# Daty i czasy

# Spis treści

Date & time . . . 1
LocalDate . . . 1
LocalTime . . . 2
LocalDateTime . . . 2
OffsetDateTime . . . 3
ZonedDateTime . . . 3
Instant . . . 4
Manipulowanie datami . . . 5
Period i Duration . . . 6
  Period . . . 6
  Duration . . . 6
Formatowanie . . . 7

Zapiski prowadzącego Karola Rogowskiego i uczestnika Bootcampu Zajavka Bartek Borowczyk aka Samuraj Programowania.

# Date & time

Daty i czasy. Generalnie dosyć przydatna rzecz, często może być tak, że w bazie danych zapisujemy, kiedy jakiś użytkownik coś zrobił. A potem używamy tej daty, żeby policzyć, kiedy mu coś wygasa. To tak z życia, ale po kolei.

Uczulam, że to, co teraz zostanie poruszone, zostało wprowadzone w Javie 8. We wcześniejszych wersjach nie da się tego używać. W Javie 8 twórcy doszli do wniosku, że posprzątają API Javy (czyli zestaw klas i metod), które odpowiada za daty i czasy, bo wcześniejsze było mało przyjazne.

Wymienimy teraz 3 klasy z przedrostkiem local. Dlatego są one local, bo nie niosą ze sobą żadnego kontekstu mówiącego, w którym miejscu na świecie się to stało. Jest to data lub czas lokalny. Jak powiem `LocalDate 02.09.2020`, to będzie to `2 Wrzesień 2020` w Polsce i tak samo `2 Wrzesień 2020` w Stanach. Jeżeli powiem `LocalTime 20:12` to będzie to `20:12` w Polsce i `20:12` w Japonii. No ale każdy ma swoją strefę czasową i w sumie po takiej informacji, nie za bardzo wiadomo, kiedy coś się stało, czy o `20:12` u nas, czy o `20:12` w Japonii. Dlatego też poruszymy kwestię stref czasowych, ale po kolei.

# LocalDate

`LocalDate` to klasa, która przechowuje tylko datę. Jest to tylko data, beż żadnego kontekstu co i gdzie (w jakiej strefie czasowej). Obiekt `LocalDate` możemy utworzyć w poniższy sposób (ta klasa ma prywatny konstruktor, można utworzyć jej instancję tylko przez statyczną metodę `of()`):

```
// 10.12.2010
System.out.println(LocalDate.of(2020, Month.DECEMBER, 10));
// 10.2.2010
System.out.println(LocalDate.of(2020, 2, 10));
```

Klasa `Month` to `enum`, zatem możemy wypisać wszystkie jego wartości:

```
Month[] months = Month.values();
```

# LocalTime

`LocalTime` to klasa, która przechowuje tylko godzinę. Jest to tylko godzina, bez żadnego kontekstu, kiedy dokładnie i gdzie (w jakiej strefie czasowej).

Ta klasa również ma prywatny konstruktor, możemy utworzyć jej instancję przez metodę `of()`:

```
// 10:20
System.out.println(LocalTime.of(10, 20));
// 10:20:05
System.out.println(LocalTime.of(10, 20, 5));
```

# LocalDateTime

`LocalDateTime` to klasa, która jest kombinacją dwóch poprzednich. Zawiera datę i godzinę, ale bez kontekstu o strefach czasowych. Czyli `LocalDateTime` w Polsce i w Stanach Zjednoczonych oznacza kompletnie inny punkt w czasie.

Ta klasa również ma prywatny konstruktor, możemy utworzyć jej instancję przez metodę `of()`:

```
// 2020-04-21T20:10
System.out.println(LocalDateTime.of(2020, Month.APRIL, 21, 20, 10));
// 2020-04-21T20:10:10
System.out.println(LocalDateTime.of(2020, Month.APRIL, 21, 20, 10, 10));
// 2020-04-21T20:10:30
System.out.println(LocalDateTime.of(2020, Month.APRIL, 21, 20, 10, 30));
// 2020-11-21T20:10
System.out.println(LocalDateTime.of(2020, 11, 21, 20, 10));
// 2020-11-21T20:10:10
System.out.println(LocalDateTime.of(2020, 11, 21, 20, 10, 10));
// 2020-11-21T20:10:30
System.out.println(LocalDateTime.of(2020, 11, 21, 20, 10, 30));
// 2010-12-30T04:10
System.out.println(LocalDateTime.of(LocalDate.of(2010, 12, 30), LocalTime.of(4, 10)));
```

Zwróć uwagę, że data od czasu oddzielona jest wielką literą '**T**'. Taka jest reprezentacja `toString()` tych klas.

Każda z powyższych klas może zostać zainicjowana 'teraz' poprzez następujące metody:

```
LocalDate ld = LocalDate.now();
LocalTime lt = LocalTime.now();
LocalDateTime ldt = LocalDateTime.now();
```

# OffsetDateTime

`OffsetDateTime` to klasa, która reprezentuje datę w połączeniu z czasem z uwzględnieniem przesunięcia godzinowego wynikającego ze stref czasowych. Przesunięcie, czyli `offset` (ja wiem, że to nie jest dokładne tłumaczenie tego słowa, ale w praktyce często mówi się na to przesunięcie) liczone jest od punktu `GMT/UTC`.

Oba te skróty mają w praktyce ten sam czas natomiast różnica między nimi (`GMT/UTC`) jest taka, że *Greenwich Mean Time (GMT)* to strefa czasowa, a *Universal Time Coordinated UTC* to standard czasu.

W praktyce, utworzenie obiektu tej klasy wygląda tak: (znowu konstruktor jest prywatny)

```
// OffsetDateTime.now: 2018-04-22T19:50:06.771733800+02:00
System.out.println("OffsetDateTime.now: " + OffsetDateTime.now());
// OffsetDateTime.of: 2020-10-10T12:01:03.000000012Z
System.out.println("OffsetDateTime.of: "
    + OffsetDateTime.of(2020, 10, 10, 12, 1, 3, 12, ZoneOffset.UTC));
// OffsetDateTime.of: 2020-10-10T20:12:02Z
System.out.println("OffsetDateTime.of: "
    + OffsetDateTime.of(LocalDate.of(2020, 10, 10), LocalTime.of(20, 12, 2), ZoneOffset.UTC));
// OffsetDateTime.of: 2020-10-10T20:12:02Z
System.out.println("OffsetDateTime.of: "
    + OffsetDateTime.of(LocalDateTime.of(2020, 10, 10, 20, 12, 2), ZoneOffset.UTC));

// ZoneOffset.of: +02:00
System.out.println("ZoneOffset.of: " + ZoneOffset.of("+02:00"));
// ZoneOffset.ofHours: +02:00
System.out.println("ZoneOffset.ofHours: " + ZoneOffset.ofHours(2));
// OffsetDateTime.of: 2020-10-10T12:01:03.000000012+02:00
System.out.println("OffsetDateTime.of: "
    + OffsetDateTime.of(2020, 10, 10, 12, 1, 3, 12, ZoneOffset.of("+02:00")));
```

Możemy ją utworzyć analogicznie do LocalDateTime, tylko potrzebujemy dodać `ZoneOffset`.

# ZonedDateTime

O ile `OffsetDateTime` wspiera przesunięcie o konkretną ilość godzin względem `UTC`, o tyle do pełnego wspierania stref czasowych potrzebna jest nam klasa `ZonedDateTime`. Jej kolejną cechą jest to, że wspiera też zmianę czasu z letniego na zimowy i odwrotnie.

W praktyce utworzenie obiektu tej klasy wygląda tak (ponownie konstruktor jest prywatny):

```
ZoneOffset utc = ZoneOffset.UTC;
ZoneId usPacific = ZoneId.of("US/Pacific");
ZoneId paris = ZoneId.of("Europe/Paris");

// ZonedDateTime.of: 2020-10-10T12:01:03.000000012Z
System.out.println("ZonedDateTime.of: "
    + ZonedDateTime.of(2020, 10, 10, 12, 1, 3, 12, utc));
// ZonedDateTime.of: 2020-10-10T20:12:02Z
System.out.println("ZonedDateTime.of: "
    + ZonedDateTime.of(LocalDate.of(2020, 10, 10), LocalTime.of(20, 12, 2), utc));
// ZonedDateTime.of: 2020-10-10T20:12:02Z
System.out.println("ZonedDateTime.of: "
    + ZonedDateTime.of(LocalDateTime.of(2020, 10, 10, 20, 12, 2), utc));
// ZonedDateTime.of: 2020-10-10T20:12:02-07:00[US/Pacific]
System.out.println("ZonedDateTime.of: "
    + ZonedDateTime.of(LocalDateTime.of(2020, 10, 10, 20, 12, 2), usPacific));
// ZonedDateTime.of: 2020-10-10T20:12:02+02:00[Europe/Paris]
System.out.println("ZonedDateTime.of: "
    + ZonedDateTime.of(LocalDateTime.of(2020, 10, 10, 20, 12, 2), paris));

// ZonedDateTime.now: 2019-02-04T20:01:50.050569700+02:00[Asia/Singapore]
System.out.println("ZonedDateTime.now: " + ZonedDateTime.now());
```

Możemy ją utworzyć analogicznie do `LocalDateTime`, tylko potrzebujemy dodać konkretną strefę czasową `ZoneId`. Możemy również przekazać `ZoneOffset`.

Aby się dowiedzieć, jakie mamy możliwości co do użycia stref czasowych, możemy to sprawdzić w ten sposób:

```
System.out.println(ZoneId.getAvailableZoneIds());
```

# Instant

Oprócz klas, które zostały wymienione, mamy również klasę, która reprezentuje konkretny punkt na osi czasu, liczony w `UTC`. Tą klasą jest `Instant`.

Jak już pewnie się domyślasz, `Instant` również ma prywatny konstruktor i wywołując konstruktor, obiektu tej klasy utworzyć nie możemy. Z racji, że jest to konkretny punkt w czasie w `UTC`, jeżeli chcemy taki punkt znaleźć, musimy podać datę i godzinę z konkretną strefą czasową. Dopiero wtedy Java jest w stanie w pełni się odnieść do punktu w czasie w konkretnym miejscu na ziemi i przenieść go na `UTC`.

```
ZoneId zone1 = ZoneId.of("Europe/Belgrade");
ZoneId zone2 = ZoneId.of("Asia/Singapore");

LocalDate date1 = LocalDate.of(2020, 11, 3);
LocalTime time1 = LocalTime.of(12, 36);

// ZonedDateTime.of: 2020-11-03T12:36+01:00[Europe/Belgrade]
System.out.println("ZonedDateTime.of: " + ZonedDateTime.of(date1, time1, zone1));
// toInstant: 2020-11-03T11:36:00Z
System.out.println("toInstant: " + ZonedDateTime.of(date1, time1, zone1).toInstant());
// ZonedDateTime.of: 2020-11-03T12:36+08:00[Asia/Singapore]
System.out.println("ZonedDateTime.of: " + ZonedDateTime.of(date1, time1, zone2));
// toInstant: 2020-11-03T04:36:00Z
System.out.println("toInstant: " + ZonedDateTime.of(date1, time1, zone2).toInstant());

// Instant.now: 2020-08-27T18:04:23.636654600Z
System.out.println("Instant.now: " + Instant.now());
```

Z każdą z powyższych klas, spróbuj się pobawić wykorzystując metody `.now()`, żeby zobaczyć, jak się one odnoszą do Twoich ustawień systemu operacyjnego.

# Manipulowanie datami

Jeżeli chcemy zmieniać datę lub czas poprzez dodawanie, lub odejmowanie godzin, lub innych jednostek czasu, wymienione klasy dostarczają nam na to dużo sposobów. Załóżmy, że mamy takie zmienne:

```
LocalDate localDate = LocalDate.of(2020, 10, 1);
LocalTime localTime = LocalTime.of(20, 10);
LocalDateTime localDateTime = LocalDateTime.of(localDate, localTime);
```

Java dostarcza nam dużo metod opisujących, co możemy dodać/odjąć, wystarczy tylko podać ilość:

```
System.out.println("plusHours: " + localDateTime.plusHours(1));
System.out.println("plusMinutes: " + localDateTime.plusMinutes(1));
System.out.println("plusNanos: " + localDateTime.plusNanos(1));
System.out.println("plusSeconds: " + localDateTime.plusSeconds(1));
// wywołania takie można chainować
System.out.println("chain: " + localDateTime.plusHours(1).plusMinutes(2).plusSeconds(1));
```

Drugim sposobem jest wykorzystanie metod `plus()` lub `minus()`, do których przekazujemy ilość, oraz wybieramy jednostkę z enuma `ChronoUnit`.

```
// Dlatego możemy wykonać to sprawdzenie,
// bo przykładowo LocalDate nie wspiera dodawania/odejmowania godzin
System.out.println("localDate.isSupported: " + localDateTime.isSupported(ChronoUnit.DAYS));
System.out.println("localDateTime.plus: " + localDateTime.plus(2, ChronoUnit.DAYS));

System.out.println("localDateTime.isSupported: " + localDateTime.isSupported(ChronoUnit.MONTHS));
System.out.println("localDateTime.plus: " + localDateTime.plus(2, ChronoUnit.MONTHS));

System.out.println("localDateTime.isSupported: " + localDateTime.isSupported(ChronoUnit.YEARS));
System.out.println("localDateTime.plus: " + localDateTime.plus(2, ChronoUnit.YEARS));
```

# Period i Duration

Oprócz wszystkich klas wymienionych wcześniej, Java dostarcza nam też klasy pozwalające operować na przedziałach czasu.

## Period

`Period` jest klasą przeznaczoną do operowania na latach, miesiącach i dniach celem mierzenia odstępu czasu w tych jednostkach.

```
// Period.of: P2Y3M10D
System.out.println("Period.of: " + Period.of(2, 3, 10));
// Period.ofDays: P2D
System.out.println("Period.ofDays: " + Period.ofDays(2));
// Period.ofMonths: P2M
System.out.println("Period.ofMonths: " + Period.ofMonths(2));
// Period.ofWeeks: P14D
System.out.println("Period.ofWeeks: " + Period.ofWeeks(2));
// Period.ofYears: P2Y
System.out.println("Period.ofYears: " + Period.ofYears(2));
```

Przydatnym zastosowaniem jest również określenie przedziału między datami:

```
LocalDate localDate1 = LocalDate.of(2020, 10, 1);
LocalDate localDate2 = LocalDate.of(2020, 12, 1);

// Period: P2M
System.out.println("Period: " + Period.between(localDate1, localDate2));
// Period: P0D
System.out.println("Period: " + Period.between(localDate1, localDate1));
// Period: P-2M
System.out.println("Period: " + Period.between(localDate2, localDate1));
```

Jak widzisz, `Period` może być ujemny ☺.

## Duration

Klasa `Duration` reprezentuje przedział czasu w sekundach lub nanosekundach i jest przeznaczona do

obsługi krótszych ilości czasu niż w przypadku `Period`.

```
// Duration.ofDays: PT24H
System.out.println("Duration.ofDays: " + Duration.ofDays(1));
// Duration.ofHours: PT1H
System.out.println("Duration.ofHours: " + Duration.ofHours(1));
// Duration.ofMinutes: PT1M
System.out.println("Duration.ofMinutes: " + Duration.ofMinutes(1));
// Duration.ofSeconds: PT1S
System.out.println("Duration.ofSeconds: " + Duration.ofSeconds(1));
// Duration.ofMillis: PT0.001S
System.out.println("Duration.ofMillis: " + Duration.ofMillis(1));
// Duration.ofNanos: PT0.000000001S
System.out.println("Duration.ofNanos: " + Duration.ofNanos(1));
```

Tak samo, jak w przypadku `Period`, `Duration` może być używane do określenia przedziału czasu:

```
LocalDate localDate1 = LocalDate.of(2020, 10, 20);
LocalDate localDate2 = LocalDate.of(2020, 11, 20);
LocalTime localTime1 = LocalTime.of(20, 10, 20);
LocalTime localTime2 = LocalTime.of(23, 10, 27);

// UnsupportedTemporalTypeException: Unsupported unit: Seconds
System.out.println("Duration.between: " + Duration.between(localDate1, localDate2));
// DateTimeException: Unable to obtain LocalTime from TemporalAccessor:
// 2020-11-20 of type java.time.LocalDate
System.out.println("Duration.between: " + Duration.between(localTime1, localDate2));
// Duration.between: PT-3H-7S
System.out.println("Duration.between: " + Duration.between(localTime2, localTime1));
// Duration.between: PT3H7S
System.out.println("Duration.between: " + Duration.between(localTime1, localTime2));
```

Jak widać, nie da się określić przedziału między `LocalDate` i `LocalTime`. W sumie, jak się logicznie zastanowić, to ma to sens ☺.

# Formatowanie

Na koniec chciałem poruszyć temat formatowania dat i czasów, czyli przedstawiania ich jako `String`.

Załóżmy, że na potrzeby kolejnych przykładów, stworzymy takie zmienne:

```
LocalDate date = LocalDate.of(2018, Month.OCTOBER, 28);
LocalTime time = LocalTime.of(1, 50);
LocalDateTime dateTime = LocalDateTime.of(date, time);
OffsetDateTime offsetDateTime = OffsetDateTime.of(dateTime, ZoneOffset.ofHours(-4));
ZonedDateTime zonedDateTime = ZonedDateTime.of(dateTime, ZoneId.of("Europe/Paris"));
```

Następnie możemy tak stworzone daty formatować w sposób zgodny ze standardem `ISO`:

```
// ISO_LOCAL_DATE_TIME: 2018-10-28T01:50:00
System.out.println("ISO_LOCAL_DATE_TIME: "
    + DateTimeFormatter.ISO_LOCAL_DATE_TIME.format(dateTime));
// ISO_OFFSET_DATE_TIME: 2018-10-28T01:50:00-04:00
System.out.println("ISO_OFFSET_DATE_TIME: "
    + DateTimeFormatter.ISO_OFFSET_DATE_TIME.format(offsetDateTime));
// ISO_ZONED_DATE_TIME: 2018-10-28T01:50:00+02:00[Europe/Paris]
System.out.println("ISO_ZONED_DATE_TIME: "
    + DateTimeFormatter.ISO_ZONED_DATE_TIME.format(zonedDateTime));
```

Możliwe jest również podanie własnego formatu daty, który będzie określony w Stringu.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd.MM.yyyy HH:mm z");
// formatter.format: 28.10.2018 01:50 UTC-04:00
System.out.println("formatter.format: "
    + formatter.format(ZonedDateTime.of(dateTime, ZoneId.of("UTC-4"))));
```

Możemy również tworzyć daty ze Stringa. Nazywa się to wtedy parsowaniem.

```
LocalDate date = LocalDate.parse("2020-03-15");
LocalDateTime dateTime = LocalDateTime.parse("2020-03-15T11:50:55");
ZonedDateTime zonedDateTime = ZonedDateTime.parse("2020-03-15T10:15:30+01:00[Europe/Paris]");

// 2020-03-15
System.out.println(date);
// 2020-03-15T11:50:55
System.out.println(dateTime);
// 2020-03-15T10:15:30+01:00[Europe/Paris]
System.out.println(zonedDateTime);

String stringDate = "Mon, 05 May 1980";
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("EEE, d MMM yyyy", Locale.ENGLISH);
LocalDate parsed = LocalDate.parse(stringDate, formatter);
// 1980-05-05
System.out.println(parsed);
```

Jeżeli zastanawiasz się, w jaki sposób można w Stringu określić format dat, umieszczam pod spodem tabelę z oznaczeniami poszczególnych znaków:

*Tabela 1. Oznaczenia poszczególnych znaków dla Stringa DateTimeFormatter*

| Znak | Opis | Przykład |
|---|---|---|
| **y** | Rok | (1992, 95) |
| **M** | Miesiąc w roku | (July, Jul, 07) |
| **d** | Dzień w miesiącu | (1-31) |
| **E** | Nazwa dnia w tygodniu | (Friday, Sunday) |
| **a** | am czy pm | (AM, PM) |
| **H** | Godzina w ciągu dnia | (0-23) |
| **h** | Godzina z oznaczeniem am/pm | (1-12) |
| **m** | Minuta w godzinie | (0-60) |
| **s** | Sekunda w minucie | (0-60) |