# LU08b - Java und die Zeit

Das speichern von Zeit und Datumswerten ist in vielen Applikationen notwendig. Java bietet uns mit den Datentypen `LocalDate`, `LocalTime`, `LocalDateTime`, `Period` und `Duration` aus der Bibliothek `java.time` einen eifach zu benutzenden Datentyp dafür.

## Einfache Datumswerte - LocalDate

Einfache Datumswerte beinhalten lediglich das Datum, ohne Zeitangabe.

### LocalDate erstellen

Es gibt verschiedene Möglichkeiten ein `LocalDate` zu erstellen, das den `8. Januar 2020` darstellt:

Wir erstellen das `LocalDate` über den Aufruf der `of()` Methode.

```
LocalDate date = LocalDate.of(2020, 1, 8);
```

Der Monat kann auch mit dem Enum Month angegeben werden:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 8)
```

Eine weitere Möglichkeit besteht darin, ein Datum durch Parsen einer Zeichenkette (String) zu erzeugen. Wir können die parse-Methode mit nur einem einzigen Argument verwenden, um ein Datum im Format yyyy-mm-dd zu analysieren:

```
LocalDate date = LocalDate.parse("2020-01-08");
```

Wir können auch ein anderes Datumsformat angeben, dies mit Hilfe der DateTimeFormatter-Klasse als zweiten Parameter der Parse-Methode:

```
LocalDate date = LocalDate.parse("8-Jan-2020",
DateTimeFormatter.ofPattern("d-MMM-yyyy"));
```

### LocalDate funktionen

Hier sind einige Funktionen aufgelistet, die LocalDate bietet:

Wir können vom LocalDate den heutige Tag erhalten:

```
LocalDate date = LocalDate.now();
```

Wir können Tage/Wochen/Monate/Jahre hinzurechnen oder entfernen:

```
LocalDate tomorrow = LocalDate.now().plusDays(1);
LocalDate yesterday= LocalDate.now().minusDays(1);
```

Es gibt auch passend dazu die Funktionen `plus/minusWeeks`, `plus/minusMonths` und `plus/minusYears`

Wir können Datumswerte vergleichen:

```
boolean notBefore =
LocalDate.parse("2042-06-12").isBefore(LocalDate.parse("2042-06-11"));
boolean isAfter =
LocalDate.parse("2042-06-12").isAfter(LocalDate.parse("2042-06-11"));
```

Wir können feststellen ob es ein Schaltjahr ist:

```
boolean leapYear = LocalDate.now().isLeapYear();
```

Wir können die Zeit zwischen den Daten berechnen:

```
LocalDate tomorrow = LocalDate.now().plusDays(1);
LocalDate yesterday= LocalDate.now().minusDays(1);

Period timeBetween = Period.between(yesterday, tomorrow); // Wir erhalten
den Datentyp Period. Mehr dazu im Kapitel zu Period
```

Wir können einzelne Tage/Monate/Jahre abfragen.

```
LocalDate today = LocalDate.now();
int day = today.getDayOfMonth();
int month = today.getMonthValue();
int year = today.getYear();
```

Es gibt noch viele weitere Funktionen die zur Verfügung stehen, eine Liste mit Beschreibung finden Sie im Java-API

# Einfache Zeitwerte - LocalTime

Einfache Zeitwerte beinhalten lediglich die Zeit, ohne Datumsangabe.

## LocalTime erstellen

Es gibt verschiedene Möglichkeiten eine `LocalTime '' zu erstellen, die den ''7:30 Uhr` darstellt:

Wir erstellen die `LocalTime` über den Aufruf der `of()` Methode.

```
LocalTime time = LocalTime.of(7, 30);
```

Eine weitere Möglichkeit besteht darin, eine Zeit durch Parsen einer Zeichenkette (String) zu erzeugen. Wir können die parse-Methode mit nur einem einzigen Argument verwenden, um ein Datum im Format hh-mm zu analysieren:

```
LocalTime time = LocalTime.parse("07:30");
```

## LocalTime funktionen

Hier sind einige Funktionen aufgelistet, die LocalTime bietet:

Wir können eine LocalTime von der aktuellen Zeit erhalten:

```
LocalTime now = LocalTime.now();
```

Wir können Sekunden/Minuten/Stunden hinzurechnen oder entfernen:

```
LocalTime in5Minutes = LocalTime.now().plusMinutes(5);
LocalTime before5Minutes = LocalTime.now().minusMinutes(5);
```

Es gibt auch passend dazu die Funktionen `plus/minusSeconds` und `plus/minusHours`

Wir können Zeitwerte vergleichen:

```
boolean notBefore =
LocalTime.parse("7:30").isBefore(LocalTime.parse("6:30"));
// oder auch
LocalTime time = LocalTime.parse("2:45");
boolean isAfter = LocalTime.parse("7:30").isAfter(time);
// oder auch
LocalTime time0 = LocalTime.parse("2:45");
LocalTime time1 = LocalTime.parse("7:30");
boolean isAfter = time1.isAfter(time);
```

Wir können die Zeit zwischen zwei Uhrzeiten berechnen:

```
LocalTime time0 = LocalTime.parse("2:45");
LocalTime time1 = LocalTime.parse("7:30");

Duration duration = Duration.between(time0, time1);  // Wir erhalten den
Datentyp Duration. Mehr dazu im Kapitel zu Duration.
```

Wir können einzelne Stunden/Minuten/Sekunden abfragen.

```
LocalTime now = LocalTime.now();
int hour = now.getHour();
int minute = now.getMinute();
```

```
int second = now.getSecond();
```

Es gibt noch viele weitere Funktionen die zur Verfügung stehen, eine Liste mit Beschreibung finden Sie im Java-API

# Einfache Zeitdatumswerte - LocalDateTime

LocalDateTime wird verwendet, um eine Kombination aus Datum und Uhrzeit darzustellen. Dies ist die am häufigsten verwendete Klasse, wenn wir eine Kombination aus Datum und Uhrzeit benötigen.

## LocalDateTime erstellen

Es gibt verschiedene Möglichkeiten ein `LocalDateTime` zu erstellen, das den `8. Januar 2020 13:30` darstellt:

Wir erstellen das `LocalDateTime` über den Aufruf der `of()` Methode.

```
LocalDateTime date = LocalDateTime.of(2020, 1, 8, 13, 30);
```

Der Monat kann auch mit dem Enum Month angegeben werden:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 8, 13, 30)
```

Eine weitere Möglichkeit besteht darin, ein Datum durch Parsen einer Zeichenkette (String) zu erzeugen. Wir können die parse-Methode mit nur einem einzigen Argument verwenden, um ein Datum im Format yyyy-mm-ddThh:mm zu analysieren:

```
LocalDateTime dateTime= LocalDateTime.parse("2020-01-08T13:30");
```

Wir können auch ein anderes Datumszeitformat angeben, dies mit Hilfe der DateTimeFormatter-Klasse als zweiten Parameter der Parse-Methode:

```
LocalDateTime dateTime = LocalDateTime.parse("08.01.2020 13:30",
DateTimeFormatter.ofPattern("d.M.yyyy HH:mm"));
```

## LocalDateTime funktionen

Hier sind einige Funktionen aufgelistet, die LocalDateTime bietet:

Wir können vom LocalDateTime den heutige Tag mit der Uhrzeit erhalten:

```
LocalDateTime date = LocalDateTime.now();
```

Wir können Tage/Wochen/Monate/Jahre hinzurechnen oder entfernen:

```
LocalDateTime tomorrow = LocalDateTime.now().plusDays(1);
LocalDateTime yesterday = LocalDateTime.now().minusDays(1);
```

Es gibt auch passend dazu die Funktionen `plus/minusSeconds`, `plus/minusMinutes`, `plus/minusHours`, `plus/minusWeeks`, `plus/minusMonths` und `plus/minusYears`

Wir können Datumswerte vergleichen:

```
boolean notBefore =
LocalDateTime.parse("2042-06-12T13:30").isBefore(LocalDateTime.parse("2042-06-11T04:30"));
boolean isAfter =
LocalDateTime.parse("2042-06-12T13:30").isAfter(LocalDateTime.parse("2042-06-11T04:30"));
```

Wir können die Zeit zwischen den Daten berechnen:

```
LocalDateTime tomorrow = LocalDateTime.now().plusDays(1);
LocalDateTime yesterday = LocalDateTime.now().minusDays(1);

Duration timeBetween = Duration.between(yesterday, tomorrow); // Wir erhalten den Datentyp Duration. Mehr dazu im Kapitel zu Duration
```

Wir können einzelne Tage/Monate/Jahre abfragen.

```
LocalDateTime today = LocalDateTime.now();
int day = today.getDayOfMonth();
int month = today.getMonthValue();
int year = today.getYear();
int hour = today.getHour();
int minute = today.getMinute();
```

Es gibt noch viele weitere Funktionen die zur Verfügung stehen, eine Liste mit Beschreibung finden Sie im Java-API

# Period

Die Klasse Period verwendet die Einheiten **Jahr, Monat und Tag** zur Darstellung eines Zeitraums.

Wir können ein Period-Objekt als Differenz zwischen zwei Daten erhalten, indem wir die Methode between() verwenden:

```
LocalDate startDate = LocalDate.of(2015, 2, 20);
LocalDate endDate = LocalDate.of(2017, 1, 15);

Period period = Period.between(startDate, endDate);
```

Anschließend können wir die Datumseinheiten des Zeitraums mit den Methoden `getYears()`,

`getMonths()`, `getDays()` ermitteln:

```
System.out.println("Years:" + period.getYears() +
  " Months:" + Zeitraum.getMonths() +
  " Days: "+Zeitraum.getTage());
```

Die Methode `isNegative()`, die `true` zurückgibt wenn eine der Einheiten negativ ist, kann verwendet werden, um festzustellen, ob das Enddatum höher ist als das Startdatum:

```
System.out.println(period.isNegative())
```

Wenn `isNegative()` den Wert false zurückgibt, dann liegt das Startdatum vor dem Enddatum.

Eine andere Möglichkeit, ein Period-Objekt zu erstellen, basiert auf der Anzahl der Tage, Monate, Wochen oder Jahre unter Verwendung spezieller Methoden:

```
Period fromUnits = Period.of(3, 10, 10);
Period fromDays = Period.ofDays(50);
Period fromMonths = Period.ofMonths(5);
Period fromYears = Period.ofYears(10);
Period fromWeeks = Period.ofWeeks(40);
```

Es gibt noch viele weitere Funktionen die zur Verfügung
stehen, eine Liste mit Beschreibung finden Sie im Java-API

# Duration

Die Klasse Duration stellt ein Zeitintervall in Sekunden oder Nanosekunden dar und eignet sich am besten für die Handhabung kürzerer Zeitspannen in Fällen, die mehr Präzision erfordern.

Wir können die Differenz zwischen zwei Zeitpunkten als Duration-Objekt mit der Methode `between()` bestimmen:

```
LocalTime start = LocalTime.of(1, 20, 25, 1024); //(Stunde, Minute, Sekunde, Nanosekunden)
LocalTime end = LocalTime.of(3, 22, 27, 1544);

Duration.between(start, end).getSeconds();
```

Dann können wir die Methoden getSeconds() oder getNanoseconds() verwenden, um den Wert der Zeiteinheiten zu bestimmen:

```
System.out.println(duration.getSeconds());
```

Die Methode isNegative() kann verwendet werden, um zu überprüfen, ob der Endzeitpunkt größer ist als der Startzeitpunkt:

```
boolean isNegative = duration.isNegative();
```

Wir können auch ein Duration-Objekt auf der Grundlage mehrerer Zeiteinheiten erhalten, indem wir die Methoden `ofDays()`, `ofHours()`, `ofMillis()`, `ofMinutes()`, `ofNanos()` oder `ofSeconds()` verwenden:

```
Duration fromDays = Duration.ofDays(1);
Duration fromMinutes = Duration.ofMinutes(60);
```

Eine Dauer kann mit `toDays()`, `toHours()`, `toMillis()` und `toMinutes()` in andere Zeiteinheiten umgewandelt werden:

```
long stunden = fromMinutes.toHours();
```

Ein Dauerwert kann durch Methoden der Form `plusX()` oder `minusX()` erhöht oder verringert werden, wobei X für Tage, Stunden, Millis, Minuten, Nanos oder Sekunden stehen kann:

```
Duration fromSeconds = Duration.ofSeconds(60);
fromSeconds.plusSeconds(60); // neu 120 Sekunden
fromSeconds.minusSeconds(30); // neu 90 Sekunden
```

Es gibt noch viele weitere Funktionen die zur Verfügung stehen, eine Liste mit Beschreibung finden Sie im Java-API

---

© Kevin Maurizi

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
**https://wiki.bzz.ch/modul/archiv/m319/learningunits/lu08/lu08b-javaunddiezeit**

Last update: **2024/03/28 14:07**