Java 8

(programowanie funkcyjne)

Hello!

Michał Nowakowski

lead software engineer @EPAM

michal@nowakowski.me.uk

Java 8

- Interfejsy funkcyjne i wyrażenia lambda
- Strumienie
- Optional
- Nowe Date/Time API

Interfejs funkcyjny
Wyrażenia lambda

Java 8
Interfejs funkcyjny

- Interfejs, który posiada tylko jedną metodę abstrakcyjną, eksponuje pojedynczą funkcjonalność
- Pozwala na użycie wyrażeń lambda zamiast jawnych implementacji interfejsu
- Może zawierać dowolną liczbę metod static i default
- Opcjonalnie oznaczony adnotacją `@FunctionalInterface`
- Może deklarować abstrakcyjne metody z `java.lang.Object` (`toString`, `equals`)

Java 8
Interfejs funkcyjny

```java
@FunctionalInterface
public interface Task {
    void doWork();
}
```

```java
public static void main(String[] args) {
    carryOutWork(new Task() {
        @Override
        public void doWork() {
            System.out.println("Hello");
        }
    });
}

public static void carryOutWork(Task task) {
    task.doWork();
}
```

Java 8
Interfejs funkcyjny

- Więcej niż jedna metoda abstrakcyjna?

```
@FunctionalInterface
public interface Task {
    void doWork();
    void sayHi();
}
```

- Error:(2, 1) java: Unexpected @FunctionalInterface annotation Task is not a functional interface, **multiple non-overriding abstract methods found** in interface Task

Java 8

Interfejs funkcyjny - lambda

```
@FunctionalInterface
public interface Task {
    void doWork();
}
```

```
public static void main(String[] args) {
    carryOutWork(
        () -> System.out.println("Hello")
    );
}

public static void carryOutWork(Task task) {
    task.doWork();
}
```

Java 8
Wyrażenia lambda

- Umożliwiają programowanie funkcjonalne, skupienie uwagi bardziej na tym co chce się zrobić niż zajmowanie się obiektami
- Anonimowa implementacja interfejsu funkcyjnego
- Blok kodu, który może zostać przekazany jako parametr i wykonany w dowolnym momencie
- Podobnie jak metoda, składa się z listy parametrów (o ile występują) i ciała, ale nie ma nazwy (!!!)

```
(type1 arg1, type2 arg2) -> {body}
```

Java 8

Comparator – Java <= 7

```
List<String> names = Arrays.asList("Kasia",
    "Ania", "Zosia", "Bartek");

Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.compareTo(o2);
    }
});
```

Java 8

Comparator – Java 8+

```
List<String> names = Arrays.asList("Kasia",
    "Ania", "Zosia", "Bartek");

Collections.sort(names, (o1, o2) -> o1.compareTo(o2));

Collections.sort(names,
    (String o1, String o2) -> o1.compareTo(o2));

Collections.sort(names, (o1, o2) -> {
    return o1.compareTo(o2);
});
```

Java 8
Wyrażenia lambda

```
(type1 arg1, type2 arg2) -> {body}
```

- Typy parametrów nie są obowiązkowe

```
(arg1, arg2) -> {body}
```

- Nawiasy wymagane, gdy brak parametrów `()` lub więcej niż jeden parametr `(arg1, arg2)`

- Nawiasy klamrowe wymagane, jeśli ciało metody ma więcej niż jedną linijkę (pamiętaj o `return` i o **średniku**)

Java 8

Wyrażenia lambda - przykłady

```
() -> 1

() -> {
  System.out.println(“hi”);
  return 0;
}

x -> x*x

(x, y) -> x*y

(int x, int y) -> { return x*y; }
```

Java 8
Ćwiczenie 1

- Wskaż błędy w podanych przykładach:

```
print(a, b -> a.startsWith(b));

print(a -> { a.startsWith(“foo”); });

x -> return x*x;

print(a -> { return a.startsWith(“foo”) });

(int x, String y) -> x*y
```

Java 8
Ćwiczenie 2

- Zaimplementuj interfejs funkcyjny `Runnable` w dwóch wersjach – z wykorzystaniem Javy 8 i „po staremu”

Java 8
Ćwiczenie 3 (I)

- Utwórz interfejs `MathOperation` z jedną metodą `calculate` do obliczeń matematycznych, parametr wejściowy – lista elementów typu `Integer`, metoda powinna zwracać wartość typu `Integer`

- Przygotuj dwie klasy, które implementują ten interfejs – `MaxOperation` i `MinOperation`, zwracające odpowiednio największy i najmniejszy element listy

Java 8
Ćwiczenie 3 (II)

- Utwórz klasę z metodą `main` i przygotuj metodę `getResult`, która przyjmie listę elementów typu `Integer` i instancję `MathOperation`. Metoda nie powinna nic zwracać (`void`), powinna wyświetlić wynik podanej operacji

- Przygotuj dane testowe i wywołaj metodę używając obiektów utworzonych klas `MinOperation` i `MaxOperation`

- Użyj wyrażeń lambda tak, aby nie trzeba było tworzyć instancji obiektów `MinOperation` i `MaxOperation`

Java 8

Interfejsy funkcyjne

- `Predicate<T>`
- `Consumer<T>`
- `Function<T, R>`
- `Supplier<T>`
- `UnaryOperator<T>`
- `BinaryOperator<T>`
- `BiPredicate<L, R>`
- `BiConsumer<T, U>`
- Więcej: https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Java 8
Predicate<T>

- Wyrażenie logiczne obiektu typu T

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

```
Predicate<String> nonEmptyString = (String s) -> !s.isEmpty();
List<String> noEmptyStrings = filter(names, nonEmptyString);
```

Java 8
Consumer<T>

- Konsument obiektu typu T (nie zwraca wartości)

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

```
Consumer<String> consumer = s -> System.out.println("name = " + s);
names.forEach(consumer);
```

Java 8
Function<T, R>

- Funkcja pobierająca obiekt typu T i zwracająca obiekt typu R (mapowanie danych do innego typu)

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

```
Function<String, Integer> f = s -> s.length();
int length = f.apply("string");
```

Java 8
Supplier<T>

- Tworzenie nowych obiektów, dostawca obiektów

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

```
Supplier<Integer> random = () -> new Random().nextInt();
int randomNumber = random.get();
```

Java 8
Interfejsy funkcyjne

- `UnaryOperator<T>` - funkcja pobierająca typ T i zwracająca typ T, `Function<T, T>`
- `BinaryOperator<T>` - funkcja pobierająca dwa parametry typu T i zwracająca typ T, `BiFunction<T, T, T>`
- `BiPredicate<L, R>` - wyrażenie logiczne obiektów różnych typów, dwa parametry typu L i R
- `BiConsumer<T, U>` - konsument dwóch obiektów, typu T i U

Java 8
Ćwiczenie 4

- Usuń z listy Stringów wszystkie puste elementy z wykorzystaniem wyrażenia lambda
- Podpowiedź: `Collection.removeIf()`

Java 8
Ćwiczenie 5

- Przy użyciu wyrażeń lambda wypisz wszystkie elementy listy Stringów wielkimi literami (ang. *uppercase*)

Java 8
Ćwiczenie 6

- Napisz funkcję, która dla danej listy zwróci listę długości jej elementów
- Np. dla [Asia, Basia] zwróci [4, 5]

Java 8
Ćwiczenie 7

- Zastąp interfejs `MathOperation` interfejsem `Function<T, R>`
- Do wyświetlania wyników użyj interfejsu funkcyjnego `Consumer<T>` (jako kolejny parametr metody `getResult`)

Referencje do metod

Java 8
Referencje do metod

- Zamiast pisać wyrażenie lambda, możemy przekazać istniejące metody, które „implementują” interfejs funkcyjny (parametry i zwracany typ muszą być takie same)

```
List<String> result = new ArrayList<>();
list.forEach(result::add);
```

- Metody statyczne: `NazwaKlasy::nazwaStatycznejMetody`
- Metody instancji klas: `nazwaObiektu::nazwaMetody`
- Konstruktor: `NazwaKlasy::new`

Java 8
Ćwiczenie 8

- Wypisz na ekran elementy listy (przy użyciu referencji do metody)

Strumienie

Java 8
Strumienie

- Rozszerzenie Collections API
- Klasy z pakietu `java.util.stream`
- Strumień to sekwencja obiektów danego typu, na których mogą zostać wykonane operacje takie jak filtrowanie, mapowanie, ograniczanie, zmniejszanie, znajdywanie, itp.
- Odejście od pętli i ciągłego sprawdzania warunków
- Łatwe zrównoleglenie operacji

Java 8
Strumienie

- Strumienie są tworzone na podstawie kolekcji danych, np. `List, Set, Arrays, IO`
- Strumienie nie przechowują elementów
- Źródłowa kolekcja nie jest modyfikowana
- Leniwe operacje (kod wykonany wyłącznie wtedy, kiedy zachodzi potrzeba)
- Strumienie mogą być nieograniczone

Java 8
Tworzenie strumieni

- Metody `stream()` i `parallelStream()` klas reprezentujących kolekcje

```
Stream<String> namesStream = names.stream();
```

- `Arrays.stream(Object[])`

```
Stream<String> namesStream = Arrays.stream(
    new String[] {"Kasia", "Asia"});
```

- `Stream.of(Object[])`

```
Stream<String> namesStream = Stream.of(
    new String[] {"Kasia", "Asia"});
```

Java 8

Tworzenie strumieni

- `IntStream.range(int, int)`

```
IntStream intStream = IntStream.range(0, 100);
```

- `Random.ints()`

```
IntStream intStream = new Random().ints();
```

Java 8
Operacje na strumieniach

- Operacje pośrednie (ang. *intermediate*)
- Operacje końcowe (ang. *terminal*)

Java 8
Operacje pośrednie

- Wynik to nowy strumień
- Pozwalają połączyć w łańcuch kilka metod (operacji)
- Są wyliczane leniwie
- `filter, map, flatMap, peek, distinct, sorted, limit`

Java 8

Operacje końcowe

- Zwracają określony typ danych
- Powodują „uruchomienie” operacji na strumieniach – kończą sekwencję operacji na strumieniach
- `forEach, toArray, reduce, collect, min, max, count, anyMatch, allMatch, noneMatch, findFirst, findAny`

Java 8
collect(Collector)

- Grupuje wszystkie elementy pozostające w strumieniu i zwraca je w postaci definiowanej przez podany Collector, np. Listę lub Set

```
Set<String> uniqueNames = names.stream()
    .distinct()
    .collect(Collectors.toSet());
```

- `toSet()`, `toList()`, `toCollection()`, `joining()`, `groupingBy()`, `partitioningBy()`, i wiele innych

Java 8
distinct()

- Zwraca strumień danych które będą unikalne

```
Set<String> uniqueNames = names.stream()
    .distinct()
    .collect(Collectors.toSet());
```

Java 8
sorted(Comparator)

- Przekształca strumień do postaci posortowanej

```
List<String> names2 = names.stream()
    .sorted((s1, s2) -> s1.length() - s2.length())
    .collect(Collectors.toList());
```

Java 8
Ćwiczenie 9

- Zdefiniuj klasę `Dish` z polami: `name, vegetarian, calories`
- Napisz funkcję, która zwróci dania posortowane od najmniej kalorycznego do najbardziej kalorycznego

Java 8
peek(Consumer)

- Wykonuje operację na elemencie bez przekształcania go

```
List<String> names2 = names.stream()
    .peek(System.out::println)
    .collect(Collectors.toList());
System.out.println(names2);
```

Java 8
reduce()

- `reduce(T identity, BinaryOperator<T> accumulator)`
- `identity` – wartość początkowa / domyślna
- `accumulator` – funkcja akumulująca wynik

- Redukcja elementów strumienia przy użycia podanej funkcji

```
String csv = names.stream()
    .reduce("", (s1, s2) -> s1 + s2 + ";");
```

Java 8
map(Function)

- Zwraca strumień z przekształconymi danymi przy pomocy funkcji

```
List<Integer> lengths = names.stream()
    .map(String::length)
    //.map(s -> s.length())
    .collect(Collectors.toList());
```

Java 8
Ćwiczenie 10

- Napisz funkcję, która zwróci listę nazw wszystkich dań

Java 8
Ćwiczenie 11

- Napisz funkcję, która zsumuje liczbę kalorii wszystkich dań

Java 8
limit(int)

- Ogranicza strumień do podanego rozmiaru

```
List<String> names2 = names.stream()
    .sorted((s1, s2) -> s1.length() - s2.length())
    .limit(2)
    .collect(Collectors.toList());
```

Java 8
filter(Predicate)

- Zwraca strumień danych dla których warunek będzie spełniony

```
List<String> longNames = names.stream()
    .filter(s -> s.length() > 10)
    .collect(Collectors.toList());
```

Java 8
flatMap(Function)

- Podobnie jak `map` przekształca dane za pomocą funkcji, z tym, że funkcja ta musi zwracać strumień (nastąpi spłaszczenie, połączenie różnych zestawów danych w jeden)

```
List<Integer> ints = Stream.of(asList(1, 2, 3), asList(9, 8, 7))
    //.flatMap(List::stream)
    .flatMap(list -> list.stream())
    .collect(Collectors.toList());
```

Java 8
forEach(Consumer)

- Wykonuje akcję z każdym elementem strumienia, zamykając go jednocześnie

```
names.stream()
     .sorted((s1, s2) -> s1.length() - s2.length())
     .limit(2)
     .forEach(System.out::println);
```

Java 8
count()

- Zwraca liczbę elementów w strumieniu

```
long longNames = names.stream()
    .filter(s -> s.length() > 10)
    .count();
```

Java 8
findAny()

- Zwraca dowolny element strumienia

```
Optional<String> longName = names.stream()
    .filter(s -> s.length() > 10)
    .findAny();
```

Java 8
anyMatch(Predicate)

- Testuje czy w strumieniu chociaż jeden obiekt spełnia podany warunek

```
boolean containsLongName = names.stream()
    .anyMatch(s -> s.length() > 10);
```

Java 8
Ćwiczenie 12

- Napisz funkcję, która zwróci tylko dania wegetariańskie

Java 8
Ćwiczenie 13

- Napisz funkcję, która zwróci 3 najbardziej kaloryczne dania

Java 8
Ćwiczenie 14

- Napisz funkcję, która zwróci dania, gdzie liczba kalorii > 500, posortowane

Java 8
Ćwiczenie 15

- Dane są słowa: „hello”, „academy”, „java”, „junior”
- Stwórz listę liter występujących w tych słowach, bez powtórzeń
- Zwróć liczbę tych liter

Optional

Java 8
Optional

- Kontener na wartości, które mogą być `null`
- Pomaga uniknąć błędów typu `NullPointerException`
- `java.util.Optional`
- Bazuje na podobnym mechanizmie w Haskellu i Scali
- Klasa używana, gdy zwracany wynik ma być inny niż `null`, ale czasami wynik ma nie zwracać niczego

Java 8
Optional

- Posiada różne metody do obsługi wartości, gdy jest ona „dostępna” lub „niedostępna”
- Odejście od sprawdzania warunku czy dana wartość jest różna od `null`
- Podobieństwo do `Optional` z biblioteki Guava

Java 8

Optional – przykład użycia

```
public static void main(String[] args) {
    String str1 = "isa";
    Optional<String> optional1 = Optional.of(str1);
    System.out.println(optional1.isPresent()); // true
    System.out.println(optional1.get()); // "isa"
    System.out.println(optional1.orElse("empty")); // "isa"

    String str2 = null;
    Optional<String> optional2 = Optional.ofNullable(str2);
    System.out.println(optional2.isPresent()); // false
    System.out.println(optional2.orElse("empty")); // "empty"
}
```

Java 8

Optional - tworzenie

- `Optional.of(T)` – kontener na wartość typu T (nie może być `null`, w przypadku `null` dostaniemy `NullPointerException`)

```
Optional<String> userOpt = Optional.of(findByName("admin"));
```

Java 8
Optional - tworzenie

- `Optional.ofNullable(T)` – kontener na wartość typu T, może być `null`

```
Optional<String> userOpt = Optional.ofNullable(findByName("admin"));
```

- `Optional.empty()` – pusty kontener

```
Optional<String> userOpt = Optional.empty();
```

Java 8
Optional - użycie

- `Optional.get()` – pobranie wartości lub `NoSuchElementException` jeśli wartość jest `null`

```
String user = userOpt.get();
```

- `Optional.orElse(T other)` – zwraca wartość lub `other`

```
String user = userOpt.orElse("unknown");
```

- `Optional.isPresent()` – zwróci `true` jeśli Optional zawiera wartość

```
boolean userFound = userOpt.isPresent();
```

Java 8
Optional - użycie

- `Optional.orElseThrow(Supplier)` – zwraca wartość lub rzuca wyjątek, jeśli wartość `null`

```
String user = userOpt.orElseThrow(IllegalArgumentException::new);
String user = userOpt.orElseThrow(
    () -> new IllegalArgumentException("User not found"));
```

- `Optional.ifPresent(Consumer)` – przekazuje wartość do podanego konsumenta, jeśli `null` – nie robi nic

```
userOpt.ifPresent(user -> processUser(user));
```

Java 8
Ćwiczenie 16

- git clone https://github.com/infoshareacademy/jjdd6-materialy-java8.git
- Sprawić, aby test `UserServiceTest` był pomyślny

Nowe Date/Time API

Java 8

Problemy z Date API

- Istniejące Date API było kłopotliwe w użyciu, niejednokrotnie dawało błędne rezultaty
- Klasa `Date` nie reprezentuje daty, ale punkt w czasie bez odniesienia do kalendarza
- `Date.toString()` wyświetla tekstową reprezentację daty w strefie czasowej właściwej dla systemu (nie aplikacji)
- Pierwszy miesiąc ma index 0
- Nie jest thread-safe

Java 8
Nowe Date/Time API

- Pakiet `java.time`
- Wzorowane na popularnej bibliotece Joda-Time
- Zawiera metody, które ułatwiają manipulowanie datą i czasem
- Łatwiejsze operowanie strefami czasowymi
- Klasy są `immutable – final`, nie posiadają setterów

Java 8
LocalDate

- `LocalDate` – dzień, bez godziny i strefy czasowej, np. 2018-01-02

```
public static void main(String[] args) {
    LocalDate today = LocalDate.now(); // 2018-01-02
    LocalDate tomorrow = today.plusDays(1); // 2018-01-03
    LocalDate twoYearsAgo = today.minusYears(2); // 2016-01-02
    LocalDate threeMonthsLater = today.plus(3, ChronoUnit.MONTHS); // 2018-04-02

    LocalDate definedDate = LocalDate.of(2018, Month.MAY, 1); // 2018-05-01
    Month month = definedDate.getMonth(); // MAY
    int year = definedDate.getYear(); // 2018

    DateTimeFormatter dtf = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
    String formattedDate = definedDate.format(dtf); // 5/1/18
    LocalDate parsedDate = LocalDate.parse("5/1/18", dtf); // 2018-05-01
}
```

Java 8
Ćwiczenie 17

- Stwórz datę (`LocalDate`) reprezentującą 14.02.2018
- Wypisz dzień, miesiąc, rok
- Wypisz datę używając `toString()`
- Wczytaj datę ze `Stringa (LocalDate.parse("yyyy-MM-dd"))`

Java 8
LocalTime

- `LocalTime` – tylko czas, bez informacji o strefie czasowej, np. 12:59:99

```
public static void main(String[] args) {
    LocalTime now = LocalTime.now(); // 12:11:34.123
    LocalTime oneHourLater = now.plusHours(1); // 13:11:34.123
    LocalTime tenMinutesAgo = now.minusMinutes(10); // 12:01:34.123
    LocalTime fiveSecondsLater = now.plus(5, ChronoUnit.SECONDS); // 12:11:39.123

    LocalTime definedTime = LocalTime.of(12, 45, 10); // 12:45:10
    int hour = definedTime.getHour(); // 12
    int seconds = definedTime.getSecond(); // 10
    LocalTime newTime = definedTime.withHour(11); // 11:45:10

    DateTimeFormatter dtf = DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
    String formattedTime = definedTime.format(dtf); // 12:45 PM
    LocalTime thatTime = LocalTime.parse("06:00 PM", dtf); // 18:00
}
```

Java 8
Ćwiczenie 18

- Stwórz obiekt `LocalTime` reprezentujący 07:15:00
- Wypisz godzinę, minutę i sekundę
- Wypisz czas używając `toString()`
- Wczytaj czas ze `Stringa`

Java 8
LocalDateTime

- `LocalDateTime` – dzień i czas, bez informacji o strefie czasowej

```
public static void main(String[] args) {
    LocalDateTime now = LocalDateTime.now(); // 2018-01-16T21:32:44.878
    LocalDateTime oneHourLater = now.plusHours(1); // 2018-01-16T22:32:44.878
    LocalDateTime twoYearsAgo = now.minusYears(2); // 2016-01-16T21:32:44.878
    LocalDateTime fiveMonthsLater = now.plus(5, ChronoUnit.MONTHS); // 2018-06-16T21:32:44.878

    LocalDateTime definedDateTime = LocalDateTime
        .of(2018, Month.MARCH, 1, 12, 45, 59); // 2018-03-01T12:45:59
    int hour = definedDateTime.getHour(); // 12
    Month month = definedDateTime.getMonth(); // MARCH
    LocalDateTime newDateTime = definedDateTime.withDayOfYear(1); // 2018-01-01T12:45:59

    DateTimeFormatter myFormatter = DateTimeFormatter.ofPattern("MM dd, yyyy - HH:mm");
    String formattedDateTime = definedDateTime.format(myFormatter); // 03 01, 2018 - 12:45
    LocalDateTime thatDateTime = LocalDateTime
        .parse("01 05, 2018 - 18:29", myFormatter); // 2018-01-05T18:29
}
```

Java 8
Ćwiczenie 19

- Stwórz datę z godziną (`LocalDateTime`) – np. 19.01.2018 08:00:00
- Pobierz osobno `LocalDate` i `LocalTime`
- Wypisz datę używając `toString()`

Java 8
Clock

- `Clock` – dostęp do *aktualnej* daty i czasu
- Uwzględnia strefe czasową
- Stosowany zamiast System.currentTimeMillis()

Java 8
Clock - przykład

```
public static void main(String[] args) {
    Clock clock = Clock.systemDefaultZone();
    long millis = clock.millis(); // 1516177997251

    ZoneId zone1 = clock.getZone(); // Europe/Warsaw
    ZoneId zone2 = ZoneId.of("Asia/Tokyo"); // Asia/Tokyo

    Instant instant = clock.instant();
    ZoneOffset zoneOffset = zone1.getRules().getStandardOffset(instant);
    System.out.println(zoneOffset); // +01:00

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

Java 8
ZonedDateTime

- `ZonedDateTime` – dzień i czas w danej strefie czasowej (`ZoneId`)

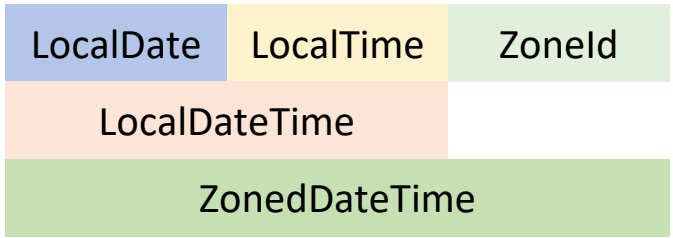
```
public static void main(String[] args) {
    ZoneId myZone = ZoneId.systemDefault(); // Europe/Warsaw
    // 2018-01-17T09:37:32.139+01:00[Europe/Warsaw]
    ZonedDateTime now = ZonedDateTime.now(myZone);
    // 2018-01-17T10:37:32.139+01:00[Europe/Warsaw]
    ZonedDateTime oneHourLater = now.plusHours(1);

    ZoneId zone2 = ZoneId.of("Asia/Tokyo"); // Asia/Tokyo
    // 2018-01-01T10:15:59+09:00[Asia/Tokyo]
    ZonedDateTime definedDateTime =
        ZonedDateTime.of(2018, 1, 1, 10, 15, 59, 0, zone2);
    int hour = definedDateTime.getHour(); // 10
    Month month = definedDateTime.getMonth(); // JANUARY
}
```

Java 8
Ćwiczenie 20

- Utwórz `LocalDateTime` i podaj jego wartość w innej, dowolnie wybranej strefie czasowej

- `LocalDateTime.atZone(ZoneId)`
- `ZoneId.of(String)`

```
1999-02-21T22:00+09:00[Asia/Tokyo]
```



Java 8
ChronoUnit

- Enum – `java.time.temporal.ChronoUnit`
- Używany do reprezentacji dni, miesięcy, itp.

```
public static void main(String[] args) {
    LocalDateTime now = LocalDateTime.now();
    System.out.println(now.plus(60, ChronoUnit.SECONDS)); // + 60 sekund
    System.out.println(now.minus(12, ChronoUnit.MONTHS)); // - 12 miesięcy
    System.out.println(now.plus(2, ChronoUnit.DECADES)); // + 20 lat
    System.out.println(now.plus(1, ChronoUnit.CENTURIES)); // + 100 lat
}
```

Java 8
Instant

- `Instant` – punkt w czasie

```
public static void main(String[] args) {
    Instant now = Instant.now(); // 2018-01-17T08:46:35.019Z

    ZonedDateTime nowDateTime = ZonedDateTime.now();
    Instant now2 = nowDateTime.toInstant(); // 2018-01-17T08:46:35.171Z
}
```

Java 8
Period

- `Period` – przedział czasowy (**dni, miesiące, lata**)

```
public static void main(String[] args) {
    LocalDate today = LocalDate.now(); // 2018-01-17
    Period period1 = Period.of(1, 1, 1);
    System.out.println(today.plus(period1)); // 2019-02-18

    LocalDate newDate = today.plusYears(5); // 2023-01-17
    Period period2 = Period.between(today, newDate);
    System.out.println(period2.getYears()); // 5 lat
}
```

Java 8
Duration

- `Duration` – przedział czasowy (**sekundy, minuty, godziny**)

```
public static void main(String[] args) {
    LocalTime now = LocalTime.now(); // 09:54:37.876
    Duration duration1 = Duration.of(10, ChronoUnit.MINUTES);
    System.out.println(now.plus(duration1)); // 10:04:37.876

    LocalTime newTime = now.plusHours(2); // 11:54:37.876
    Duration duration2 = Duration.between(now, newTime);
    System.out.println(duration2.getSeconds()); // 7200 sekund
}
```

Java 8
TemporalAdjusters

- Wykonywanie obliczeń na datach, np. drugi piątek miesiąca, poprzednia sobota

```
public static void main(String[] args) {
    LocalDate now = LocalDate.now(); // 2018-01-17
    LocalDate previousSaturday = now.with(TemporalAdjusters.previous(DayOfWeek.SATURDAY));
    System.out.println(previousSaturday); // 2018-01-13

    LocalDate firstFriday = now.with(TemporalAdjusters.firstInMonth(DayOfWeek.FRIDAY));
    System.out.println(firstFriday); // 2018-01-05
    LocalDate secondFriday = now.with(TemporalAdjusters.firstInMonth(DayOfWeek.FRIDAY))
        .with(TemporalAdjusters.next(DayOfWeek.FRIDAY));
    System.out.println(secondFriday); // 2018-01-12
}
```

Java 8
Ćwiczenie 21

- Oblicz przedział między dwiema datami

Java 8
Ćwiczenie 22

- Oblicz czas wykonywania się pętli (np. wypisywania liczb od 1 do 100)
- Użyj wybranych klas: `Clock, Duration, Instant`

Thanks!!

Q&A