Zaawansowane programowanie obiektowe i funkcyjne

Wykład 3: Czas

dr inż. Marcin Luckner
mluckner@mini.pw.edu.pl

Wydział Matematyki i Nauk Informacyjnych

Wersja 1.1
5 marca 2021

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Problemy z czasem

- Sekunda, jako jednostka czasu, została ustalona na podstawie obrotu Ziemi wokół własnej osi.
- Jeżeli pełen obrót ziemi (doba) trwa 24 godziny, to można opisać dobę jako $24 \times 60 \times 60 = 86400$ sekund.
- Niestety, ziemia jest kulista, ale *u biegunów – spłaszczona nieco* i nie jest to precyzyjne odwzorowanie rzeczywistości.
- Żeby nadrobić czas wprowadzono *sekundy przestępne*.

Jak komputery radzą sobie z czasem?

- Wprowadzenie sekund przestępnych jest kłopotliwe dla sprzętu komputerowego.
- Wymaga spowalniania lub przyśpieszania czasu.
- Wobec tego komputery nadal zakładają, że każda doba liczy 86400 sekund.
- Sprawdza się to lokalnie, ale doprowadza do rozbieżności pomiędzy maszynami.
- Rozwiązuje się ten problem poprzez synchronizację czasu.

Problemy z Javą

- Java 1.0 stosowana była klasa `Date`.
  - Naiwne podejście do czasu, prawie całkowicie odrzucone w 1.1.
- Java 1.1 stosowana była klasa `Calendar`.
  - Nieintuicyjne w użyciu.
  - Ignoruje niektóre zagadnienia związane z czasem (sekundy przestępne, strefy czasowe i zmiany czasu).
- Java 8.0 wprowadza API `java.time`

Założenia reprezentacji czasu w Javie

- API `java.time` nakłada następujące założenia dla skali czasu.
  - Doba ma liczyć 86 400 sekund.
  - Skala ma *idealnie* odpowiadać oficjalnemu czasowi w południe każdego dnia.
  - Skala ma *dokładnie* odpowiadać oficjalnemu czasowi w jego innych punktach wedle ściśle zdefiniowanych reguł.

Reprezentacja momentu czasu

- Klasa *Instant* reprezentuje określony moment czasu.
- Początek nazwany *epoką* ustalono na na północ 1 stycznia 1970 r. w Greenwich.
  - Ta sama konwencja stosowana jest w systemach UNIX i POSIX.
- Czas może być mierzony do przodu i do tyłu z dokładnością do nanosekund.
- Wartości klasy `Instant` są ograniczone.
  - `Instant.MIN` miliard lat wstecz.
  - `Instant.MAX` do 31 grudnia roku 1 000 000 000.

Pomiar upływu czasu

- Statyczna metoda `now` klasy `Instant` zwraca instancję klasy reprezentującą bieżącą datę i godzinę.
- Upływ czasu między dwiema instancjami można zmierzyć stosując metodę `between`.

```
1  Instant start = Instant.now();
2  runAlgorithm();
3  Instant end = Instant.now();
4  Duration timeElapsed = Duration.between(start, end);
5  long millis = timeElapsed.toMillis();
6  //101 milliseconds
```

- Metoda `toMillis` zwraca wartość w milisekundach. Istnieją inne metody np. `toSeconds`, `toHours` ale należy pamiętać, że zwracają wartość zaokrągloną w dół.

Obliczanie czasu wykonania algorytmu

- Napiszmy procedurę do obliczania czasu działania algorytmu.

```
1  public static long calculateComputationTime(Runnable r){
2    Instant start = Instant.now();
3    r.run();
4    Instant end = Instant.now();
5    Duration timeElapsed = Duration.between(start, end);
6    return timeElapsed.toMillis();
7  }
```

Porównanie czasu wykonania algorytmów

- Porównajmy dwa algorytmy łączące teksty na różne sposoby.

Konkatenacja tekstu

```
1  public static void
       stringConcatenation(){
2    String s ="a";
3    for(int i=0; i<LIMIT; i++) {
4      s = s+"a";
5    //s = new StringBuilder(s)
6    //   .append("a").toString();
7    }
8  }
```

Budowanie tekstu

```
1  public static void
       stringBuilding(){
2    String s ="a";
3    StringBuilder sB = new
         StringBuilder(s);
4    for(int i=0; i<LIMIT; i++) {
5      sB.append("a");
6    }
7    s = sB.toString();
8  }
```

- Porównajmy czas wykonania dla 10 tysięcy tekstów.

```
Concatenation time 259ms
Building time 3ms
```

Operacje na `Duration`

- Typ `Duration` reprezentuje odcinek czasu.
- Możemy na nim wykonywać operacje arytmetyczne i sprawdzać jego wartość.

Czy algorytm jest dziesięciokrotnie szybszy?

```
1  buildingTime.multipliedBy(10).minus(concatenationTime).isPositive()
2  //false
```

Daty lokalne

- Daty lokalne są datami działającymi z pominięciem strefy czasowej.
- Reprezentowana jest przez typ `LocalDate`.
- Tworzenie dat odbywa się przez metody statyczne.

```
1  LocalDate today = LocalDate.now(); // today date
2  LocalDate firstSpamEmailDate = LocalDate.of(1978, 5, 1);
3  firstSpamEmailDate = LocalDate.of(1978, Month.MAY, 1);
```

Operacje na datach lokalnych

- Do daty możemy dodawać jednostki czasu.

```
1  LocalDate programmersDay = LocalDate.of(2019, 1,
       1).plusDays(255);
2  //2019-09-13
```

- Możemy odczytywać składowe daty.

```
1  programmersDay.getDayOfWeek()
2  //FRIDAY
```

- Możemy obliczyć okres pomiędzy dwoma datami.
- Okres zwracany jest jako instancja klasy `Period`.

```
1  LocalDate.now().until(programmersDay)
2  //P9M29D
```

Warunkowe wyszukiwanie daty

- Możemy wprowadzić dodatkowe warunki przy tworzeniu daty.
- Klasa `TemporalAdjusters` udostępnia metody wyszukujące np. następny/poprzedni dzień tygodnia.
- Warunki są przekazywane poprzez metodę `with` klasy `LocalDate`.

Pierwszy poniedziałek nowego roku

```
1  LocalDate firstMonday = LocalDate.of(2019, 1, 1).with(
2  TemporalAdjusters.nextOrSame(DayOfWeek.MONDAY));
3  //2019-01-07
```

Czas lokalny

- Czas lokalny jest czasem z pominięciem strefy czasowej.
- Reprezentowany jest przez typ `LocalTime`.
- Konstrukcja odbywa się przez metody statyczne.

```
1  LocalTime rightNow = LocalTime.now();
2  LocalTime bedtime = LocalTime.of(22, 30); // or
       LocalTime.of(22, 30, 0)
```

- Podobnie jak dla `LocalDate` istnieją operatory modyfikujące czas lokalny

```
1  LocalTime wakeup = bedtime.plusHours(8);
```

- Istnieje klasa `LocalDateTime` łącząca datę i czas.
- Może ona uwzględniać strefę czasową, ale nie wspiera dokonywania obliczeń międzystrefowych.

Czas strefowy

- Czas strefowy obejmuje dwa zagadnienia.
  - Zmianę czasu pomiędzy strefami czasowymi.
  - Zmianę czasu z letniego na zimowy i odwrotnie.
- Baza danych dotycząca stref czasowych jest utrzymywana przez agencję Internet Assigned Numbers Authority (IANA)[1].
  - Aktualnie w bazie znajduje się 599 stref.
  - Ostatnie zmiany w bazie dotyczą głównie zmian sezonowych.
- Język Java korzysta z bazy IANA.
- W związku z powyższym ewentualne zmiany w prawodawstwie UE zostaną automatycznie uwzględnione.

[1] `http://www.iana.org/time-zones`

Strefy czasowe w Javie

- Java może pobrać dane dotyczące strefy czasowej znając jej identyfikator.
- Przykładowo, identyfikator dla Polski to *Europe/Warsaw*.
- Spis wszystkich stref można uzyskać wywołując

```
1  ZoneId.getAvailableZoneIds()
```

- Dysponując identyfikatorem strefy czasowej można pobrać obiekt `ZoneId`

```
1  ZoneId.of("Europe/Warsaw")
```

Wyliczenie czasu dla strefy czasowej

- Klasa `ZonedDateTime` pozwala wyliczyć czas dla danej strefy czasowej.
- Można utworzyć zlokalizowany czas metodami `of` lub `now`.

```
1  ZonedDateTime timeNewYork = ZonedDateTime.of(2018,
       11,16,17,15,0,0,ZoneId.of("America/New_York"));
```

- Klasa `ZonedDateTime` dysponuje szeregiem metod podobnych jak w klasie `LocalDateTime`.

Lot międzykontynentalny

- Użyjemy `ZonedDateTime` do obliczenia czasu przybycia w locie międzykontynentalnym.
- Wykorzystamy metodę `withZoneSameInstant` zwraca aktualny czas w danej strefie czasowej.

```
1  int flightTime = 8;
2
3  ZonedDateTime departureTime  =
4    ZonedDateTime.of(2018,
         11,16,16,55,0,0,ZoneId.of("Europe/Warsaw"));
5  //2018-11-16T16:55+01:00[Europe/Warsaw]
6  ZonedDateTime arrivalTime =
7    departureTime.plusHours(flightTime)
8    .withZoneSameInstant(ZoneId.of("America/New_York"));
9  //2018-11-16T18:55-05:00[America/New_York]
10 departureTime  = ZonedDateTime.of(2018,
       11,25,22,40,0,0,ZoneId.of("America/New_York"));
11 //2018-11-25T22:40-05:00[America/New_York]
12 arrivalTime = departureTime.plusHours(flightTime)
13   .withZoneSameInstant(ZoneId.of("Europe/Warsaw"));
14 //2018-11-26T12:40+01:00[Europe/Warsaw]
```

Zmiana czasu

- Utwórzmy ZonedDateTime dla 2019-03-31 2:30.

```
ZonedDateTime timestamp = ZonedDateTime.of(
LocalDate.of(2019, 3, 31),
LocalTime.of(2, 30),
ZoneId.of("Europe/Warsaw"));
```

- Jaka data powstanie?

```
2019-03-31T03:30+02:00[Europe/Warsaw]
```

- Godzina, którą chcieliśmy utworzyć nie istnieje ze względu na zmianę czasu.

Upływ czasu a jego zmiana

- Załóżmy, że mieliśmy spotkanie robocze 2019-03-30 o 11:30.

```
1  ZonedDateTime meeting = ZonedDateTime.of(
2                  LocalDate.of(2019, 3, 30),
3                  LocalTime.of(11, 30),
4                  ZoneId.of("Europe/Warsaw"));
```

- Chcemy się spotkać ponownie za tydzień o tej samej porze.

```
1  ZonedDateTime nextMeeting =
       meeting.plus(Duration.ofDays(7));
```

- O której będzie następne spotkanie?

```
1  2019-04-06T12:30+02:00[Europe/Warsaw]
```

- Nie uwzględniliśmy zmiany czasu. Zamiast instancji `Duration` powinniśmy użyć `Period`.

```
1  nextMeeting = meeting.plus(Period.ofDays(7));
2  //2019-04-06T11:30+02:00[Europe/Warsaw]
```

Formatowanie

- Możemy ustalić w jaki sposób będą wyświetlane informacje o dacie i czasie.
- Klasa `DateTimeFormatter` udostępnia mechanizmy oferujące:
  - predefiniowane formatowanie.
  - lokalizację.
  - niestandardowe formatowanie.

Predefiniowane formatowanie

- Znając nazwę predefiniowanego formatowania możemy uzyskać datę w określonym formacie.

```
1  ZonedDateTime programmersDayCellebration =
       ZonedDateTime.of(2019, 1, 1,19,0,0,0,
       ZoneId.of("Europe/Warsaw")).plusDays(255);
2
3  DateTimeFormatter.RFC_1123_DATE_TIME
4    .format(programmersDayCellebration);
5    //Fri, 13 Sep 2019 19:00:00 +0200
```

Przykładowe formatowanie predefiniowane

formatowanie	wynik
RFC_1123_DATE_TIME	Fri, 13 Sep 2019 19:00:00 +0200
ISO_ORDINAL_DATE	2019-256+02:00
ISO_ZONED_DATE_TIME	2019-09-13T19:00:00+02:00[Europe/Warsaw]
ISO_LOCAL_DATE	2019-09-13
ISO_WEEK_DATE	2019-W37-5+02:00

- Formatowanie predefiniowane służy głównie do komputerowego przetwarzania znaczników czasu.

Formatowanie lokalizowane

- Formatowanie przyjazne użytkownikowi uzyskamy tworząc `DateTimeFormatter`.

```
1  DateTimeFormatter formatter =
       DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG);
2  String formatted =
       formatter.format(programmersDayCellebration);
3  //13 wrzesnia 2019 19:00:00 CEST
```

- Istnieją cztery formaty lokalizowanego zapisu daty.

Formatowanie lokalizowane

formatowanie	wynik
FULL	piątek, 13 września 2019 19:00:00 Czas środkowoeuropejski letni
LONG	13 września 2019 19:00:00 CEST
MEDIUM	13.09.2019, 19:00:00
SHORT	13.09.2019, 19:00

- Domyślną lokalizację można zastąpić inną.

```
1  DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG)
2    .withLocale(Locale.ENGLISH).format(programmersDayCellebration);
3  //September 13, 2019 at 7:00:00 PM CEST
```

Niestandardowe formatowanie

- Możemy utworzyć własne formatowanie `DateTimeFormatter`.

```
1  DateTimeFormatter formatter =
       DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");
2  formatted =
       formatter.format(programmersDayCellebration);
3  //pt. 2019-09-13 19:00
```

- Znaczenie symboli we wzorcu widnieje w dokumentacji Javy[2].

Przykładowe symbole formatowania

pole	format i wynik
YEAR_OF_ERA	yy: 69, yyyy: 1969
MONTH_OF_YEAR	M: 7, MM: 07, MMM: lip, MMMM: lipca, MMMMM: l
DAY_OF_MONTH	d: 6, dd: 06
DAY_OF_WEEK	e: 3, E: Śr, EEEE: środa, EEEEE: Ś
HOUR_OF_DAY	H: 9, HH: 09

[2] https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

Parsowanie

- Wzorce formatowania daty mogą być też stosowane do wczytywania i parsowania danych.
- Możemy użyć standardowych mechanizmów formatujących.

```
1  LocalDate employeeBirthday =
       LocalDate.parse("1969-06-14");
2  //1969-06-14
```

- Możemy też użyć niestandardowego formatowania.

```
1  ZonedDateTime systemRebootTime =
2    ZonedDateTime.parse("2018-07-16 03:32:00-0400",
3    DateTimeFormatter.ofPattern("yyyy-MM-dd
         HH:mm:ssxx"));
4  //2018-07-16T03:32-04:00
```

Współdziałanie ze starym kodem

- Mechanizmy wprowadzone w Java 8 są znacznie lepsze od poprzednich rozwiązań, które nie powinny być używane.
- Jednakże w przypadku współpracy ze starym kodem możemy spotkać się ze starymi formatami.
- Java 8 oferuje ich konwersję do nowych formatów.
  - Klasa `java.util.Date` zyskała metody `toInstant` i statyczną metodę `from` do konwersji `Date` na `Instant` i odwrotnie.
  - Klasa `Calendar` zyskała metody `toZonedDateTime` i statyczną metodę `from` do konwersji `GregorianCalendar` na `ZonedDateTime` i odwrotnie.

Bibliografia