEIDEXAMPLE produziert folgende Ausgaben:

```
Asia/Chungking: 20:10:22.219
Africa/Nairobi: 15:10:22.220
America/Los_Angeles: 05:10:22.222

Some timezones in europe:
Europe/London
Europe/Brussels
Europe/Warsaw
```

Im obigen Listing sehen wir die Verarbeitung von Zeitzonen basierend auf der Klasse `ZoneId`. Etwas unschön ist, dass im JDK statt Konstanten lediglich einfache Strings als IDs genutzt werden. Praktischerweise erhält man die Menge der gültigen Zeitzonen-IDs durch Aufruf von `ZoneId.getAvailableZoneIds()`.

Die Klasse `ZoneOffset`

Auf die Klasse `ZoneOffset` zur Modellierung des Offsets zur GMT (Greenwich Mean Time) (die auch – wenn auch formal nicht ganz korrekt – als UTC (Universal Coordinated Time) bezeichnet wird) möchte ich ganz kurz anhand eines kleinen Beispiels eingehen. Startend beim jetzigen Zeitpunkt `LocalDateTime.now()` berechnen wir für unterschiedliche Zeitzonen ein zugehöriges `ZonedDateTime`-Objekt und geben dessen Offset zur GMT aus:

```
public static void main(final String[] args)
{
    final Stream<String> zoneIdNames = Stream.of("Europe/Berlin",
                                                 "America/Los_Angeles",
                                                 "Australia/Adelaide");

    zoneIdNames.forEach(zoneIdName ->
    {
        final ZoneId zoneId = ZoneId.of(zoneIdName);
        final LocalDateTime ldt = LocalDateTime.now();
        final ZonedDateTime zdt = ldt.atZone(zoneId);
        final ZoneOffset offset = zdt.getOffset();

        System.out.format("offset for 'zoneId' is %s\n", zoneId, offset);
    });
}
```

Listing 11.14 *Ausführbar als '***ZONEOFFSETEXAMPLE***'*

Führen wir das Programm ZONEOFFSETEXAMPLE aus, erhalten wir folgende Ausgaben, mit deren Hilfe wir die zu den Zeitzonen korrespondierenden Offsets erfahren:

```
offset for 'Europe/Berlin' is +02:00
offset for 'America/Los_Angeles' is -07:00
offset for 'Australia/Adelaide' is +10:30
```

11.1.11 Die Klasse `Clock`

In einigen technischen Anwendungsfällen benötigt man Zugriff auf Millisekundenangaben. Früher hat man dann Aufrufe von `System.currentTimeMillis()` genutzt, um die aktuelle Zeit in Millisekunden seit dem 1.1.1970 zu ermitteln. Nun kann man dazu die Klasse `java.time.Clock` verwenden und schreibt etwa Folgendes:

```
public static void main(final String[] args)
{
    printClockAndMillis(Clock.systemUTC()); // Basis UTC
    printClockAndMillis(Clock.systemDefaultZone()); // Basis Default-Zeitzone
}

private static void printClockAndMillis(final Clock clock)
{
    System.out.println(clock + " / ms: " + clock.millis());
}
```

Listing 11.15 *Ausführbar als '***FIRSTCLOCKEXAMPLE***'*

Das Programm FIRSTCLOCKEXAMPLE gibt etwa Folgendes aus:

```
SystemClock[Z] / ms: 1430766415687
SystemClock[Europe/Berlin] / ms: 1430766415745
```

Wenn man genau hinschaut, sieht man eine minimale Differenz in den Millisekunden. Das liegt vor allem daran, dass die Methode `millis()` immer bezogen auf die Zeitzone GMT bzw. UTC arbeitet.

Es ist nicht direkt ein wirklicher Mehrwert dieser Klasse sichtbar. Dieser liegt darin, dass die Klasse `Clock` als Taktgeber für viele Zeitklassen genutzt werden kann: Dazu besitzen diverse Klassen neben der Methode `now()` auch eine Methode `now(Clock)`, der man eine alternative `Clock`-Instanz übergeben kann, etwa wie folgt:

```
final LocalDateTime now = LocalDateTime.now(mySpecialClock);
```

Wozu könnte das sinnvoll einsetzbar sein? Beispielsweise lassen sich für Unit Tests fixe Zeitangaben realisieren. Schauen wir zum Verständnis auf folgendes Beispiel, das durch Aufruf von `fixed()` eine sich nicht verändernde `Clock` erzeugt:

```
public static void main(final String[] args) throws InterruptedException
{
   final Clock clock1 = Clock.systemUTC();
   final Clock clock2 = Clock.systemDefaultZone();
   final Clock clock3 = Clock.fixed(Instant.now(), ZoneId.of("Asia/Hong_Kong"));

   printClocks(clock1, clock2, clock3);

   Thread.sleep(10_000);
   System.out.println("\nAfter 10 s\n");

   printClocks(clock1, clock2, clock3);
}

private static void printClocks(final Clock... clocks)
{
   for (final Clock clock : clocks)
   {
      System.out.println("LocalTime: " + LocalTime.now(clock));
   }
}
```

Listing 11.16 *Ausführbar als* **'SECONDCLOCKEXAMPLE'**

Führen wir das Programm SECONDCLOCKEXAMPLE aus, so zeigt sich im Gegensatz zum Aufruf von `millis()`, dass die mithilfe einer `Clock` erzeugten Instanzen die Zeitzonen berücksichtigen. Es wird auch deutlich, dass die Wartezeit von 10 Sekunden für die ersten beiden `Clock`s jeweils eine Änderung von 10 Sekunden bewirkt. Die FixedClock verändert sich dagegen nicht:

```
LocalTime: 20:48:10.936
LocalTime: 22:48:10.937
LocalTime: 04:48:10.936

After 10 s

LocalTime: 20:48:20.948
LocalTime: 22:48:20.948
LocalTime: 04:48:10.936
```

Wie dieses Beispiel zeigt, wird es mit einer speziellen `Clock` möglich, die Datumsarithmetik unter anderen Gegebenheiten, etwa veränderter Zeitzone, zu testen, ohne dass man dazu die Systemeinstellungen verändern müsste. Gerade für Unit Tests kann dies nützlich sein, um definierte Ausgangssituationen zu erhalten.

11.1.12 Formatierung und Parsing

Die mit JDK 8 eingeführte Klasse `java.time.format.DateTimeFormatter` macht die formatierte Ausgabe und das Parsing von Datumswerten einfach. Neben diversen vordefinierten Formaten kann man nahezu beliebige eigene Formatierungsvarianten bereitstellen. Das wird im folgenden Listing gezeigt:

```
import static java.time.format.DateTimeFormatter.*;
import static java.time.format.FormatStyle.SHORT;

public static void main(final String[] args)
{
    // Definition einiger spezieller Formatter
    final DateTimeFormatter ddMMyyyyFormat = ofPattern("dd.MM.yyyy");
    final DateTimeFormatter italian_dMMMMyy = ofPattern("d.MMMM y",
                                                        Locale.ITALIAN);
    final DateTimeFormatter shortGerman = ofLocalizedDateTime(SHORT).
                                              withLocale(Locale.GERMAN));

    // Achtung: Die textuellen Teile sind in Hochkomma einzuschließen
    final String customPattern = "'Der 'dd'. Tag im 'MMMM' im Jahr 'yy'.'";
    final DateTimeFormatter customFormat = ofPattern(customPattern);

    // Mapping für verschiedene Formate definieren
    final Map<String, DateTimeFormatter> formatters = new LinkedHashMap<>();
    formatters.put("BASIC_ISO_DATE", BASIC_ISO_DATE);
    formatters.put("ISO_DATE_TIME", ISO_DATE_TIME);
    formatters.put("dd.MM.yyyy", ddMMyyyyFormat);
    formatters.put("d.MMMM y (it)", italian_dMMMMyy);
    formatters.put("SHORT_GERMAN", shortGerman);
    formatters.put("CUSTOM_FORMAT", customFormat);

    System.out.println("Formatting:\n");
    applyFormatters(LocalDateTime.of(2014, MAY, 28, 1, 2, 3), formatters);

    // Parsen von Datumswerten
    System.out.println("\nParsing:\n");

    final LocalDate fromIsoDate = LocalDate.parse("1971-02-07");
    final LocalDate fromCustomPattern = LocalDate.parse("18.03.2014",
                                                        ddMMyyyyFormat);
    final LocalDateTime fromShortGerman = LocalDateTime.parse("18.03.14 11:12",
                                                  shortGerman);

    System.out.println("From ISO Date:       " + fromIsoDate);
    System.out.println("From custom pattern: " + fromCustomPattern);
    System.out.println("From short german:   " + fromShortGerman);
}

private static void applyFormatters(final LocalDateTime base,
                        final Map<String, DateTimeFormatter> formatters)
{
    System.out.println("Starting on: " + base);
    formatters.forEach((name, formatter) ->
    {
        System.out.println("applying '" + name + "': " +
                                      base.format(formatter));
    });
}
```

Listing 11.17 *Ausführbar als '***FORMATTINGANDPARSINGEXAMPLE***'*

Das Programm FORMATTINGANDPARSINGEXAMPLE produziert folgende Ausgaben, die ein erstes Gefühl für die Formatierung und das Parsing vermitteln:

```
Formatting:

Starting on: 2014-05-28T01:02:03
applying 'BASIC_ISO_DATE': 20140528
applying 'ISO_DATE_TIME': 2014-05-28T01:02:03
applying 'dd.MM.yyyy': 28.05.2014
applying 'd.MMMM y (it)': 28.maggio 2014
applying 'SHORT_GERMAN': 28.05.14 01:02
applying 'CUSTOM_FORMAT': Der 28. Tag im Mai im Jahr 14.

Parsing:

From ISO Date:       1971-02-07
From custom pattern: 2014-03-18
From short german:   2014-03-18T11:12
```

Es gibt eine Vielzahl weiterer Möglichkeiten, die Klasse `DateTimeFormatter` zu nutzen, die hier nicht alle vorgestellt werden können. Deshalb ist ein Blick auf die ausführliche Onlinedokumentation des JDKs lohnenswert.

11.2 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);
}
```