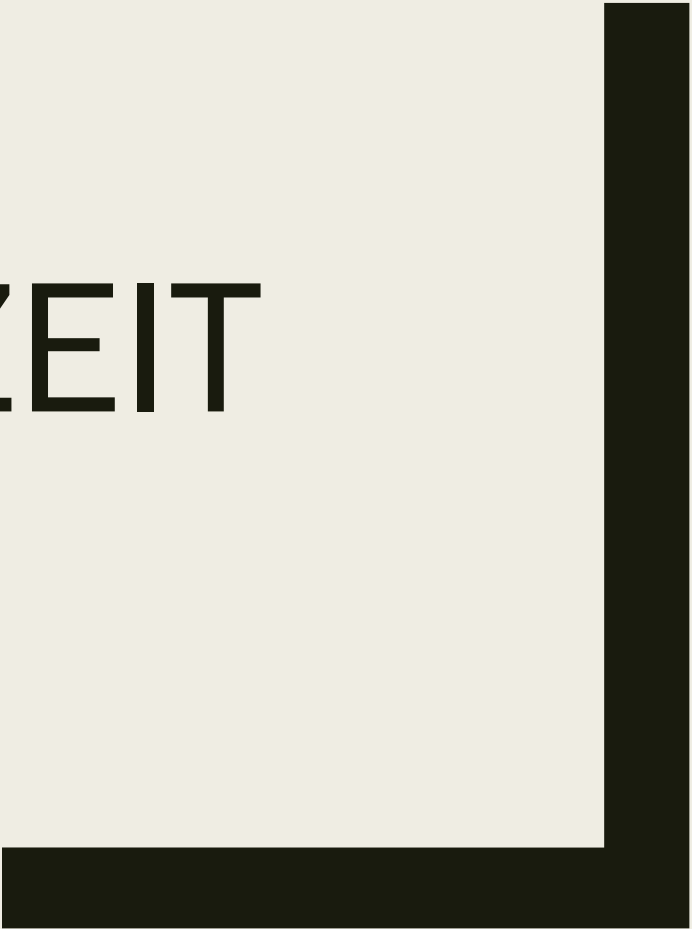




# DATUM UND ZEIT

Nach JSR-310  
JSR ThreeTen



# Java Specification Requests

In den JSR werden Informationen zu den entsprechenden Änderungen oder Neuerungen summiert bekannt gegeben. Die JSR 310 (ThreeTen manchmal genannt) beschreibt die neue „Date und Time API“, welche mit Java 8 eingeführt wurde.

Neben der ThreeTen wurde in Java SE 8 noch JSR 335 – Lambda Expressions sowie die 308 - Annotations on Java Types eingeführt.

## Weblinks zu den JSR´s

- <https://jcp.org/en/jsr/detail?id=335>  
Lambda Expressions
- <https://jcp.org/en/jsr/detail?id=310>  
Date und Time Api
- <https://jcp.org/en/jsr/detail?id=308>  
Annotations on Java Types

# Pre und Post Java 8

In den pre Java 8 Releases war das Arbeiten mit Date und Time Formaten der Arbeitsweise von Maschinen zu Grunde gelegt.

Mit der Klasse Date wurde ein Vollständiges Datum aufgebaut. Jahr, Monat, Tag, Stunde, Sekunde. Vieles in der Klasse Date wurde von Java 1.0 zu 1.1 als @Deprecated gekennzeichnet und von anderen „Intuitiveren“ Klassen übernommen.

Dies führte zu der Klasse GregorianCalendar welche von der Abstrakten Klasse Calendar abgeleitet wird. In der Klasse Calendar wurden viele Konstanten vordefiniert, womit erleichterte und für Menschen besser verständliche Nutzung der abgeleiteten Klassen ermöglicht werden sollte.

# Calendar - Konstanten

Schon zu Release wurde über den Sinn bzw. Unsinn der unterschiedlichen Nummerierungen für die Konstanten diskutiert.

Warum werden die Monate beginnend mit 0(Zero) gezählt? Also Januar 0, Februar 1, März 2 usw.

Die Antwort lautet schlicht, die IT und so gut wie jede Maschine sieht die 0 als erste Zahl im Gegensatz zum Menschen, der die 1 als erste Zahl und die 0 als unwertig ansieht.

Okay, damit konnten IT´ler wunderbar zurechtkommen, schließlich war es schon immer so in der IT gewesen das bei 0 angefangen wird zu zählen.

# Calendar Konstanten - Monate

```
618+    * Value of the {@link #MONTH} field indicating the...
621    public final static int JANUARY = 0;
622
624+    * Value of the {@link #MONTH} field indicating the...
627    public final static int FEBRUARY = 1;
628
630+    * Value of the {@link #MONTH} field indicating the...
633    public final static int MARCH = 2;
634
636+    * Value of the {@link #MONTH} field indicating the...
639    public final static int APRIL = 3;
640
642+    * Value of the {@link #MONTH} field indicating the...
645    public final static int MAY = 4;
646
```

# Calendar Konstanten - Wochentage

```
576+    * Value of the {@link #DAY_OF_WEEK} field indicating..  
579    public final static int SUNDAY = 1;  
580  
582+    * Value of the {@link #DAY_OF_WEEK} field indicating..  
585    public final static int MONDAY = 2;  
586  
588+    * Value of the {@link #DAY_OF_WEEK} field indicating..  
591    public final static int TUESDAY = 3;  
592  
594+    * Value of the {@link #DAY_OF_WEEK} field indicating..  
597    public final static int WEDNESDAY = 4;  
598  
600+    * Value of the {@link #DAY_OF_WEEK} field indicating..  
603    public final static int THURSDAY = 5;  
604  
606+    * Value of the {@link #DAY_OF_WEEK} field indicating..  
609    public final static int FRIDAY = 6;  
610  
612+    * Value of the {@link #DAY_OF_WEEK} field indicating..  
615    public final static int SATURDAY = 7;  
616
```



# Calendar – Konstanten - Wochentage

Warum werden dann die Wochentage anders behandelt und dort bei 1 begonnen zu zählen.

Hier zwei dürftige Erklärungsversuche:

- Amerikanisches Datums Format und da beginnt die Woche mit dem Sonntag
  - *Okay, aber warum 1 statt 0?*
- Die Wochentage sind ein Menschliches Konstrukt und unterliegen keiner Ordnung
  - *Okayyyyy, aber warum 1 statt 0?*

Mir ist unbekannt ob es abschließend geklärt wurde, warum dies so durchgeführt wurde. Es führt beim Arbeiten mit der Pre 8 Date API immer wieder zu Verwirrungen.

# Arbeiten mit der Pre 8 DateTime API

- Bei dem Arbeiten mit der Pre 8 DateTime API musste also immer wieder darauf geachtet werden, dass Monate mit 0 und Wochentage mit 1 beginnen. Auch für das Abfragen von Tag des Monats wurde eine Nummerierung von 1 an geschaffen.
- Um das Arbeiten mit der DateTime API zu vereinfachen wurden immer mehr Helfer-, Sub- und Wrapperklassen geschaffen.
- Über die Jahre hinweg wurde die DateTime API also immer wieder erweitert und um Funktionalität aufgewertet. Einige Funktionen wurden als Deprecated markiert, um auf verbesserte Funktionen hinzuweisen.
- Zum Formatieren von Datumswerten werden 2 Klassen zur Verfügung gestellt. Beide arbeiten unterschiedlich sollen aber das gleiche Ergebnis erzielen.
  - *Eine folgt dem Factory Pattern, die andere wird mit einem Konstruktor erstellt. Was eine Erleichterung für den Programmierer darstellen sollte, „SimpleDateFormat extends DateFormat“, steuerte leider auch nur dazu bei, die Flut der DateTime API aufzublähen.*



# AUFTRITT JSR THREETEN

Eine Grundlegende Überarbeitung der Date und Time API.

# Date and Time API Java 8 SE

- Es wurde eine neue „zusätzliche“ **Date and Time API** geschaffen.
- Basieren auf den Erfahrungen die mit anderen Third-Party Bibliotheken gewonnen wurden, wurde die neue API erstellt.
  - *“Joda-Time is the de facto standard date and time library for Java prior to Java SE 8. Users are now asked to migrate to java.time (JSR-310).”*
- Die Konzepte und Arbeitsweisen der “Joda-Time-API”, “ICU”, “TIME and Money” und “Calender Date” wurden analysiert.
- JSR 310's API is built around the same five basic date/time concepts used in Joda Time:
  - *A Discrete Timeline*
  - *Instant*s
  - *Partial*s
  - *Duration*s
  - *Period*s
  - *Interval*s

# The Discrete Timeline

Like Joda Time, JSR 310 uses a discretized timeline: time is modeled as a sequence of consecutive instants separated by small, fixed durations.

The discrete timeline of JSR 310 has nanosecond resolution, so it is possible to express the time "One nanosecond after midnight on 01/01/08," but not "One picosecond after midnight on 01/01/08."

Each discrete nanosecond on the timeline is considered an instant -> nächste Folie

# Instants

- An instant is a specific point on a discretized timeline. An example of an instant is "January 7, 2008, at time 23:00:00.0 UTC." An instant can also be defined as a nanosecond offset from a standard epoch, such as "20,000,000 nanoseconds after January 1, 1970 at midnight UTC." Both of these descriptions define a single, unique point on the discrete timeline. Instants are different from `partials`, which define a set of moments on the timeline, rather than one unique moment.
- The JSR 310 API provides several classes that represent instants: `Instant`, `OffsetDateTime`, and `ZonedDateTime`, all of which implement the `ReadableInstant` interface. The `OffsetDateTime` class represents a date, time of day, and offset from coordinated universal time (UTC), such as `+1:00`. The similar `ZonedDateTime` class incorporates a time zone ID, such as `America/New_York`, instead of an offset. A given time zone may use several different offsets depending on the time of year; for example, the `America/New_York` time zone's offset is `-4:00` during daylight savings time and `-5:00` at other times. Thus, the `ZonedDateTime` class should be used when locale-specific time rules, such as daylight savings time, must be taken into account.
- The `ZonedDateTime` class provides several categories of methods for creating, accessing, and modifying instants. To create a new instance of `ZonedDateTime` representing the current system time in your computer's default time zone, you can use the `Clock.currentZonedDateTime()` factory method, as shown by the following example.

# Instants

- `Clock systemClock = Clock.system(); ZonedDateTime currentTime = systemClock.currentZonedDateTime();`
- To create a `ZonedDateTime` instance that represents a specific, predetermined date, you can use one of several `ZonedDateTime.dateTime` factory methods. The following example demonstrates the creation of a `ZonedDateTime` representing midnight on January 1, 2000, in your computer's default time zone.
- `Clock systemClock = Clock.system(); TimeZone defaultTimeZone = systemClock.timeZone(); int year = 2000; int month = 1; int day = 1; int hour = 0; int minute = 0; ZonedDateTime theDate = ZonedDateTime.dateTime(year, month, day, hour, minute, defaultTimeZone);`
- Instances of `OffsetDateTime` can be created in a manner similar to that of `ZonedDateTime`, except that a `ZoneOffset` rather than a `TimeZone` is passed into the `OffsetDateTime.dateTime` factory method. To get an instance of `ZoneOffset`, you can use the static `ZoneOffset.zoneOffset(int hours)` method, where `hours` is the hour offset from UTC.

# Partials

- A partial is an indication of date or time that is not sufficient to specify a specific, unique point on the timeline. For example, "June 7" is a partial, because it specifies a month and day, but it does not specify a year or time of day. Thus, the above partial is not sufficient to identify a unique point on the timeline. Because partials do not identify specific times, they cannot be expressed as nanosecond offsets from a standard epoch. Their definition is necessarily field-based, using calendar fields such as year, month, day of month, and time of day.
- Partials can be categorized based on the set of calendar fields that they define. For example, a partial that represents a yearly holiday might contain month and day fields, whereas a partial that represents a store's opening time might contain hour and minute fields. JSR 310 provides classes for commonly used partials, such as `MonthDay` (the aforementioned "holiday" partial), `LocalDate` (a date with no time or time zone), and `LocalTime` (a time with no time zone). To create an instance of `MonthDay` or one of the other ready-made partial classes, you can use one of the provided static factory methods. The example below is specific to `MonthDay`, but it can easily be adapted for one of the other partial classes.
- ```
//Create a MonthDay representing December 25 (Christmas)
int theMonth = 12; int theDay = 25;
MonthDay christmas = MonthDay.monthDay(theMonth, theDay);
```

# Durations

- A duration represents a length of elapsed time, defined to the nanosecond level; for example, "100,000 nanoseconds" is a duration. Durations are somewhat similar to periods, which also define a length of elapsed time; however, unlike periods, durations represent a precise number of elapsed nanoseconds.
- A duration can be added to an instant to return a new instant that is offset from the original instant by the number of nanoseconds in the duration. For example, if the duration "86,400,000,000,000 nanoseconds" (24 hours) is added to the instant "March 1, 2008, at midnight UTC," the resulting instant is "March 2, 2008, at midnight UTC."
- Durations may be created in two ways: by supplying a second, millisecond, or nanosecond timespan for the duration, or by providing starting and ending instants. The first method is appropriate for creating a duration of a predetermined length:
- ```
System.out.println("Enter a duration length in hours:"); Scanner s = new Scanner(System.in); int durationSeconds = 3600 * s.nextInt(); Duration d = Duration.duration(durationSeconds);
```
- Alternatively, you may wish to determine the duration between two predetermined instants. The `staticDuration.durationBetween(ReadableInstant startInclusive, ReadableInstant endExclusive)` factory method is useful in this case.
- ```
Clock systemClock = Clock.system(); ZonedDateTime instant1 = systemClock.currentZonedDateTime(); try{Thread.sleep(1000)}  
//Use up some time catch(InterruptedExcepiton e){System.out.println("Error")} ZonedDateTime instant2 =  
systemClock.currentZonedDateTime(); Duration d = Duration.durationBetween(instant1, instant2);
```



# Period

- Like durations, periods represent a length of elapsed time. Examples of periods are "4 years, 8 days," and "1 hour." As shown by these examples, periods are defined using calendar fields (years, days, hours, etc.), rather than by an exact number of nanoseconds.
- At first, periods and durations may seem to be different ways of expressing the same concept; however, a given period cannot be converted to an exact number of nanoseconds, as the following example of instant/period addition demonstrates. Instant/period addition adds the value of each of the period's calendar fields to the corresponding field of the instant. This is in contrast to instant/duration addition, which adds the duration's length in nanoseconds to the instant. At first glance, it may seem that adding an 86,400,000,000,000-nanosecond (24-hour) duration to a given instant should produce the same result as adding a period of "1 day," but this is not always the case, because the calendar field "day" does not have a fixed nanosecond length. Most days are 24 hours long, but some are longer or shorter due to daylight savings time. Adding a 24-hour duration to a instant will always advance the instant by exactly 24 hours, whereas adding a one-day period will advance the day by one, while leaving the time of day unchanged.
- For example, if the period "1 day" is added to the instant "March 9, 2008, at midnight EST," the resulting field-based addition produces "March 10, 2008, at midnight EST." However, if a 24-hour duration is added to the instant "March 9, 2008, at midnight EST," the result is "March 10, 2008, at 01:00:00.0 EST." The discrepancy stems from the fact that daylight savings time begins at 02:00:00 EST on March 9, 2008; thus, this day is only 23 hours long.

# Period

- JSR 310 provides period functionality with several classes, the most important of which are `Period`, `Periods.Builder`, and the subclasses of `Periods.Builder`: `Periods.SecondsBuilder`, `Periods.MinutesBuilder` (which is a subclass of `SecondsBuilder`), and so on, up to `Periods.YearsBuilder`.
- To create an instance of `Period`, you must first obtain an instance of `Periods.Builder` using the static `Periods.periodBuilder()` method. This method returns a `Periods.YearsBuilder` object, which is an indirect subclass of `Periods.Builder`. The `Periods.YearsBuilder` class provides a method, `years(int numYears)`, that adds `numYears` years to the period being built. Methods provided by the direct and indirect superclasses of `YearsBuilder` (`MonthsBuilder`, `DaysBuilder`, `HoursBuilder`, `MinutesBuilder`, and `SecondsBuilder`) allow other calendar fields to be added to the period. For example, the `MinutesBuilder.minutes(int numMinutes)` adds `numMinutes` minutes to the period being built. All of these methods return `this`, allowing periods with multiple calendar fields to be built with a single statement. Note that it is possible to subtract from a period by passing a negative integer into the appropriate builder method. Once all desired fields have been added to the period, the `Periods.Builder.build()` method is called to return the completed instance of `Period`.
- ```
//Build a period representing "8 years, 3 months." Period thePeriod =  
Periods.periodBuilder().years(8).months(3).build();
```

# Intervals

- An interval represents a period of time between two instants on the timeline. Thus, "midnight UTC on January 1, 2007 (inclusive) to midnight UTC on January 1, 2008 (exclusive)" is an interval. The interval can be inclusive or exclusive with regard to the starting and ending instants.
- JSR 310 provides the `InstantInterval` class to represent intervals. To create an instance of `InstantInterval`, you can use one of several factory methods:

```
//Create some intervals. ZonedDateTime time1 = systemClock.currentZonedDateTime();
try{Thread.sleep(1000)} //Use up some time catch(InterruptedException
e){System.out.println("Error")} ZonedDateTime time2 =
systemClock.currentZonedDateTime(); //Create an interval with inclusive start and exclusive
end. InstantInterval interval1 = InstantInterval.intervalBetween( time1.toInstant(),
time2.toInstant()); //Create an interval with exclusive start and inclusive end. boolean
startInclusive = false; boolean endInclusive = true; InstantInterval interval2 =
InstantInterval.intervalBetween( time1.toInstant(), startInclusive, time2.toInstant(),
endInclusive);
```

# Java und die Nummerierung

- Der Monat beginnt nun auch bei 1. Mit der neuen API wurde eine Angleichung der Monatsnummerierung vorgenommen.
  - *Januar ist jetzt Monat 1.*
  - *Sonntag ist Tag 1*
  - *Der erste Tag im Monat wie auch im Jahr beginnt auf mit 1.*

# WORKING WITH CALENDAR DATA

OCA Objective 9.3

# OCA Relevante Klassen und Interfaces

- Aus dem gesamten Paket der neuen Date and Time API wird nur ein Teil im OCA Objective 9.3 abgefragt. Es wird wissen zu:
  - ***java.time.LocalDate***: Erzeugt „immutable“ Objekte welche ein bestimmtes Datum darstellen.
  - ***java.time.LocalTime***: Erzeugt „immutable“ Objekte welche eine bestimmte Zeit darstellen.
  - ***java.time.LocalDateTime***: Erzeugt „immutable“ Objekte von denen jedes ein bestimmtes Datum und eine bestimmte Uhrzeit darstellt. Es stellt eine Fusion aus *LocalDate* und *LocalTime* dar.
  - ***java.time.format.DateTimeFormatter***: Wird benutzt und die vorher genannten Klassen für die Ausgabe zu formatieren oder um Eingaben zu Analysieren und zu konvertieren.

# OCA Relevante Klassen und Interfaces

## ■ Weitere Relevante Klassen

- ***java.time.Period***: Erzeugt „immutable“ Objekte welche einen Zeitraum darstellen.
  - Ein Jahr, 5 Monate und 4 Tage.
  - Für Zeitabschnitte kleiner als einen Tag können mit der `java.time.Duration` Klasse erzeugt werden. *Duration ist kein bestandteil des Exams.*
- ***java.time.temporal.TemporalAmount***: Dieses Interface wird von *Period* implementiert. Wenn *Period* benutzt wird um Kalenderobjekte zu bearbeiten, werden Methoden verwendet die Objekte verwenden, die mit *TemporalAmount* in einer IS-A Beziehung stehen.



# FACTORY PATTERN



# Date and Time API – Factory Pattern

- Eine Klasse die keinen Öffentlichen Konstruktoren besitzt, aber dafür Statische Methoden die intern eine Instanz erzeugen und diese an die Aufrufende Stelle liefern werden Factory Classes genannt.

Die entsprechenden Methoden werden Factory methods genannt.

- *Die Klassen können nicht direkt mit dem Schlüsselwort „new“ instanziiert werden.*
- *Instanzen werden mit sogenannten Factory Methoden erzeugt.*

## Exam Hinweis:

Sofern eine Exam Frage das Thema Date and Time betrifft. Schauen Sie nach dem Schlüsselwort “new”. Daran sehen Sie das der Code nicht Kompiliert:

```
LocalDateTime d1 = new LocalDateTime(); // Kompiliert nicht
```

Merken Sie sich, das die Exam Relevaten Date and Time Klassen „Factory“ Methoden verwenden.

# Einheitlicher Aufbau

- Dem Factory Pattern der neuen Date and Time API haben wir es zu verdanken, dass keine Mischungen aus new und Factory methoden mehr vorhanden sind.
- Durch ein einheitliches Software Design besitzen die 3 wichtigen Klassen einheitliche Factory Methoden.
- Somit finden wir unterschiedlich viele from(), of(), ofXXX() und parse() Methoden in den Klassen, alle mit dem Ziel eine Instanz zu erzeugen. Der Aufbau der Methoden ist jedoch immer sehr ähnlich.
- Die Methoden of(), ofXXX() usw. zeigen sich immer wieder in den Neuerungen welche Oracle seit Java 8 eingeführt hat.

LOCALDATE



# java.time.LocalDate

- Erzeugt „immutable“ Objekte welche ein bestimmtes Datum darstellen.
- Darüber hinaus stellt diese Klasse Methoden bereit, mit denen neue Instanzen basierend auf den Werten einer anderen Instanz erzeugt werden.
- LocalDate-Objekte beinhalten nur Datumsangaben ohne Zeit.

```
LocalDate date1 = LocalDate.of(2017, 1, 31);  
Period period1 = Period.ofMonths(1);  
System.out.println(date1);  
date1.plus(period1); // new value is lost  
System.out.println(date1);  
LocalDate date2 = date1.plus(period1); // new value is captured  
System.out.println(date2);
```

# java.time.LocalDate

- Folgende Factory Methoden stehen zur Verfügung um neue Instanzen zu erzeugen
  - *from()*
  - *now()* – mit 3 überladenen Methoden
  - *of()* – mit 2 überladenen Methoden – *OCA Exam Relevant*
  - *ofEpochDay()*
  - *ofYearDay()*
  - *parse()* – mit 2 überladenen Methoden
- In Summe haben wir somit 10 verschiedenen Möglichkeiten uns Instanzen von `LocalDate` zu bilden.
- Für das OCA Exam Konzentrieren wir uns jedoch nur auf die `.of()` Methoden.

Nein nicht alle sind relevant

# java.time.LocalDate

```
public static LocalDate of(int year, Month month, int dayOfMonth)
```

- Diese of() Methode nimmt 3 Parameter entgegen und liefert etwas vom Typ LocalDate zurück.
- Die Parameter:
  - *int year* – Das Jahr das dargestellt werden soll
    - Gültige werte von MIN\_YEAR bis MAX\_YEAR (-999\_999\_999 bis 999\_999\_999)
  - *Month month* – Der Monat des Jahres der dargestellt werden soll.
    - Gültige werte werden von dem ENUM java.time.Month erzeugt.
  - *int dayOfMonth* – Tag des Monats der dargestellt werden soll.
    - Gültige Werte 1 – 28,30,31 Monats abhängig.
      - Das Überschreiten eines gültigen wertes für den Monat erzeugt eine *DateTimeException*.

```
Exception in thread "main" java.time.DateTimeException: Invalid date 'FEBRUARY 30'
```



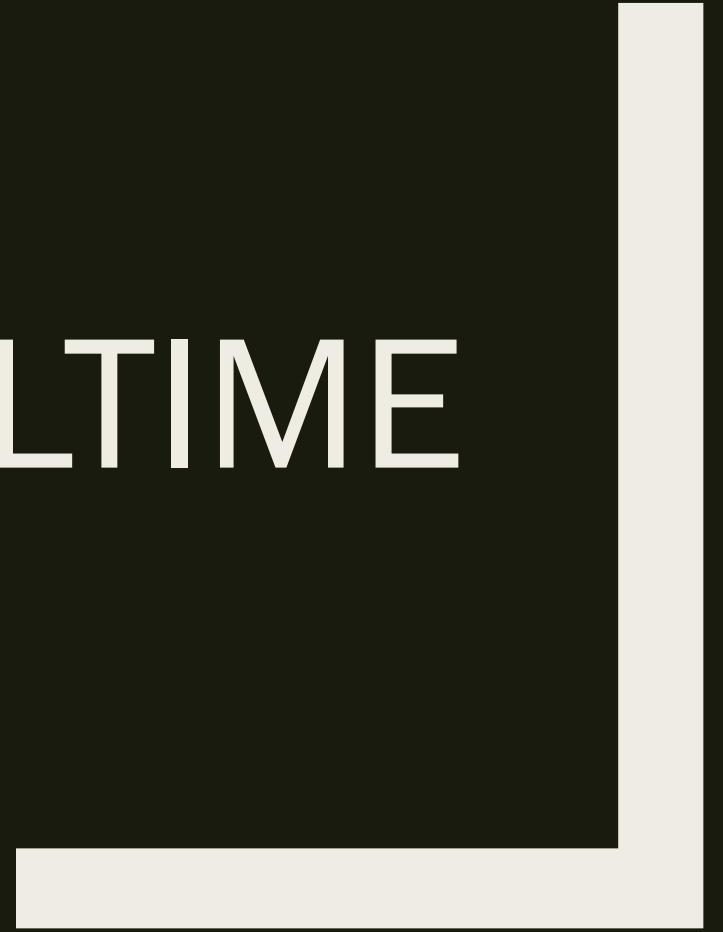
# java.time.LocalDate

```
public static LocalDate of(int year, int month, int dayOfMonth) {
```

- Diese of() Methode nimmt 3 Parameter entgegen und liefert etwas vom Typ LocalDate zurück.
- Die Parameter:
  - *int year* – Das Jahr das dargestellt werden soll
    - Gültige werte von MIN\_YEAR bis MAX\_YEAR (-999\_999\_999 bis 999\_999\_999)
  - *int month* – Der Monat des Jahres der dargestellt werden soll.
    - Gültige werte werden von dem ENUM java.time.Month erzeugt.
  - *int dayOfMonth* – Tag des Monats der dargestellt werden soll.
    - Gültige Werte 1 – 28,30,31 Monats abhängig.
      - Das Überschreiten eines gültigen wertes für den Monat erzeugt eine DateTimeException.

```
Exception in thread "main" java.time.DateTimeException: Invalid date 'FEBRUARY 30'
```

LOCALTIME



# java.time.LocalDateTime

- Erzeugt „immutable“ Objekte welche eine bestimmte Zeit darstellen.
- Darüber hinaus stellt diese Klasse Methoden bereit, mit denen neue Instanzen basierend auf den Werten einer anderen Instanz erzeugt werden.
- LocalDateTime-Objekte beziehen sich nur auf Stunden, Minuten, Sekunden und Sekundenbruchteile.

```
LocalTime lt = LocalDateTime.now();  
System.out.println(lt);
```

```
lt = LocalDateTime.of(12, 12);  
// Obtains an instance of LocalDateTime from an hour and minute.  
System.out.println(lt);
```

```
lt = LocalDateTime.of(12, 12, 12);  
// Obtains an instance of LocalDateTime from an hour and minute.  
System.out.println(lt);
```

# java.time.LocalDateTime

- Folgende Factory Methoden stehen zur Verfügung um neue Instanzen zu erzeugen
  - *from()*
  - *now()* – mit 3 überladenen Methoden
  - *of()* – mit 3 überladenen Methoden – *OCA Exam Relevant*
  - *ofSecondOfDay()*
  - *ofNanoOfDay()*
  - *parse()* – mit 2 überladenen Methoden
- In Summe haben wir somit *viele* verschiedenen Möglichkeiten uns Instanzen von `LocalTime` zu bilden.
- Für das OCA Exam Konzentrieren wir uns jedoch nur auf die `.of()` Methoden.

*Nein nicht alle sind relevant*

# Die 3 of() Methoden

- Source Demo02LocalTime.java

```
LocalTime lt = LocalTime.now();  
lt = LocalTime.of(12, 12);  
// Obtains an instance of LocalTime from an hour and minute.  
  
lt = LocalTime.of(12, 12, 12);  
// Obtains an instance of LocalTime from an hour and minute.  
  
lt = LocalTime.of(12, 12, 12, 12);  
// Obtains an instance of LocalTime from an hour, minute, second and  
// nanosecond.
```

# LOCALDATETIME

Die Fusion aus Datum und Zeit



# java.time.LocalDateTime

- Folgende Factory Methoden stehen zur Verfügung um neue Instanzen zu erzeugen
  - *from()*
  - *now()* – mit 3 überladenen Methoden
  - *of()* – mit 7 überladenen Methoden – *OCA Exam Relevant*
  - *ofInstant()*
  - *ofEpochSecond()*
  - *parse()* – mit 2 überladenen Methoden
- In Summe haben wir somit *noch mehr* Möglichkeiten uns Instanzen von `LocalDateTime` zu bilden.
- Für das OCA Exam Konzentrieren wir uns jedoch nur auf die `.of()` Methoden.

*Nein nicht alle sind relevant*

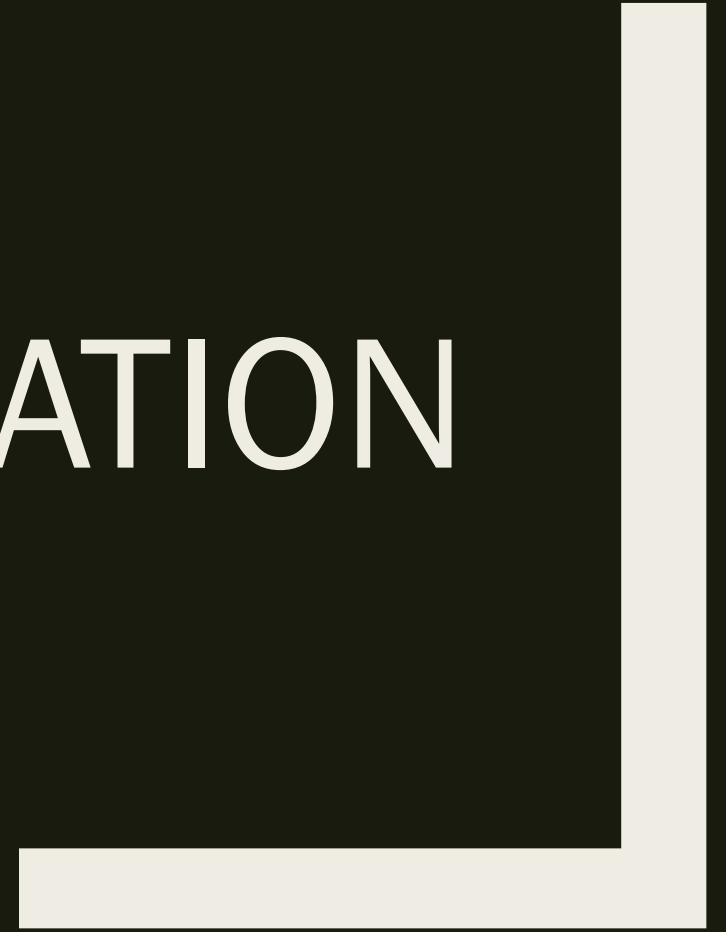


# Die ... of() Methoden

- Source Demo03LocalDateTime.java
- Da es sich um eine Fusion der Klassen Date und Time handelt, sind die Factory Methoden ebenfalls eine Fusion aus den beiden Klassen

```
LocalDateTime ldt = LocalDateTime.now();  
ldt = LocalDateTime.of(LocalDate.now(), LocalTime.now());  
// Obtains an instance of LocalDateTime from a date and time.  
  
ldt = LocalDateTime.of(2018, 12, 24, 15, 30);  
// Obtains an instance of LocalDateTime from year, month, day, hour and  
// minute, setting the second and nanosecond to zero.  
  
ldt = LocalDateTime.of(2018, Month.DECEMBER, 24, 15, 30);  
//ldt = LocalDateTime.of(2018, 12, 24, 15, 30);  
  
//ldt = LocalDateTime.of... usw usw usw
```

MANIPULATION



# Manipulation der Daten

- Die Klassen arbeiteten nach ISO-8601.
  - [https://de.wikipedia.org/wiki/ISO\\_8601](https://de.wikipedia.org/wiki/ISO_8601)  
*ISO 8601 ist ein internationaler Standard der ISO, der Empfehlungen über numerische Datumsformate und Zeitangaben enthält.*
- Manipulieren mit den withxxx() Methoden.
  - *Achtung, die withXXX methoden erzeugen eine neue Instanz mit veränderten Grundwerten. Es sind ersetzende Factory Methoden.*
- Manipulieren mit den MinusXXX / Plus Methoden
  - *Diese Methoden nehmen die übergebene Instanz als Basis und berechnen je nach Methode neue Instanzen.*
- Da die Instanzen der 3 Klassen Immutable sind, wird immer eine neue Instanz erzeugt.

# Wer kann was? PlusXXX / MinusXXX

	LocalDate	LocalTime	LocalDateTime
plusYears/minusYears	Ja	Nein	Ja
plusMonths/minusMonths	Ja	Nein	Ja
plusWeeks/minusWeeks	Ja	Nein	Ja
plusDays/minusDays	Ja	Nein	Ja
plusHours/minusHours	Nein	Ja	Ja
plusMinutes/minusMinutes	Nein	Ja	Ja
plusSeconds/minusSeconds	Nein	Ja	Ja
plusNanos/minusNanos	Nein	Ja	Ja

PERIOD



# java.time.Period

- Diese Klasse wird verwendet, um unveränderliche Objekte zu erstellen, die einen Zeitraum repräsentieren.
  - *beispielsweise "zwei Jahre, vier Monate und fünf Tage".*
- Diese Klasse arbeitet in Jahren, Monaten und Tagen.
  - *Wenn Sie Zeitabschnitte in Schritten von weniger als einem Tag (z. B. Stunden und Minuten) darstellen möchten, können Sie die Klasse java.time.Duration verwenden.*
  - *java.time.Duration ist nicht bestandteil des OCA Exams.*
- Die Klasse Period steht zum Interface TemporalAmount in einer IS-A Beziehung
  - *Daher kann Sie für Methoden die eine Instanz vom Typ TemporalAmount als Parameter anfordern genutzt werden.*

# of() und ofXXX();

- Die Klasse Period kommt mit einigen, wie verwunderlich, of() und ofXXX() Methoden daher.

```
Period periodDays = Period.ofDays(5);  
//Obtains a Period representing a number of days.  
Period periodWeeks = Period.ofWeeks(12);  
//Obtains a Period representing a number of weeks.  
Period periodMonth = Period.ofMonths(5);  
//Obtains a Period representing a number of months.  
Period periodYears = Period.ofYears(5);  
//Obtains a Period representing a number of years.
```



# Nutzen von Period

- Die Instanz von Period, kann der Überladenen ***minus bzw. plus(TemporalAmount amountToAdd)*** Methode übergeben werden

```
LocalDate date1 = LocalDate.of(2017, 12, 24);  
System.out.println(date1.plus(periodDays));  
System.out.println(date1.minus(periodWeeks));  
System.out.println(date1.plus(periodMonth));  
System.out.println(date1.minus(periodYears));
```



# DATETIMEFORMATTER



# java.time.DateTimeFormatter

- Diese Klasse wird von den gerade beschriebenen Klassen verwendet, um Datums- / Uhrzeitobjekte für die Ausgabe zu formatieren sowie Eingaben zu analysieren und sie in Datums- / Uhrzeitobjekte zu konvertieren.
- DateTimeFormatter-Objekte sind ebenfalls unveränderlich.
- Es stehen einige ISO\_XXX Konstanten zur Verfügung, womit einfach ISO Konforme Datums- und Zeitformate erstellt werden können.
- Des Weiteren sind mehrere ofXXX Methoden vorhanden.
  - *ofPattern()* -2 überladene Methoden <- **nur diese sind OCA Relevant**
  - *ofLocalizedDate()*
  - *ofLocalizedTime()*
  - *ofLocallizedDateTime()* – 2 überladene Methoden

# java.time.DateTimeFormatter

- Mit den erzeugten Instanzen des DateTimeFormatter lassen sich wunderbar die Klassen Date, Time und DateTime formatieren.
- Jedoch ist vorsicht geboten, welche Art Instanz erzeugt wurde und auf welche Der 3 Klassen diese angewendet werden soll.
  - *Ein Time Formatter kann nicht auf eine Date Instanz ausgeführt werden*
  - *Ein Date Formatter kann nicht auf eine Time Instanz ausgeführt werden*
  - *Aber eine DateTime Instanz kann einen Formatter mit Date oder Time Pattern verwenden.*
  - *Time wie auch Date können auch nicht mit einer Formatter mit DateTime Pattern ausgeführt werden.*
- Die Klasse Locale wird vom DateTimeFormatter unterstützt.

# LOCALE

Sprachen und Regionen



# Sprachen und Regionen

- Locale-Objekte repräsentieren geografische, politische oder kulturelle Regionen.
- Sprache und die Region müssen getrennt werden Region oder Land gibt die Sprache nicht eindeutig vor:
  - *Beispiel: Kanada in der Umgebung von Quebec französische Ausgabe ist relevant, die unterscheidet sich von der englischen.*
- Sprach-Objekte werden immer mit dem Namen der Sprache und optional mit dem Namen des Landes beziehungsweise einer Region erzeugt.
- Im Konstruktor werden Länderabkürzungen angegeben

```
import java.util.Locale;
```

```
Locale greatBritain = new Locale( "en", "GB" );  
Locale french       = new Locale( "fr" );
```

```
System.out.println(greatBritain); //en_GB  
System.out.println(french);       //fr
```

# Konstrukturen der Klasse Locale

- `Locale (String language)`  
Erzeugt ein neues Locale-Objekt für die Sprache (language), die nach dem ISO-693-Standard gegeben ist.
- `Locale(String language, String country)`  
Erzeugt ein Locale-Objekt für eine Sprache (language) nach ISO 693 und ein Land (country) nach dem ISO-3166-Standard

```
//Aktuell eingestellte Sprache  
System.out.println(Locale.getDefault());    //de_DE
```

# Konstanten für Länder und Sprachen

- Die Locale-Klasse besitzt Konstanten für häufig auftretende Länder und Sprachen
- Konstante für Großbritannien `Locale.UK` statt der Instantiierung `new Locale("en", "GB")`
- Konstante für Deutschland

```
Locale german = new Locale("de", "De");  
System.out.println(Locale.GERMAN);    //de  
System.out.println(Locale.GERMANY);    //de_DE  
  
System.out.println(german);            //de_DE
```