Date-Time-API

Seit Java 8 gibt es ein neues Paket `java.time`, das alle bisherigen Java-Typen rund um Datum- und Zeitverarbeitung überflüssig macht. Mit anderen Worten: Mit den neuen Typen lassen sich `Date`, `Calendar`, `GregorianCalendar`, `TimeZone` usw. streichen und ersetzen. Natürlich gibt es Adapter zwischen den APIs, doch gibt es nur noch sehr wenige zwingende Gründe, heute bei neuen Programmen auf die älteren Typen zurückzugreifen – ein Grund ist natürlich die heilige Kompati-bilität.

Die neue API basiert auf dem standardisierten Kalendersystem von ISO-8601, und das deckt ab, wie ein Datum, wie Zeit, Datum und Zeit, UTC, Zeitintervalle (Dauer/Zeitspanne) und Zeitzonen repräsentiert werden. Die Implementierung basiert auf dem gregorianischen Kalender, wobei auch andere Kalendertypen denkbar sind. Javas Kalendersystem greift auf andere Standards bzw. Implementierungen zurück, unter anderem auf das Unicode Common Locale Data Repository (CLDR) zur Lokalisierung von Wochentagen oder die Time-Zone Database (TZDB), die alle Zeitzonenwechsel seit 1970 dokumentiert. In Java nutzen die XML-APIs schon länger ISO-8601-Kalender, denn Schema-Dateien nutzen einen `XMLGregorianCalendar`, und selbst für Dauern gibt einen eigenen Typ `Duration`.

Geschichte

Über die alten Datumsklassen meckert die Java-Community seit über zehn Jahren; nicht ganz zu Unrecht, da ein `Date` zum Beispiel ein Date-Time ist, Kalender fehlen, die Sommerzeitumstellung verschiedener Länder nicht korrekt behandelt wird und wegen weiterer Schwächen.[1] Daher geht die Entwicklung der Date-Time-API lange zurück und basiert auf Ideen von Joda-Time (*http://joda-time.sourceforge.net/*), einer populären quelloffenen Bibliothek. Spezifiziert im JSR-310 (eingereicht am 30. Jan 2007)[2] und angedacht für Java 7 (was vier Jahre später, im Juli 2011 kam) wurde die API erst in Java 8 Teil der Java SE. Für Java 1.7 gibt es einen Back-Port (*https://github.com/ThreeTen/threetenbp*), um später leicht die Codebasis auf Java 8 zu migrieren, der durchaus inte-ressant ist.

Erster Überblick

Die zentralen temporalen Typen aus der Date-Time-API sind schnell dokumentiert:

Typ	Beschreibung	Feld(er)
`LocalDate`	Repräsentiert ein übliches Datum.	Jahr, Monat, Tag
`LocalTime`	Repräsentiert eine übliche Zeit.	Stunden, Minuten, Sekunden, Nanosek.
`LocalDateTime`	Kombination aus Datum und Zeit	Jahr, Monat, Tag, Stunden, Minuten, -Sekunden, Nanosek.
`Period`	Dauer zwischen zwei `LocalDate`s	Jahr, Monat, Tag
`Year`	nur Jahr	Jahr
`Month`	nur Monat	Monat
`MonthDay`	nur Monat und Tag	Monat, Tag
`OffsetTime`	Zeit mit Zeitzone	Stunden, Minuten, Sekunden, Nanosek., Zonen-Offset
`OffsetDateTime`	Datum und Zeit mit Zeitzone als UTC-Offset	Jahr, Monat, Tag, Stunden, Minuten, -Sekunden, Nanosek., Zonen-Offset

[1] Der Quellcode stammt von IBM. Trifft Oracle jetzt die Schuld, weil Sun die Implementierung damals übernahm? Siehe zu den Kritiken auch *http://tutego.de/go/dategotchas*.

[2] *https://jcp.org/en/jsr/detail?id=310*

`ZonedDateTime`	Datum und Zeit mit Zeitzone als ID und Offset	Jahr, Monat, Tag, Stunden, Minuten, -Sekunden, Nanosek., Zonen-Info
`Instant`	fortlaufende Maschinenzeit	Nanosekunden
`Duration`	Zeitintervall zwischen zwei `Instant`s	Sekunden/Nanosek.

*Tabelle **Fehler! Kein Text mit angegebener Formatvorlage im Dokument.**.1: Alle temporalen Klassen aus java.time*

1.1.1 Menschenzeit und Maschinenzeit

Datum und Zeit, die wir als Menschen in Einheiten wie Tagen und Minuten verstehen, nennen wir Menschenzeit (engl. *human time*), die fortlaufende Zeit des Computers, die eine Auflösung im Nanosekundenbereich hat, Maschinenzeit. Die Maschinenzeit startet dabei von einer Zeit, die wir Epoche nennen.

Aus Tabelle **Fehler! Kein Text mit angegebener Formatvorlage im Dokument.**.1 lässt sich gut ablesen, dass die meisten Klassen für uns Menschen gemacht sind, und sich nur `Instant`/`Duration` auf die Maschinenzeit bezieht. `LocalDate`, `LocalTime` und `Local-DateTime` repräsentieren Menschenzeit ohne Bezug zu einer Zeitzone, `ZonedDateTime` mit Zeit-zone. Bei der Auswahl der richtigen Zeitklassen für eine Aufgabenstellung ist das natürlich die erste Überlegung, ob die Menschenzeit oder die Maschinenzeit repräsentiert werden soll, dann was genau für Felder nötig sind und ob eine Zeitzone relevant ist oder nicht. Soll zum Beispiel die Ausführungszeit gemessen werden, ist es unnötig, zu wissen, an welchem Datum die Messung begann und endet, hier ist `Duration` korrekt, nicht `Period`.

Beispiel

```
LocalDate now = LocalDate.now();
System.out.println( now ); // 2014-01-30
System.out.printf( "%d. %s %d", now.getDayOfMonth(), now.getMonth(), now.getYear() );// 3
0. JANUARY 2014
LocalDate bdayMLKing = LocalDate.of( 1929, Month.JANUARY, 15 );
DateTimeFormatter formatter = DateTimeFormatter.ofPattern( "d. MMMM yyyy" );
System.out.println( formatter.format( bdayMLKing ) ); // 15. Januar 1929
```

Alle Klassen basieren standardmäßig auf dem ISO-System, andere Kalendersysteme, wie der -japanische Kalender, werden über Typen aus `java.time.chrono` erzeugt, natürlich sind auch ganz neue Systeme möglich.

Beispiel

```
ChronoLocalDate now = JapaneseChronology.INSTANCE.dateNow();
System.out.println( now );                  // Japanese Heisei 26-01-30
```

Paketübersicht

Die Typen der Date-Time-API verteilen sich auf verschiedene Pakete:

- `java.time`: Enthält die Standardklassen wie `LocalTime`, `Instant` usw. Alle Typen basieren auf dem Kalendersystem ISO-8601, das landläufig unter gregorianischer Kalender bekannt ist. Der wird erweitert zum sogenannten Proleptic Gregorian Kalender, das ist ein gregorianischer Kalender, der auch für die Zeit vor 1582 (der Einführung des Kalenders) gültig ist, damit eine konsistente Zeitlinie entsteht.
- `java.time.chrono`: Hier befinden sich vorgefertigte alternative (also Nicht-ISO-) Kalender-systeme, wie japanischer Kalender, Thai-Buddhist-Kalender, islamischer Kalender (genannt Hijrah) und ein paar weitere.
- `java.time.format`: Klassen zum Formatieren und Parsen von Datum- und Zeit, wie der genannte `DateTimeFormatter`

- `java.time.zone`: Unterstützende Klassen für Zeitzonen, etwa `ZonedDateTime`
- `java.time.temporal`: Tiefer liegende API, die Zugriff und Modifikation einzelner Felder eines Datums/Zeitwerts erlaubt

Design-Prinzipen

Bevor wir uns mit den einzelnen Klassen auseinandersetzen, wollen wir uns mit den Design-Prinzipien beschäftigen, denn alle Typen der Date-Time-API folgen wiederkehrenden Mustern. Die erste und wichtigste Eigenschaft ist, dass alle Objekte *immutable* sind, also nicht veränderbar. Das ist bei der „alten" API anders, `Date` und die `Calendar`-Klassen sind veränderbar, mit teils verheerenden Folgen; denn werden diese Objekte rumgereicht und verändert, kann es zu unkalkulierbaren Seiteneffekten kommen. Die Klassen der neuen Date-Time-API sind immutable, und so stehen die Datums-/Zeit-Klassen wie `LocalTime` oder `Instant` den veränderbaren Typen wie `Date` oder `Calendar` gegenüber. Alle Methoden, die nach Änderung aussehen, erzeugen neue Objekte mit den gewünschten Änderungen. Seiteneffekte bleiben also aus, und alle Typen sind threadsicher.

Unveränderbarkeit ist eine Designeigenschaft wie auch die Tatsache, dass `null` nicht als Argument erlaubt wird. In der Java-API wird oftmals `null` akzeptiert, weil es etwas Optionales ausdrückt, doch die Date-Time-API straft dies in der Regel mit einer `NullPointerExcpetion`. Dass `null` nicht als Argument und nicht als Rückgabe im Einsatz ist, kommt einer weiteren Eigenschaft zugute: Die API gestattet „flüssige" Ausdrücke, also kaskadierte Aufrufe, da viele Methoden die `this`-Referenz zurückgeben, so wie das auch von `StringBuilder` bekannt ist.

Zu diesen eher technischen Eigenschaften kommt eine neue Namensgebung hinzu, die sich von der Namensgebung der bekannten JavaBeans absetzt. So gibt es keine Konstruktoren und keine Setter (das brauchen die immutablen Klassen nicht), sondern neue Muster, die viele der neuen Typen aus der Date-Time-API einhalten:

Methode	**Klassen-/Exemplarmethode**	**Grundsätzliche Bedeutung**
`now()`	statisch	Liefert Objekt mit aktueller Zeit/aktuellem Datum.
`ofXXX()`	statisch	Erzeugt neue Objekte.
`fromXXX()`	statisch	Erzeugt neue Objekte aus anderen Repräsentationen.
`parseXXX()`	statisch	Erzeugt neues Objekt aus einer String-Repräsentation.
`format()`	Exemplar	Formatiert und liefert einen String.
`getXXX()`	Exemplar	Liefert Felder eines Objekts.
`isXXX()`	Exemplar	Fragt Status eines Objekts ab.
`withXXX()`	Exemplar	Liefert Kopie des Objekts mit einer geänderten -Eigenschaft.
`plusXXX()`	Exemplar	Liefert Kopie des Objekts mit einer aufsummierten -Eigenschaft.
`minusXXX()`	Exemplar	Liefert Kopie des Objekts mit einer reduzierten -Eigenschaft.
`toXXX()`	Exemplar	Konvertiert Objekt in neuen Typ.
`atXXX()`	Exemplar	Kombiniert dieses Objekt mit einem anderen Objekt.
`XXXInto()`	Exemplar	Kombiniert eigenes Objekt mit einem anderen Ziel-objekt.

*Tabelle **Fehler! Kein Text mit angegebener Formatvorlage im Dokument.**.2: Namensmuster in der Date-Time-API*

Die Methode `now()` haben wir schon in den ersten Beispielen verwendet, sie liefert zum Beispiel das aktuelle Datum. Weitere Erzeugermethoden sind die mit dem Präfix `of`, `from` oder `with`; Konstruktoren gibt es nicht. `withXXX()`-Methoden nehmen die Rolle der Setter ein.

1.1.2 Datumsklasse LocalDate

Ein Datum (ohne Zeitzone) repräsentiert die Klasse `LocalDate`. Damit lässt sich zum Beispiel ein Geburtsdatum repräsentieren.

Ein temporales Objekt kann über die statischen `of(…)`-Fabrikmethoden aufgebaut oder von einem anderen Objekt abgeleitet werden. Interessant sind die Methoden, die mit einem `TemporalAdjuster` arbeiten.

Beispiel

```
LocalDate today = LocalDate.now();
LocalDate nextMonday = today.with( TemporalAdjusters.next( DayOfWeek.SATURDAY ) );
System.out.printf( "Heute ist der %s, und frei ist am Samstag, den %s",
                   today, nextMonday );
```

Mit den Objekten in der Hand können wir diverse Getter nutzen und einzelne Felder erfragen, etwa `getDayOfMonth()`, `getDayOfYear()` (liefern `int`) oder `getDayOfWeek()`, das eine Aufzählung vom Typ `DayOfWeek` liefert, und `getMonth()`, das eine Aufzählung vom Typ `Month` liefert. Dazu kommen Methoden, die mit `minusXXX(…)` oder `plusXXX(…)` neue `LocalDate`-Objekte liefern, wenn zum Beispiel mit `minusYear(long yearsToSubtract)` eine Anzahl Jahre zurückgelaufen werden soll. Durch die Negation des Vorzeichens kann auch die jeweils entgegengesetzte Methode genutzt werden, sprich `LocalDate.now().minusMonths(1)` kommt zum gleichen Ergebnis wie `LocalDate.now().plusMonths(-1)`. Die `withXXX(…)`-Methoden belegen ein Feld neu und liefern ein modifiziertes neues `LocalDate`-Objekt.

Von einem `LocaleDate` lassen sich andere temporale Objekte bilden, `atTime(…)` etwa liefert `LocalDateTime`-Objekte, bei denen gewisse Zeit-Felder belegt sind. `atTime(int hour, int minute)` ist so ein Beispiel. Mit `until(…)` lässt sich eine Zeitdauer vom Typ `Period` liefern.

1.1.3 Ostertage *

In vielen Geschäftsprogrammen gibt es Fragen nach dem *Ostersonntag*[3], da er der Bezugspunkt für viele Feiertage ist:

- *Aschermittwoch* (Beginn des 40-tägigen Fastens) ist 46 Tage *vor* Ostersonntag (und zwei Tage nach Rosenmontag, egal, ob mit Helau! oder Alaaf!).
- *Christi Himmelfahrt* (Auffahrt): 39 Tage *nach* Ostersonntag und immer an einem Donnerstag
- *Pfingstsonntag*: 49 Tage nach Ostersonntag
- *Fronleichnam* (Fronleichnamsfest): 60 Tage nach Ostersonntag
- In manchen Gegenden werden drei Tage vor Christi Himmelfahrt *Bitttage* gefeiert.
- Der vorletzte Sonntag vor Ostern heißt *Passionssonntag*.
- Der Sonntag vor Ostern ist der *Palmsonntag*.

Zur Berechnung des beweglichen Ostersonntags gibt es unzählige Algorithmen. Eine Formel wurde 1876 in Butchers „Ecclesiastical Calendar“ veröffentlicht. Er arbeitet für Jahre ab 1582 im gregorianischen Kalender korrekt, berücksichtigt aber keine julianischen Zeiten. Der Ostersonntag liegt zwischen dem 22. März und dem 25. April:

com/tutego/insel/time/easter/Easter.java

```
package com.tutego.insel.time.easter;
```

[3] Viele wissen es nicht mehr: Da ist Jesus auferstanden.

```
import java.util.*;

public class Easter {

  /**
   * Returns the date of Easter Sunday for a given year.
   * @param year > 1583
   * @return The date of Easter Sunday for a given year.
   */
   public static LocalDate easterSunday( int year ) {
    int i = year % 19;
    int j = year / 100;
    int k = year % 100;

    int l = (19 * i + j - (j / 4) - ((j - ((j + 8) / 25) + 1) / 3) + 15) % 30;
    int m = (32 + 2 * (j % 4) + 2 * (k / 4) - l - (k % 4)) % 7;
    int n = l + m - 7 * ((i + 11 * l + 22 * m) / 451) + 114;

    int month = n / 31;
    int day   = (n % 31) + 1;

    return LocalDate.of( year, month, day );
  }
}
```

Ein Test soll für das aktuelle Jahr und die vorangehenden Jahre den Ostersonntag ausgeben:

*Listing **Fehler! Kein Text mit angegebener Formatvorlage im Dokument.**.1: com/tutego/insel/easter/time/EasterDemo.java. main()*

```
String format = "%1$ta %1$tD%n";
LocalDate date = Easter.easterSunday( Year.now().getValue() );
System.out.printf( format, date );                          // zum Beispiel So 04/20/14
System.out.printf( format, Easter.easterSunday( 2012 ) );   // So 04/08/12
System.out.printf( format, Easter.easterSunday( 2013 ) );   // So 03/31/13
```

1.1.4 Die Klasse YearMonth

Die Klasse `YearMonth` repräsentiert nur einen Monat und ein Jahr. Aufgebaut werden kann das Objekt wieder über `now()` oder über statische `of(…)`-Methoden. Auch `YearMonth` hat nicht viele Getter, das erwartete `Month getMonth()` oder `int getMonthValue()`. `isAfter(…)/isBefore(…)` ermöglicht relative Vergleiche, `with(…)` verändert Felder in einem neuen `YearMonth`-Objekt.

Beispiel

Zu den Gettern zählt `lengthOfMonth()`, das die maximale Anzahl der Tage eines Monats liefert:

```
System.out.println( YearMonth.parse( "2010-02" ).lengthOfMonth() ); // 28
System.out.println( YearMonth.parse( "2012-02" ).lengthOfMonth() ); // 29
```

1.1.5 Die Klasse MonthDay

Ein `MonthDay` repräsentiert einen Tag im Monat, wie den Geburtstag (aber eben ohne Jahr) oder Neujahr. Neben den erwartbaren Anfragemethoden wie `Month getMonth()`, `int getDayOfMonth()` ist `isValidYear(int)` interessant, das uns sagt, ob der Tag/Monat für ein Jahr gültig ist.

Beispiel

Der Februar 2010 hat 28, nicht 29 Tage und ist ein Schaltjahr.

```
System.out.println( YearMonth.parse( "2010-02" ).lengthOfMonth() );    // 28
System.out.println( MonthDay.parse( "--02-29" ).isValidYear( 2010 ) ); // false
```

1.1.6 Aufzählung DayOfWeek und Month

Das ganze Paket `java.time` verfügt nur über zwei Aufzählungen, eine für die Wochentage und eine für die Monate. Auch wenn `DayOfWeek` und `Month` so klingen wie temporale Klassen, so lassen sie sich *nicht* zum Speichern eines Monats oder Wochentages verwenden, es sind reine Aufzählungstypen ohne benutzerdefinierten Zustand.

Aufzählung DayOfWeek für alle Wochentage

`LocalDate` liefert mit `getDayOfWeek()` kein `int` zwischen 1 und 7, sondern eine Aufzählung vom Typ `DayOfWeek`; sie besteht aus sieben Konstanten von `MONDAY` bis `SUNDAY`. Das nette an dem Aufzählungstyp sind Methoden wie `plus(…)`/`minus(…)`, um einfach auf einen Tag eine Anzahl zu -addieren bzw. von ihm zu subtrahieren.

Beispiel

In 100 Tagen läuft eine Wette ab, an welchem Tag genau?

```
LocalDate now = LocalDate.now();
DayOfWeek dayOfWeek = now.getDayOfWeek();
System.out.println( dayOfWeek.plus( 100 ) ); // z.B. SATURDAY
```

Die Ausgaben mit `toString()` sind natürlich die großgeschriebenen `enum`-Konstanten. Formatierung ist zum Beispiel über die `DayOfWeek`-Methode `getDisplayName(TextStyle, Locale)` möglich, wobei `TextStyle` auch wiederum ein Aufzählungstyp ist und die Ausführlichkeit bestimmt.

Beispiel

Gib vom oberen Beispiel den Wochentag in Deutsch aus:

```
DayOfWeek dow = LocalDate.now().getDayOfWeek().plus( 100 );
System.out.println( dow.getDisplayName( TextStyle.FULL, Locale.GERMANY ) ); // Samstag
```

Aufzählung Month für alle Monate

So wie auch `DayOfWeek` ein Aufzählungstyp ist, so deklariert Java für die Monate den Aufzählungstyp `Month` mit zwölf Konstanten von `JANUARY` bis `DECEMBER`. Die Methode `length(boolean leapYear)` liefert in Abhängigkeit von der Schaltjahr-Information die Anzahl der Tage des Monats, `minLength()`/`maxLength()` die kleinste bzw. größte Anzahl an Tagen, was natürlich nur wieder für den Februar einen Unterschied macht. Februar hat ein Minimum von 28 Tagen und ein Maximum von 29, zusammen mit allen anderen Monaten ergibt sich also ein Wertebereich von 28 bis 31.

Des Weiteren gibt es wieder eine Methode `getDisplayName(TextStyle, Locale)`.

1.1.7 Klasse LocalTime

Während `LocalDate` nur das Datum repräsentiert, speichert `LocalTime` nur die Zeit. Oftmals wird `now()` die zentrale Methode sein, um einen Zeitstempel zu erfragen, und wir haben die üblichen `of(…)`-Methoden zum Aufbau eines `LocalTime`-Objekts. Mit `ofNanoOfDay(long nanoOfDay)` bzw. `ofSecondOfDay(long secondOfDay)` ist ein `LocalTime` auch aus Nanosekunden/Sekunden seit 00:00 aufgebaut. Mit den Gettern `getHour()`, `getMinute()`, `getSecond()`, `getNano()` sind die Felder dann einfach erfragt.

1.1.8 Klasse LocalDateTime

`LocalDateTime` kombiniert `LocalTime` und `LocalDate`, speichert aber keine Zeitzone. Damit stehen Entwicklern Jahr, Monat, Tag und Stunden, Minuten, Sekunden, Nanosekunden zur Verführung. Dem Aufbau des Objekts dienen wieder die `of(…)`-Methoden oder die anderen `withXXX()`-Methoden, um ein existierendes Datum abzuwandeln.

Beispiel

Welches Datum und welche Zeit sind heute in einem Jahr und einer Stunde?

```
LocalDateTime dateTime = LocalDateTime.now()
                                      .plusYears( 1 ).plusHours( 1 );
System.out.println( dateTime );       // zum Beispiel 2016-05-02T00:12
```

Mit der Methode `until(…)` bekommen wir relativ zu diesem Zeitpunkt eine Zeiteinheit auf-addiert.

Beispiel

Wie viel Tage sind bisher vom 1. März 1973 bis heute vergangen?

```
long until = LocalDate.of( 1973, Month.MARCH, 1 )
                      .until( LocalDate.now(), ChronoUnit.DAYS );
System.out.println( until );
```

Auf das gleiche Ergebnis kommt auch:

```
until = ChronoUnit.DAYS.between( LocalDate.of(1973,Month.MARCH,1), LocalDate.now() );
```

Die anderen temporalen Klassen bieten übrigens auch `until(…)`.

Während die `of(…)`-Methoden Objekte immer aufbauen, konvertieren die `toXXX()`-Methoden, sodass eine Umwandlung von `LocalDateTime` in `LocalTime` und `LocalDate` möglich ist.

Beispiel

```
LocalDate date = LocalDate.now();
LocalTime time = LocalTime.now();
LocalDateTime dateTimeFromDateAndTime = LocalDateTime.of( date, time );
LocalDate dateFromDateTime = dateTimeFromDateAndTime.toLocalDate();
LocalTime timeFromDateTime = dateTimeFromDateAndTime.toLocalTime();
```

1.1.9 Klasse Year

Mit `Year` bekommen wir ein Jahr gespeichert. Vorteil gegenüber der üblichen Speicherung als `int` sind Abfragemethoden, wenn ein Jahr ein eigener Datentyp ist. `length()` liefert die Anzahl der Tage im Jahr, mit `isLeap()` bekommen wir heraus, ob das Jahr ein Schaltjahr ist, das Gleiche übernimmt auch die statische Methode `isLeap(long year)`. Eine weitere Abfragemethode ist `-isValidMonthDay(MonthDay monthDay)`, die uns sagt, ob der Tag/Monat für das gegebene Jahr erlaubt ist.

Es gibt einige `atXXX()`-Methoden, die mit weiteren Informationen neue temporale Objekte -bilden, etwa `LocalDate atDay(int dayOfYear)`, `YearMonth atMonth(int month)`, `YearMonth -atMonth(Month month)`, `LocalDate atMonthDay(MonthDay monthDay)`.

1.1.10 Zeitzonen-Klassen ZoneId und ZoneOffset

Unsere bisherigen Klassen für ein Datum speichern keine Informationen über eine Zeitzone. Dabei sind Zeitzonen das gewisse Extra, denn wir können nicht ignorieren, dass Menschen in verschiedenen Zeitzonen wohnen und wir selbst innerhalb eines Landes (wie Russland, USA, aber nicht China) unterschiedliche Zeitzonen haben; Online-Meetings in diesen Ländern fallen regelmäßig flach, weil Mitarbeiter vergessen, die Zeitzone zu berücksichtigen.

Eine Zeitzone ist also eine Region mit der gleichen Zeit, und sie wird in einer Textform festgehalten, die Region und Stadt benennt, etwa Europe/Berlin. Mit `ZoneId.systemDefault()` bekommen wir die aktuelle Zeitzone, und eine weitere API-Funktion ermittelt alle diese Kennungen.

Beispiel

```
Set<String> allZones = new TreeSet<>( ZoneId.getAvailableZoneIds() );
System.out.println( allZones ); // [Africa/Abidjan, …, Zulu]
```

Jede dieser Zeitzonen ist mit einem Offset assoziiert, der relativ zur Greenwich/UTC-Zeit ist. Wir in Deutschland sind zum Beispiel UTC+1 bei der mitteleuropäischen Zeit (MEZ) und UTC+2 bei der mitteleuropäischen Sommerzeit (MESZ). All diese Informationen über Zeitzonen und Offsets stecken in zwei Klassen, `ZoneId` und `ZoneOffset`. `ZoneId` selbst repräsentiert den Identifizierer, `ZoneOffset` den Zeitzonen-Offset.

Beispiel

```
Set<String> allZones = new TreeSet<>( ZoneId.getAvailableZoneIds() );
for ( String zone : allZones ) {
  ZonedDateTime zdt = LocalDateTime.now().atZone( ZoneId.of( zone ) );
  ZoneOffset zoneOffset = zdt.getOffset();
  System.out.println( zone + " " + zoneOffset.getId() );
}
```

Es gibt nur eine statische Methode `getAvailableZoneIds()`, die alle verfügbaren Zeitzonen in String-Form liefert, sodass wir im nächsten Schritt über `ZoneId.of(…)` arbeiten müssen, um ein `ZoneId`-Objekt zu bekommen. Das hat jedoch keine Offset-Informationen, sodass wir von dort aus weitergehen müssen: Das über `now()` bezogene temporale Objekt `LocalTime` können wir über `atZone(…)` mit einer Zeitzone verbinden; `ZonedDateTime` ist sozusagen das `LocalDateTime` mit Zeitzonen-Information (`ZoneDate` alleine gibt es nicht). Bei `ZonedDateTime` können wir dann den Offset erfragen.

1.1.11 Temporale Klassen mit Zeitzoneninformationen

Die Date-Time-API bietet Entwicklern drei Klassen mit Zeitzonen-Informationen, wobei die Klasse `ZonedDateTime` schon kurz im Beispiel vorkam. In einem `ZonedDateTime` sind die üblichen Werte über das Datum (Tag, Monat, Jahr) und die Zeit (Stunde, Minute, Sekunde, Nanosekunde) kodiert sowie die Zeitzone (also Region/Stadt) und den Offset. `ZonedDateTime` ist dabei keine Unterklasse von `LocalDateTime`, die Klasse hat keine besondere Basisklasse. Doch nicht immer ist die Region (Time-Zone-ID) relevant, daher existieren zwei Klassen nur mit dem Offset-Wert einer Zeitzone und das sind die Klassen `OffsetDateTime` und `OffsetTime`, wobei der Klassename schon klarmacht, dass Ersteres wieder Datum plus Zeit repräsentiert, Letzteres aber nur die Zeit mit einem Offset.

Die Klassen `OffsetDateTime` und `OffsetTime` sind immer dann nützlich, wenn die Region eben keine Rolle spielt, doch müssen sich Entwickler bewusst sein, dass nur `ZonedDateTime` die ganzen Regeln für die Zeitzonen, Zeitwechsel (Sommer-/Winterzeit) berücksichtigt.

ZonedDateTime

Ein Objekt vom Typ `ZonedDateTime` lässt sich mit diversen `ofXXX(…)`-Methoden aufbauen. Die Daten kommen direkt von den Datum-Zeit-Werten oder aus vorhandenen `LocalXXX`-Objekten. Immer angegeben werden muss die `ZoneId`.

- `static ZonedDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond, ZoneId zone)`
- `static ZonedDateTime of(LocalDate date, LocalTime time, ZoneId zone)`
- `static ZonedDateTime of(LocalDateTime localDateTime, ZoneId zone)`

Eine weitere Methode erlaubt die Angabe einer `Instant` (Sekunden seit dem 1.1.1970):

- `static ZonedDateTime ofInstant(Instant instant, ZoneId zone)`

Weiterhin existieren drei Methoden mit zusätzlichem `ZoneOffset`:

- `static ZonedDateTime ofInstant(LocalDateTime localDateTime, ZoneOffset offset, ZoneId zone)`
- `static ZonedDateTime ofLocal(LocalDateTime localDateTime, ZoneId zone, ZoneOffset preferredOffset)`
- `static ZonedDateTime ofStrict(LocalDateTime localDateTime, ZoneOffset offset, ZoneId zone)`

Die letzte Methode sticht hervor, weil sie prüft, ob die Zeit überhaupt möglich ist, etwa dann, wenn versucht wird, einen Zeitpunkt zwischen einem Wechsel von Winter- nach Sommerzeit anzugeben, in der die Uhr eine Stunde vorgestellt wird und in der folglich eine Stunde fehlt.

Eine Alternative ist, auf einem `LocalDateTime`-Objekt über die Methode `atZone(ZoneId)` in ein `ZonedDateTime` zu konvertieren.

Beispiel

```
ZoneId zone = ZoneId.of( "Europe/Berlin" );
ZonedDateTime zdt1 = LocalDateTime.now().withMonth( Month.JANUARY.getValue() ).atZone( zone );
ZonedDateTime zdt2 = LocalDateTime.now().withMonth( Month.JULY.getValue() ).atZone( zone );
System.out.println( zdt1.getOffset().getId() + " " + zdt2.getOffset().getId() );
```

Die Ausgabe ist +01:00 +02:00 und zeigt den Unterschied zwischen Winterzeit im Januar und Sommerzeit im Juni.

`ZoneId` hat eine interessante Methode, `getRules()`, die wiederum `isDaylightSavings(…)` bietet, worüber wir einen Wahrheitswert erfahren, ob wir in der Sommerzeit (engl. *Daylight Saving -Time*, kurz DST) oder Winterzeit sind.

Beispiel

```
ZoneId zone = ZoneId.of( "Europe/Berlin" );
ZonedDateTime date = LocalDateTime.now().atZone( zone );
Instant instant1 = date.withMonth( Month.JANUARY.getValue() ).toInstant();
System.out.println( zone.getRules().isDaylightSavings( instant1 ) ); // false
Instant instant2 = date.withMonth( Month.JULY.getValue() ).toInstant();
System.out.println( zone.getRules().isDaylightSavings( instant2 ) ); // true
```

Intern greift `ZonedDateTime` auf die Klasse `ZoneRules` zurück, die alle Zeitzonen und Veränderungen der letzten Jahre kennt. Zur Erinnerung: `OffsetDateTime` und `OffsetTime` greifen nicht auf `-ZoneRules` zurück.

Beispiel

Libyen hat im letzten Jahrhundert relativ viele Zeitzonenwechsel erlebt:

```
ZoneRules rules = ZoneRulesProvider.getRules( "Libya", false );
for ( ZoneOffsetTransition rule : rules.getTransitions() )
  System.out.println( rule );
```

Die Ausgabe besteht aus 30 Einträgen von

```
Transition[Gap at 1920-01-01T00:00+00:52:44 to +01:00]
Transition[Gap at 1951-10-14T02:00+01:00 to +02:00]
```

bis

```
Transition[Overlap at 2012-11-10T02:00+02:00 to +01:00]
Transition[Gap at 2013-03-29T01:00+01:00 to +02:00]
```

In der Ausgabe entdecken wir „Overlap“ immer dann, wenn die Zeit zurückgestellt wird, und „Gap“, wenn eine Lücke entsteht, dadurch dass die Uhr vorgestellt wird.

Die `ZoneRules` arbeiten eher im Hintergrund und sind für übliche Anwendungen wenig relevant. Viel wichtiger ist die Tatsache, dass `ZonedDateTime` automatisch auf diese zurückgreift, wenn zum Beispiel das aktuelle `ZonedDateTime`-Objekt in eine Zeitzone verschoben wird.

Beispiel

Wie spät haben wir es jetzt in Deutschland, und wie viel Uhr wäre das auf den Philippinen?

```
ZonedDateTime zdt1 = ZonedDateTime.now( ZoneId.of( "Europe/Berlin" ) );
ZonedDateTime zdt2 = zdt1.withZoneSameInstant( ZoneId.of( "Asia/Manila" ) );
System.out.println( zdt1 );  // 2014-02-02T19:03:33.361+01:00[Europe/Berlin]
System.out.println( zdt2 );  // 2014-02-03T02:03:33.361+08:00[Asia/Manila]
```

In dem Zusammenhang sind die `plusXXX()`/`minusXXX()`-Methoden wieder interessant.

Beispiel

Morgen um 10 Uhr geht ein Flieger von Frankfurt nach Manila. Der Flug dauert 15 Stunden. Wann kommt die Maschine in Manila an?

```
ZoneId departureTimezone = ZoneId.of( "Europe/Berlin" );
ZoneId arrivalTimezone   = ZoneId.of( "Asia/Manila" );
ZonedDateTime zdt1 = LocalTime.of( 10, 0 ).atDate( LocalDate.now() )
                      .atZone( departureTimezone );
ZonedDateTime zdt2 = zdt1.withZoneSameInstant( arrivalTimezone ).plusHours( 15 );
System.out.println( zdt1 );  // 2014-02-02T10:00+01:00[Europe/Berlin]
System.out.println( zdt2 );  // 2014-02-03T08:00+08:00[Asia/Manila]
```

Zusammenfassung zur Umwandlung

Ist ein `ZonedDateTime` gefragt, und ein `LocalTime` + `LocalDate` bzw. `LocalDateTime` gegeben, so ist die Objektkonstruktion über `of(LocalDate date, LocalTime time, ZoneId zone)` oder `of(LocalDateTime localDateTime, ZoneId zone)` möglich. Alternativ liefert die `atZone(ZoneId)`-Methode von `LocalDateTime` ein `ZonedDateTime`-Objekt. Eine Verschiebung der Zeit/des Datums wird nicht vorgenommen. Intern referenziert `ZonedDateTime` lediglich `LocalDateTime`, `ZoneOffset` und `ZoneId`.

Umgekehrt existieren die Methoden `LocalDate toLocalDate()`, `LocalDateTime toLocalDateTime()`, `LocalTime toLocalTime()` für die Konvertierung in die andere Richtung. Ein erfragtes `DateTime`-Objekt behält das gleiche Jahr und den gleichen Monat und Tag, eine Konvertierung findet hier nicht statt.

Beispiel

Setzen von Zeitzone und Entfernen von Zeitzone:

```
LocalDateTime date = LocalDateTime.of( 2014, 3, 1, 2, 0, 0, 0 );
ZonedDateTime zdt1 = ZonedDateTime.of( date, ZoneId.of( "Europe/Berlin" ) );
ZonedDateTime zdt2 = ZonedDateTime.of( date, ZoneId.of( "Asia/Oral" ) );
System.out.println( zdt1 );    // 2014-03-01T02:00+01:00[Europe/Berlin]
System.out.println( zdt1.toLocalDateTime() ); // 2014-03-01T02:00
System.out.println( zdt2 );    // 2014-03-01T02:00+05:00[Asia/Oral]
System.out.println( zdt2.toLocalDateTime() ); // 2014-03-01T02:00
```

Klassen OffsetTime und OffsetDateTime

Die beiden Klassen `OffsetTime` und `OffsetDateTime` speichern die Zeit bzw. Datum und Zeit, aber keine wirklichen Zeitzonen, sondern nur eine Verschiebung bezüglich der Greenwich/UTC-Zeit. Dieser Zeitzonen-Offset wird nicht einfach als `double` repräsentiert, sondern vom Typ `ZoneOffset`, womit dann die Information in einer String-Repräsentation aussieht wie „+/–Stunden:Minuten“.

Aufgebaut werden die temporalen Objekte `OffsetTime` und `OffsetDateTime` wieder über die `of(…)`-Methoden, wobei diese im Argument ein `ZoneOffset`-Objekt annehmen.

Beispiel

```
LocalDateTime date = LocalDateTime.now();
ZoneOffset offset = ZoneOffset.of( "+01:00" );
OffsetDateTime offsetDate = OffsetDateTime.of( date, offset );
System.out.println( offsetDate ); // 2014-02-02T19:25:44.983+01:00
```

1.1.12 Klassen Period und Duration

Für den Abstand zweier Zeiteinheiten gibt die Date-Time-API zwei Klassen vor:

- Den Abstand zweier datumsbasierter Zeiteinheiten repräsentiert `Period`, und es entspricht den uns bekannten Zeitintervallen wie „1 Jahr, 3 Monate, 4 Tage“. Wichtig ist zu wissen, dass es keinen Zeitanteil gibt, also eine `Period` nur Jahr, Monat, Tag speichert, aber keine Stunden und anderen kleineren Anteile.
- Den Abstand zweier Zeitwerte repräsentiert `Duration`; die betrachtete Einheit ist Sekunden bzw. Nanosekunden, Datumsanteile hat `Duration` nicht.

Üblicherweise verwenden wir eine der beiden Klassen, um Abstände zu repräsentieren, und natürlich bieten die Objekte auch elegante Methoden. Wer Differenzen nicht als Objekt braucht, sondern mit einem `long`

zufrieden ist, kann auch `between(Temporal temporal1Inclusive, Temporal temporal2Exclusive)` auf dem Aufzählungstyp `ChronoUnit` nutzen.

Period

Die statischen Methoden `of(int years, int months, int days)`, `ofDays(int days)`, `ofMonths(int months)`, `ofWeeks(int weeks)`, `ofYears(int years)` liefern ein neues `Period`-Objekt. Abwandeln tun es die Methoden `withDays(int days)`, `withMonths(int months)` und `withYears(int years)` und die Additions-/Subtraktionsmethoden. Oftmals wird eine `Period` aber auch als Differenz zweier Datumswerte berechnet, dafür bietet `Period` die Methode `between(LocalDate startDateInclusive, LocalDate endDateExclusive)`. Auch liefert `until(ChronoLocalDate endDateExclusive)` aus `LocalDate` ein `Period`-Objekt zurück, sonst nutzt keine andere temporale Klasse `Period`, von `IsoChronology` einmal abgesehen …

Beispiel

```
Period duration = Period.between( LocalDate.of( 1973, 2, 3 ), LocalDate.now() );
System.out.println( duration );                          // P40Y11M2D
boolean hasBirthday = duration.withYears( 0 ).isZero();
System.out.println( hasBirthday );
```

Die `toString()`-Methode liefert eine standardisierte Kennung, wie sie ISO-8601 beschreibt. Wenn wir eine Person mit Geburtstag haben, gibt es nur einen Unterschied in Jahren, aber Tag und Monat sind bei `Period` null; also können wir auf einen Geburtstag prüfen, in dem wir bei der Period das Jahr auf null setzen und dann mit `isZero()` testen, ob alle Felder 0 sind.

`Period` bietet die erwarteten Abfragemethoden wie `getMonths()`, `getDays()` und `getYears()`. `isNegative()` und `isZero()` sind selbsterklärend. Falls die Gesamtdauer zum Beispiel in Tagen erfragt werden soll, kann `Period` nicht helfen, hier kann zur `between(…)`-Methode von `ChronoUnit` gegriffen werden. Die Felder einer `Period` können negativ sein, wenn der Endwert vor dem Startwert liegt.

Bisher unangesprochen ist die Frage, wie sich Zeitzonen verhalten. Wenn wir `Period.between(…)` nutzen, ist das Argument sowieso ein `LocalDate`, und das hat keine Zeitzoneninformationen; die Differenz zwischen zwei Datumswerten mit Zeitzoneninformationen muss also anders berechnet werden. Hier bietet sich zum Beispiel `long until(Temporal endExclusive, TemporalUnit unit)` von `ZonedDateTime` an, aber das hat nichts mehr mit `Period` zu tun. `Period` kann bei `ZonedDateTime` nur zum Addieren/Subtrahieren verwendet werden, denn diversen Methoden lässt sich ein `TemporalAmount` mitgeben, und eine `Period` (und auch `Duration`) implementiert die Schnittstelle `TemporalAmount`.

Duration

Die Klasse `Period` als Zeitintervall für zwei Datumswerte haben wir gerade besprochen. Eine weitere Klasse `Duration` repräsentiert Dauern von Zeiten und ist weder mit Zeitzonen verbunden noch mit anderen Zeitleisten. Daher ist auch ein Tag idealisiert exakt 24 Stunden lang, Schaltsekunden kennt die Klasse nicht, und einen Tag zu addieren heißt, 24 Stunden aufzurechnen.

Die interne Berechnungseinheit ist Sekunden bzw. Nanosekunden, auch wenn Hilfsmethoden Zeiteinheiten bis Stunden erlauben. Darüber wird eine `Duration` auch aufgebaut, über `ofXXX()`-Methoden wie `ofSeconds(long seconds, long nanoAdjustment)` oder `ofDays(long days)`. Differenzen bildet wieder `Duration between(Temporal startInclusive, Temporal endExclusive)`, wobei `Temporal` eine Schnittstelle ist, die etwa von `LocalDate`, `LocalDateTime`, `LocalTime`, `OffsetDateTime`, `OffsetTime`, `ZonedDateTime`, `Year`, `YearMonth`, `Instant` implementiert wird, nicht aber von `Period`.

Abgewandelt wird eine `Duration` wieder über `withXXX(…)` oder die `minusXXX(…)/plusXXX(…)`-Methoden. `toNanos()`, `toMillis()`, `toMinutes()`, `toHours()`, `toDays()` konvertieren in den gewünschten Typ, `getNano()` und `getSeconds()` liefern die zwei Bestandteile einer `Duration`. Wichtig ist der Unterschied: Die Getter liefern den Feldwert von `Duration`, während `toXXX()` immer konvertiert, also ist eine `Duration` mit 1 Sekunde und 0 Nanosekunden gleich 1 bei `getSecond()` und 0 bei `getNano()`, aber 1.000.000.000 Nanosekunden bei `toNanos()`.

Beispiel

Wie viel Zeit vergeht zwischen der Ausführung?

```
Instant start = Instant.now();
try {
  Files.walk( Paths.get( System.getProperty( "user.home" ) ) ).count();
} catch ( Exception e ) { }
Instant end = Instant.now();
System.out.println( Duration.between( start, end ).toMillis() + " ms" );
```

1.1.13 Klasse Instant

Die bisherigen temporalen Klassen arbeiten mit Konzepten, die für Menschen im Mittelpunkt stehen, also mit Stunden, Minuten, Tagen, Monaten. Das ist bei der Klasse `Instant` anders, sie repräsentiert eine Anzahl Millisekunden seit dem 1.1.1970 – der Referenzpunkt heißt Epoche. Der Wert einer `Instant` ist positiv (später als 1970) oder negativ (vor 1970) und ist eine gute Basis, wenn eine Applikation einen Zeitstempel benötigt, etwa für eine Log-Datei.

Exemplar einer `Instant` bilden bekannte Methoden wie `now()` oder `ofEpochMilli(…)`, `ofEpochSecond(…)`, `ofEpochSecond(…)`. Bei den Abfragemethoden sind `isAfter(Instant)/isBefore(Instant)` für Vergleiche nützlich – im Übrigen implementiert die Klasse auch `Comparable` mit exakt dieser Vergleichslogik. Mit der Methode `until(…)` bekommen wir wieder den Abstand berechnet, bei `Instant` also die Zeitdifferenz zwischen zwei `Instant`-Objekten.

Beispiel

Wie viele Tage sind nach dem 1.1.1970 bis heute vergangen?

```
long daysFromEpoch = Instant.ofEpochSecond( 0 ).until( Instant.now(), ChronoUnit.DAYS );
System.out.println( daysFromEpoch );  // z.B. 16104
```

Die Berechnung basiert auf der idealisierten Form, dass ein Tag genau 60 * 60 * 24 Sekunden hat. Drehen wir die Berechnung um, schreiben also `Instant.now().until(Instant.ofEpochSecond(0), ChronoUnit.DAYS)`, ist das Ergebnis negativ.

Existierende `Instant`-Exemplare können als Basis für neue abgewandelte Exemplare dienen, etwa mit den `with(…)`-Methoden oder mit `plusXXX(…)/minusXXX(…)`.

Beispiel

Wie ist die `Instant` in einer Stunde?

```
Instant inOneHourInstant = Instant.now().plus( 1, ChronoUnit.DAYS );
System.out.println( inOneHourInstant );  // 2014-02-04T10:12:13.344Z
```

Die `toString()`-Methode liefert eine ISO-8601-Kennung wie `LocalDateTime`, eine `format()`-Methode gibt es bei `Instant` aber *nicht*. `Instant` bietet eben keine Abfragemethoden nach Jahr, Tag; das obliegt den anderen Klassen, die aber durchaus ein `Instant` annehmen, um darüber ein tem-porales Objekt zu bilden. Um die Zeit/das Datum zu formatieren, gibt es unterschiedliche -Varianten:

- `Instant` bietet über `atOffset(ZoneOffset offset)` (liefert `OffsetDateTime`) oder `atZone(ZoneId zone)` (liefert `ZonedDateTime`) ein neues temporales Objekt mit `format(…)`-Methode.
- `LocalDateTime` bietet die statische Methode `ofInstant(Instant instant, ZoneId zone)`, um zusammen mit `Instant` und der Zeitzone eine `LocalDateTime` zu bilden. Im Prinzip könnten auch dem `Instant` über `getNano()` die Nanosekunden entlockt und über die statische `LocalDate-Time`-Methode `ofEpochSecond(long epochSecond, int nanoOfSecond, ZoneOffset offset)` ein `LocalDateTime` aufgebaut werden.

1.1.14 Parsen und Formatieren von Datumszeitwerten

Alle temporalen Typen überschreiben standardmäßig die `toString()`-Methode und liefern eine standardisierte Ausgabe. Daneben bieten die Typen eine `format(…)`-Methode, der ein Formatierungsobjekt übergeben wird, sodass individuelle Ausgaben nach einem Muster möglich sind. Neben dem Formatieren

bieten die Typen auch eine statische `parse(…)`-Methode, die einmal mit einem String-Parameter das Format erwartet, was `toString()` liefert, und eine Version mit zwei Parametern, wobei dann ein Formatierungsobjekt erlaubt ist, um genau das Format anzugeben, nach dem geparst werden soll. Weiterhin lassen sich die in Java 8 eingeführten temporalen -Typen auch bei `String.format(…)` und den `java.util.Formatter` einsetzen.

Beispiel

```
LocalDate now = LocalDate.now();
System.out.println( now );          // 2014-03-21
DateTimeFormatter formatter = DateTimeFormatter.ofPattern( "d. MMMM yyyy" );
String nowAsString = now.format( formatter );
System.out.println( nowAsString );  // 21. März 2014
LocalDate nowAgain = LocalDate.parse( nowAsString, formatter );
System.out.println( nowAgain );     // 2014-03-21
```

Die API-Dokumentation zu `DateTimeFormatter` zeigt einige vordefinierte Formatierungsobjekte und listet alle Formatspezifizierer auf.

1.1.15 Das Paket java.time.temporal *

Bisher haben wir uns nicht mit den Vererbungsbeziehungen der Date-Time-Klassen auseinandergesetzt – das wollen wir jetzt nachholen. Gleichzeitig zeigt der Abschnitt auch die tiefer liegenden Schichten der Date-Time-API, die die einzelnen Felder (wie Jahr, Sekunde) eines Datums/einer Zeit in den Fokus rücken.

Schnittstelle Temporal

Alles das, was in der Date-Time-API einen temporalen Typ darstellt, implementiert die Schnittstelle `Temporal`. Dazu gehören:

- `Instant`, `LocalDate`, `LocalDateTime`, `LocalTime`, `OffsetDateTime`, `OffsetTime`, `Year`, `YearMonth`, `ZonedDateTime`
- `HijrahDate`, `JapaneseDate`, `MinguoDate`, `ThaiBuddhistDate`

`Temporal` deklariert Methoden zum Lesen und Verändern von temporalen Werten, wobei `Period` nicht dazugehört. Die wichtigsten Methoden sind:

- `plus(…)`/`minus(…)`
- `until(…)`
- `with(TemporalAdjuster)`
- `with(TemporalField field, long newValue)`

Schnittstelle TemporalAmount

Ein Zeitabschnitt drückt die Schnittstelle `TemporalAmount` aus, sie wird von `Period` und `Duration` implementiert. Wir finden in der Schnittstelle etwa `addTo(Temporal)` und `subtractFrom(Temporal)`, die ein `Temporal` zurückgeben.

Schnittstelle TemporalAccessor

Während `Temporal` zugleich Lese- und Modifikationsmethoden deklariert, gibt `TemporalAccessor` nur Methoden zum Lesen der Zustände vor – `Temporal` ist folglich eine Erweiterung von `TemporalAccessor`. Zu den implementierenden Klassen zählen die, die natürlich auch `Temporal` erweitern und zusätzlich `DayOfWeek`, `Month`, `IsoEra`, `MonthDay`, `ZoneOffset` und ein paar Klassen von den Nicht-ISO-Kalender-Systemen. Zusammenfassend:

- `DayOfWeek`, `Instant`, `IsoEra`, `LocalDate`, `LocalDateTime`, `LocalTime`, `Month`, `MonthDay`, `OffsetDateTime`, `OffsetTime`, `Year`, `YearMonth`, `ZonedDateTime`, `ZoneOffset`
- `HijrahDate`, `HijrahEra`, `JapaneseDate`, `JapaneseEra`, `MinguoDate`, `MinguoEra`, `ThaiBuddhistDate`, `ThaiBuddhistEra`

Die beiden Nicht-Default-Methoden sind `long getLong(TemporalField field)` und `boolean isSupported(TemporalField field)`; sie machen schon klar, dass es um einen Zugriff auf ein Feld geht, also etwa ein `ChronoField` `HOUR_OF_DAY`, `YEAR`, …

Schnittstelle TemporalField und Aufzählungstyp ChronoField

Bisher haben wir es in der Date-Time-API in der Regel mit Abfragemethoden wie `getYear()` oder `getSeconds()` zu tun gehabt. Doch das sind nichts anderes als Felder eines temporalen Typs, und für genauso ein Feld wie Jahr, Monat, Sekunde gibt es eine Schnittstelle `TemporalField`.

`TemporalField` deklariert Methoden wie `range()` oder `getFrom(TemporalAccessor temporal)`, und eine Implementierung der Schnittstelle ist der Aufzählungstyp `ChronoField`. Es gibt reichlich Aufzählungen für alle Felder, die ein Datum bzw. eine Zeit in der Date-Time-API haben können:

- `ALIGNED_DAY_OF_WEEK_IN_MONTH`, `ALIGNED_DAY_OF_WEEK_IN_YEAR`, `ALIGNED_WEEK_OF_MONTH`, `ALIGNED_WEEK_OF_YEAR`
- `AMPM_OF_DAY`
- `CLOCK_HOUR_OF_AMPM`, `CLOCK_HOUR_OF_DAY`
- `DAY_OF_MONTH`, `DAY_OF_WEEK`, `DAY_OF_YEAR`
- `EPOCH_DAY`
- `ERA`
- `HOUR_OF_AMPM`, `HOUR_OF_DAY`
- `INSTANT_SECONDS`
- `MICRO_OF_DAY`, `MICRO_OF_SECOND`
- `MILLI_OF_DAY`, `MILLI_OF_SECOND`
- `MINUTE_OF_DAY`, `MINUTE_OF_HOUR`, `MONTH_OF_YEAR`
- `NANO_OF_DAY`, `NANO_OF_SECOND`
- `OFFSET_SECONDS`
- `PROLEPTIC_MONTH`
- `SECOND_OF_DAY`, `SECOND_OF_MINUTE`
- `YEAR`
- `YEAR_OF_ERA`

Beispiel

```
TemporalAccessor now = LocalDate.now();
System.out.println( ChronoField.YEAR.isSupportedBy( now ) ); // true
System.out.println( ChronoField.YEAR.getFrom( now ) ); // 2014
System.out.println( ChronoField.MILLI_OF_DAY.isSupportedBy( now ) ); // false
// System.out.println( ChronoField.MILLI OF DAY.getFrom( now ) ); // UnsupportedTemporalTyp
eException
```

Neben der Aufzählung `ChronoField` gibt es zudem eine Klasse `IsoFields` mit einigen `TemporalField`-Konstanten wie `QUARTER_OF_YEAR`, was nützlich für ISO-8601 Kalender ist, die mit Quartalen und Kalenderwochen arbeiten. Weitere Konstanten stecken in `WeekFields` und `JulianFields`.

Schnittstelle TemporalUnit und Aufzählungstyp ChronoUnit

Die in `ChronoField` deklarieren Aufzählungen repräsentieren alle Felder eines Datums bzw. einer Zeit (wie Tag, Sekunden), und `getFrom(TemporalAccessor temporal)` liefert ein `long` mit der Belegung des Feldes. Allerdings sagt es nichts über den Typ des Feldes aus. Um das auszudrücken, gibt es eine weitere Schnittstelle `TemporalUnit` und einen Aufzählungstyp `ChronoUnit`, der `TemporalUnit` implementiert:

- `CENTURIES`, `DAYS`, `DECADES`, `FOREVER`, `HALF_DAYS`, `HOURS`, `MICROS`, `MILLENNIA`, `MILLIS`, `MINUTES`, `MONTHS`, `NANOS`, `SECONDS`, `WEEKS`, `YEARS`

Die Genauigkeit geht folglich von Millisekunden bis Millennium.

In der `TemporalField`-Schnittstelle (und somit vom Aufzählungstyp `ChronoField` realisiert) gibt es zwei Operationen, die die `TemporalUnit` eines Feldes zurückgeben: `getBaseUnit()` und `getRange-Unit()`.

Beispiel

```
System.out.println( ChronoField.MONTH_OF_YEAR.getBaseUnit() );  // Months
System.out.println( ChronoField.MONTH_OF_YEAR.getRangeUnit() ); // Years
```

Um zu erfragen, ob ein Typ der Date-Time-API auch eine Einheit unterstützt, deklariert jeder `TemporalAccessor` die Methode `isSupported(TemporalUnit)`. Eine `Instant` zum Beispiel unterstützt kein Monat oder Jahr, also gibt `Instant.now().isSupported(ChronoUnit.MONTHS) false` -zurück.

`TemporalUnit` finden wir auch bei den Klassen `Duration` und `Period`, die die Schnittstelle `TemporalAmount` implementieren, denn dort sind deklariert: `long get(TemporalUnit unit)` und `List<TemporalUnit> getUnits()`.

Funktionale Schnittstelle TemporalAdjuster

Zum Verändern eines Datum-/Zeitwerts gibt es im Paket `java.time.temporal` den `TemporalAdjuster`. Das ist eine funktionale Schnittstelle mit einer Methode `adjustInto(Temporal temporal)`, die Anpassungen an einem übermittelten temporalen Objekt vornimmt, wobei das Ergebnis ein neues justiertes temporales Objekt ist. Implementierende Klassen sind unter anderem `DayOfWeek`, `Month`, `MonthDay`, `Year`, `YearMonth`, `Instant`, `LocalDate`, `LocalTime`, `LocalDateTime`, `OffsetDate-Time`, `OffsetTime`, `ZoneOffset`; sie belegen das übergebene `Termporal` mit dem eigenen Wert.

Beispiel

Nehmen wir ein `LocalDateTime` (unser `Temporal`) an, und setzen wir Zeit- und Datumsanteil neu:

```
TemporalAdjuster temporalAdjuster1 = LocalDate.of( 1986, Month.JANUARY, 28 );
TemporalAdjuster temporalAdjuster2 = LocalTime.of( 11, 39 );
Temporal temporal = LocalDateTime.MIN;
temporal = temporalAdjuster1.adjustInto( temporal );
temporal = temporalAdjuster2.adjustInto( temporal );
System.out.println( temporal ); // 1986-01-28T11:39
```

Die Schreibweise mit `temporal = temporalAdjuster.adjustInto(temporal)` ist ungewöhnlich, und klarer ist stattdessen `temporal = temporal.with(temporalAdjuster)`. Hierbei ist offensichtlicher, dass der `TemporalAdjuster` eine Justierung vornimmt.

Beispiel

Auf unser Beispiel umgeschrieben:

```
TemporalAdjuster temporalAdjuster1 = LocalDate.of( 1986, Month.JANUARY, 28 );
TemporalAdjuster temporalAdjuster2 = LocalTime.of( 11, 39 );
Temporal temporal = LocalDateTime.MIN;
temporal = temporal.with( temporalAdjuster1 ).with( temporalAdjuster2 );
System.out.println( temporal ); // 1986-01-28T11:39
```

Dass `LocalDate` und `LocalTime` spezielle `TemporalAdjuster` sind, fällt nicht so auf, weil sie den Wert vom `Temporal` einfach nur setzen. Interessanter werden die `TemporalAdjuster` aber bei komplexerem Verhalten, etwa dem Setzen auf den nächsten Arbeitstag oder auf den 1.1 des nächsten Jahrs oder Ähnliches. Für einige Szenarien deklariert die Utility-Klasse `java.time.temporal.TemporalAdjusters` diverse statische Methoden.

Beispiel

Von heute aus gesehen, wann ist der nächste Montag?

```
Temporal now = LocalDate.now();
System.out.println( now );          // 2014-02-25
```

```
Temporal nextMonday = now.with( TemporalAdjusters.next( DayOfWeek.MONDAY ) );
System.out.println( nextMonday );   // 2014-03-03
```

`java.time.temporal.TemporalAdjusters` deklariert folgende Methoden:

- `static TemporalAdjuster dayOfWeekInMonth(int ordinal, DayOfWeek dayOfWeek)`
- `static TemporalAdjuster firstDayOfMonth()`
- `static TemporalAdjuster firstDayOfNextMonth()`
- `static TemporalAdjuster firstDayOfNextYear()`
- `static TemporalAdjuster firstDayOfYear()`
- `static TemporalAdjuster firstInMonth(DayOfWeek dayOfWeek)`
- `static TemporalAdjuster lastDayOfMonth()`
- `static TemporalAdjuster lastDayOfYear()`
- `static TemporalAdjuster lastInMonth(DayOfWeek dayOfWeek)`
- `static TemporalAdjuster next(DayOfWeek dayOfWeek)`
- `static TemporalAdjuster nextOrSame(DayOfWeek dayOfWeek)`
- `static TemporalAdjuster ofDateAdjuster(UnaryOperator<LocalDate> dateBasedAdjuster)`
- `static TemporalAdjuster previous(DayOfWeek dayOfWeek)`
- `static TemporalAdjuster previousOrSame(DayOfWeek dayOfWeek)`

Beispiel

Implementierung von `firstDayOfNextYear()` in `TemporalAdjusters`:

```
public static TemporalAdjuster firstDayOfNextYear() {
  return (temporal) -> temporal.with(DAY_OF_YEAR, 1).plus(1, YEARS);
}
```

Eigene `TemporalAdjuster` sind damit natürlich leicht implementiert.

Beispiel

Nachrichten kommen jeden Tag um 12 Uhr und um 20 Uhr. Schreibe einen `TemporalAdjuster`, der ein `Temporal` für die nächste Nachrichtensendung erzeugt:

```
TemporalAdjuster nextNewsTemporalAdjuster = temporal -> {
  LocalDateTime result = LocalDateTime.from( temporal );
  if ( result.getHour() < 12 )
    result = result.withHour( 12 );
  else if ( result.getHour() < 20 )
    result = result.withHour( 20 );
  else
    result = result.withHour( 12 ).plusDays( 1 );
  return result.withMinute( 0 ).withSecond( 0 ).withNano( 0 );
};

System.out.println( LocalDateTime.now().with( nextNewsTemporalAdjuster ) );
```

1.1.16 Konvertierungen zwischen der klassischen API und Date-Time-API

Die alten und neuen Typen sind komplett getrennt und eigenständig, aber natürlich ist es -notwendig, Brücken zu haben. Dabei ist interessant zu sehen, dass diese nur in eine Richtung gehen, von der klassischen API hin zur Date-Time-API, aber die neue API ist nicht mit `Date`, `Calendar`, und Ähnlichem „verschmutzt“. So finden sich in den alten Klassen ab Java 8 folgende Neuerungen:

- In `java.util.Calendar`: Objektmethode `toInstant()` liefert `Instant`.
- In `java.util.Date`: Objektmethode `toInstant()` liefert `Instant`.

- In `java.util.Date`: Statische Methode `from(java.time.Instant)` liefert `java.util.Date`.
- In `java.util.GregorianCalendar`: Statische Methode `from(java.time.ZonedDateTime)` liefert `GregorianCalendar`.
- In `java.util.GregorianCalendar`: Objektmethode `toZonedDateTime()` liefert `ZonedDateTime`.
- In `java.util.TimeZone`: Statische Methode `getTimeZone(ZoneId zoneId)` liefert `TimeZone`, passend zur existierenden Klassemethode `getTimeZone(String ID)`.
- In `java.util.TimeZone`: Objektmethode `toZoneId()` liefert `ZoneId`.

Erwartet eine neue API die neuen Date-Time-API-Typen, doch ist eigener alter Code mit den klassischen Datum-Zeit-Klassen vorhanden, werden `toInstant()` oder `toZonedDateTime()` am häufigsten vorkommen.

Hinweis

Richtig typsicher ist die neue API nicht, denn ohne Compilerfehler geht zwar eine Zeile wie `LocalDate.from(new Date().toInstant())` durch, aber zur Laufzeit gibt es eine `java.time.DateTimeException`: „Unable to obtain LocalDate from TemporalAccessor: 2014-02-04T08:39:34.060Z of type java.time.Instant.“ Von einem `Instant` kann so nicht ein `LocalDate` erfragt werden, korrekt wäre aber `LocalDateTime.ofInstant(new Date().toInstant(), ZoneId.systemDefault())`, und mit einem `.toLocalDate()` hinten dran, wenn nur `LocalDate` gebraucht wird, bzw. mit `.toLocalTime()` am Ende, wenn nur `LocaleTime` interessant ist. Insgesamt ist festzuhalten, dass die Mischung von alten und neuen Klassen zu Code-Monstern führt, etwa:

```
YearMonth yearMonth = YearMonth.from(
  LocalDateTime.ofInstant( new Date().toInstant(), ZoneId.systemDefault() ) );
System.out.println( yearMonth );    // z.B. 2014-02
```

Code-Migration

Wer Code migrieren möchte, hat etwas mehr Arbeit vor sich, da die neue Date-Time-API viel präziser ist und weil es deutlich mehr Typen gibt. Wer ernsthaft Code umstellen möchte, muss sich noch einmal daran erinnern, was die ursprünglichen Typen eigentlich ausdrücken sollten. Repräsentierte zum Beispiel `Date` a) nur eine Zeit, b) nur ein Datum oder c) ein Datum und eine Zeit oder d) nur eine Anzahl Millisekunden seit dem 1.1.1970? In Abhängigkeit von der Antwort wird die Austauschklasse `LocalDate`, `LocalTime`, `LocalDateTime` oder `Instant` lauten. Bei `GregorianCalendar` ist der Austausch schon etwas einfacher, hier ist `ZonedDateTime` üblich, wenn denn `GregorianCalendar` eine Zeitzone nutzt; wenn keine Zeitzone reinspielt, sind auch `LocalDateTime` bzw. `LocalDate` möglich. Im Fall von Zeitzonenklassen verteilen sich die Möglichkeiten von `java.util.TimeZone` nun auf `java.time.ZoneId` und `java.time.ZoneOffset`.