Date and Time

Module 11

Java 8 Date Time API

- Java 8 introduced a new date and time api
- Issues with the old apis (Date and Calendar):
  - Not thread safe
  - Poorly designed and lacking important methods
  - Can only partially handle timezones
- New package java.time.*

Great example of the power of open source: Joda Time library emerged, out of frustration, from the developer community which later heavily inspired the new time & date api design in Java 8 - read more

ISO 8601: Human-Readable time representation

Format specification: Overview from w3

- International standard to agree on an unambiguous way of representing time and date
- Complete date plus hours, minutes, seconds and a decimal fraction of a second and time zone designator

1997 - 07-16 T19: 20:30.45 +01:00

YYYY - MM-DD Thh:mm:ss.s TZD

Decimal fractions

1ms -- .001000
10ms -- .010000
100ms -- .100000
1µs -- .000001

ISO 8601: Human-Readable time representation

Rules and things to look out for

- Time units are sorted from largest to smallest unit
  - ✗ 1997-**16-07**T19:20:30.45
- It is allowed to drop number values from right to left for reducing precision
  - ✗ 1997-**16**T19:20:30
  - ☑ 1997-07T19:20:30
- Time units conform to fixed length
  - ☑ 1997-16T07:20:30
  - ✗ 1997-16T**7**:20:30
- Optional use of separators '-', ':'
  - ☑ 1997-07-16T19:20:30.45
  - ☑ 19970716T192030.45
- Allows to specify UTC offsets

YYYY - MM-DD Thh:mm:ss.s TZD

☑ 1997 - 07-16 T19: 20:30.45 +01:00

LocalDate

- LocalDate represents a date in ISO format (yyyy-MM-dd) without time

```
LocalDate localDate=LocalDate.now();// date of today
LocalDate.of(2019,1,07);// date of January 7, 2019
LocalDate.parse("2019-01-07");// date of January 7, 2019
```

Working with LocalDate

```
LocalDate.now().plusDays(1); // tomorrow
LocalDate.now().minus(2, ChronoUnit.MONTHS); // 2 months ago
LocalDate.now().isLeapYear(); // if this year is a leap year

// if the first date is before the second date
LocalDate.of(2019,1,1).isBefore(LocalDate.of(2019,2,2));

// Constants like
LocalTime [] constants = {LocalTime.MAX, LocalTime.MIN, LocalTime.NOON, LocalTime.MIDNIGHT};
```

LocalTime

- LocalTime represents a time without a date

```
LocalTime now = LocalTime.now(); // the time now
LocalTime evening = LocalTime.of(19,30); // time of 19:30
LocalTime morning = LocalTime.parse("07:05"); // time of 07:05
```

Working with LocalTime

```
LocalTime t = LocalTime.now().plus(2, ChronoUnit.HOURS); // in two hours
LocalTime.now().getHour(); // get current hour

// if 06:30 is before 07:30
LocalTime.parse("06:30").isBefore(LocalTime.parse("07:30"));
```

LocalDateTime

- LocalDateTime represents a combination of date and time

```
LocalDateTime.now(); // now
LocalDateTime.now().plusDays(1); // tomorrow at this time
LocalDateTime.now().minusHours(2); //2 hours ago

// the date and time of January 7th 2019 at 06:30
LocalDateTime.of(2019,Month.JANUARY,07,06,30);
```

Period

- The Period class is used to modify values of a date or to obtain the difference between two dates

```
LocalDate date = LocalDate.now().plus(Period.ofDays(5)); // 5 days from today

// difference between the two dates in days
int numberOfDays = Period.between(LocalDate.now(), date).getDays();
```

Duration

- Duration class is used to represent a duration

```
// the time in 9 seconds
LocalTime time = LocalTime.now().plus(Duration.ofSeconds(9));

// Works also with LocalDateTime
LocalDateTime futureDateTime = LocalDateTime.now().minusHours(2);
Duration duration = Duration.between(LocalDateTime.now(), futureDateTime);
boolean isTargetDateInPast = duration.isNegative();
long hours = duration.abs().toHours();
```

Date and Time formatting

- Date and time can be formatted with constants or patterns

```
// January 7th, 2019, at 9.00 am
LocalDateTime localDateTime = LocalDateTime.of(2019,1,7,9,0);

// 2019-01-07T09:00:00
localDateTime.format(DateTimeFormatter.ISO_DATE_TIME);

// 2019/01/07 09:00
localDateTime.format(DateTimeFormatter.ofPattern("yyyy/MM/dd hh:mm"));
```

Timezones

A quick refresher

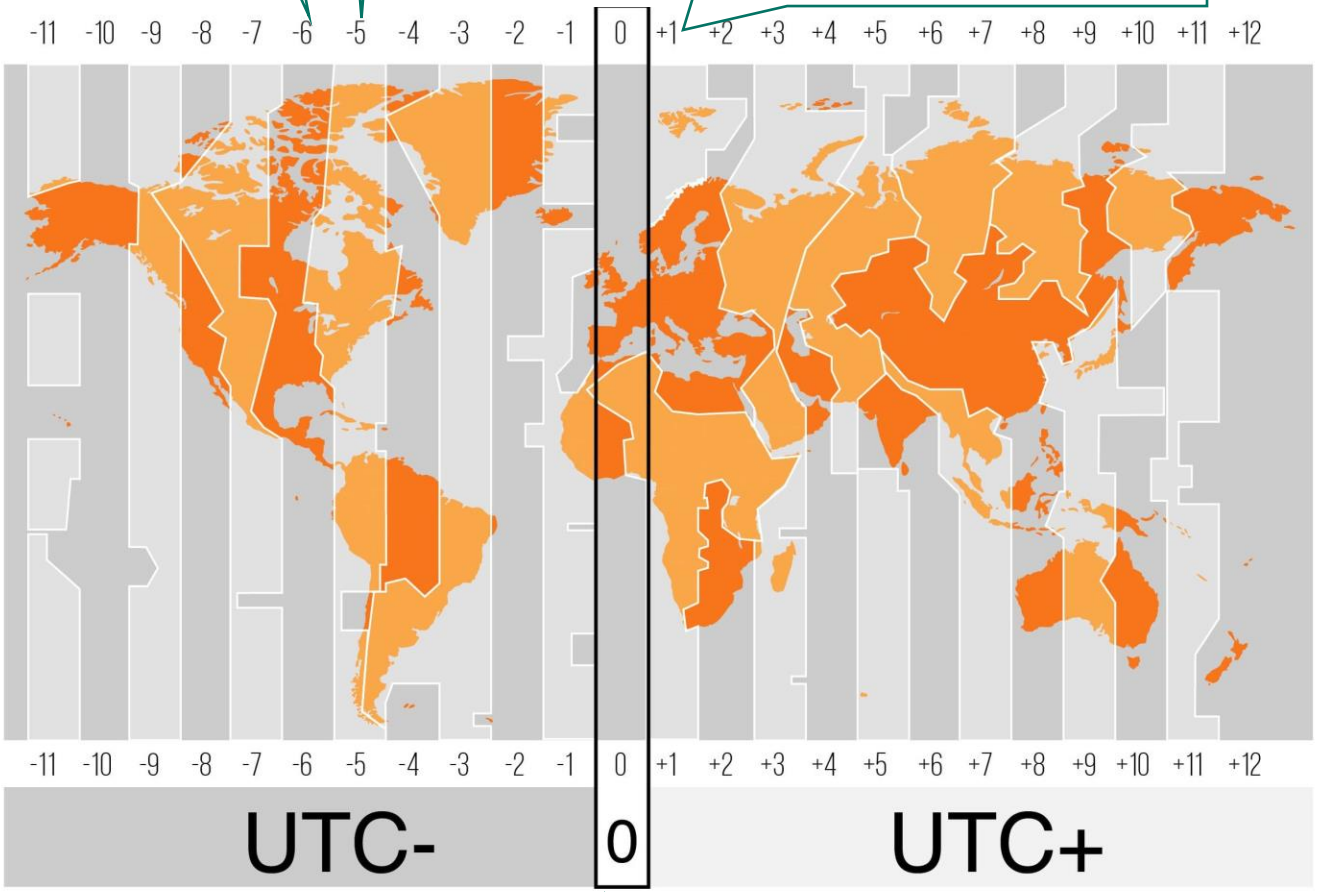
- World's primary time standard: UTC (Coordinated Universal Time)
- A timezones can have different UTC offsets depending on region, country, daylight savings: *St. John's, NL, Canada (GMT-2:30)*
- *To keep track of variations and changes timezones are distributed in a central **time zone database** maintained by* iana

see how those changes are incorporated into Java releases here

CST (Central Standard Time)

EST (Eastern Standard Time)

ECT (European Central Time)



Source: https://24timezones.com/cms-static/images/uploads/time-zone2-min.jpg

GMT (Greenwich Mean Time)

(!) UTC/GMT used interchangeably

Two representation of time zones in Java

- 1: Fixed offsets - a fully resolved offset from UTC/Greenwich, that uses the same offset for all local date-times

```
ZoneOffset zoneOffset = ZoneOffset.of("+01:00");
```

- 2: Geographical regions - an area where a specific set of rules (ZoneRules) for finding the offset from UTC/Greenwich apply

Based on time zone database

```
ZoneId zoneId = ZoneId.of("Europe/Berlin");
```

- Extension of ISO 8601

This is not part of the ISO standard norm

```
ZonedDateTime.parse("2022-11-12T10:10:00+01:00[Europe/Berlin]");
OffsetDateTime.parse("2022-11-12T10:10:00+01:00");
```

ZonedDateTime & OffsetDateTime

```
// ZonedDateTime
ZonedDateTime.of(LocalDateTime.now(), ZoneId.of("UTC"));
LocalDateTime.now().atZone(ZoneId.of("UTC"));
ZonedDateTime.parse("2022-11-12T10:10:00+01:00[Europe/Berlin]");

// OffsetDateTime
OffsetDateTime.of(LocalDateTime.now(), ZoneOffset.of("+01:00"))
LocalDateTime.now().atOffset(ZoneOffset.of("-05:00"));
OffsetDateTime.parse("2022-11-12T10:10:00+01:00");
```

Working with ZonedDateTime & OffsetDateTime

- Convert between timezones

```
// ZonedDateTime
ZonedDateTime zonedDateTime = ZonedDateTime.of(LocalDateTime.now(),, ZoneId.of("Europe/Paris"));
zonedDateTime.withZoneSameInstant(ZoneId.of("Asia/Ho_Chi_Minh"));

// OffsetDateTime
OffsetDateTime offsetDateTime = OffsetDateTime.of(LocalDateTime.now(), ZoneOffset.of("+01:00"));
OffsetDateTime.of(offsetDateTime.toLocalDateTime(),ZoneOffset.UTC);
```

- Same methods as other time classes

```
zonedDateTime.toOffsetDateTime(); //convert
zonedDateTime.getHour();
zonedDateTime.plusDays(12);
```

Timestamps

A machine readable time format

- ISO 8601 standard is designed for humans to read, timestamps are for machines
- Unix timestamp
  - also known as epoch time, unix epoch, POSIX time
  - count of seconds since the epoch of the first moment of 1970 UTC (not counting leap seconds)
- Advantage: Everywhere in the world the same, no time offsets needed
- When to use: For instance in database fields, logging or measuring elapsed time
- But once you need to display date & time to user convert to human readable format like ISO 8601

Working with timestamps: Instant class

Academy

(!) *always use long (64-bit). Integer representation (32-bit) will overflow soon (read further on the "Year 2038 problem")*

- Unix timestamps in seconds are presented as **long**
- For higher precision the running second is represented as **int** in nanoseconds
- Timestamps before 1970 are represented as negative offsets

```
Instant unixTimestamp = Instant.ofEpochSecond(1660979930L);
long timeStampInSeconds = Instant.now().getEpochSecond();
int timeStampNanoSecondPortion = Instant.now().getNano();

// Measure elapsed time
Instant start = Instant.now();
// CODE HERE
Instant finish = Instant.now();
long timeElapsed = Duration.between(start, finish).toMillis();
```

Working with legacy code

```
// Converting to new time & date representations
LocalDateTime.ofInstant(new Date().toInstant(), ZoneId.systemDefault());

LocalDateTime.ofInstant(new GregorianCalendar().toInstant(),
ZoneId.systemDefault());
```

Date-time types in Java	Modern class	Legacy class
Moment in UTC	java.time. **Instant**	java.util. ~~**Date**~~ java.sql. ~~**Timestamp**~~
Moment with offset-from-UTC (hours-minutes-seconds)	java.time. **OffsetDateTime**	(lacking)
Moment with time zone (Continent/Region)	java.time. **ZonedDateTime**	java.util. ~~**GregorianCalendar**~~ javax.xml.datatype. ~~**XMLGregorianCalendar**~~
Date & Time-of-day (no offset, no zone) Not a moment	java.time. **LocalDateTime**	(lacking)
Date only (no offset, no zone)	java.time. **LocalDate**	java.sql. ~~**Date**~~
Time-of-day only (no offset, no zone)	java.time. **LocalTime**	java.sql. ~~**Time**~~
Time-of-day, with offset (impractical & unused) (matches SQL-standard TIME_WITH_TIMEZONE)	java.time. **OffsetTime**	(lacking)

Use at your own risk. Updated 2020-06-03.

AW Academy