Zaawansowana Java

dzień 2

v3.1

Java 8

Java 8

Java 8 jest to długo wyczekiwana wersją platformy programistycznej. Wydana została 18 marca 2014 roku. Datę premiery nowej platformy planowano początkowo na 2012 rok, lecz z powodu dużej ilości zmian pracę znacznie się wydłużyły

Wersja ta zawiera między innymi nowe API do obsługi daty i czasu, wyrażenia lambda, zmiany w interfejsach.

Zagadnienia te omówimy podczas kursu.

Wyrażenia lambda i interfejsy funkcyjne

Wyrażenia lambda, są jedną z najbardziej wyczekiwanych przez programistów funkcjonalności w języku Java.

Wprowadzają one elementy programowania funkcyjnego do ściśle obiektowego języka programowania jakim jest język Java.

Programowanie funkcyjne możemy rozumieć jako możliwość określenia pewnego kodu (**funkcji**) , który przekażemy dalej bez definicji specjalnej klasy ani metody.

Poprawne używanie wyrażeń lambda pozwala znacznie uprościć kod i sprawia, że staje się on bardziej zwięzły i przejrzysty, są również bardzo pomocne przy wykonywaniu operacji na kolekcjach.

Można je stosować w tych miejscach, w których oczekiwany argument jest typu będącego **interfejsem funkcyjnym**.

Interfejs funkcyjny

Interfejs funkcyjny to interfejs który może posiadać tylko jedną metodę abstrakcyjną.

Interfejs taki może być dodatkowo opatrzony adnotacją **`@FunctionalInterface`**.

Adnotacja ta nie jest obowiązkowa. Jeśli zostanie użyta, a w interfejsie pojawi się kolejna metoda abstrakcyjna, kompilator zgłosi błąd typu:

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

Interfejsy funkcjonalne dostępne w Javie 8 zgrupowane są w pakiecie **`java.util.function`**.

Interfejs funkcyjny

Listę gotowych do użycia interfejsów funkcyjnych znajdziemy pod adresem:

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Ponadto interfejsami funkcyjnymi są także **Comparator** (poznany już przez nas wcześniej) , **Runable**, **Callable** (dwa pozostałe poznamy na zajęciach dotyczących wątków).

Interfejs funkcyjny - przykład

Przypomnijmy poznany już interfejs **`Comparator`**

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
...
}
```

posiada on metodę **`compare`**, która zwraca:

- wartość ujemną jeżeli wartość pierwsza jest mniejsza od drugiej
- wartość dodatnią jeżeli wartość pierwsza jest większa od drugiej
- zero jeżeli wartości są równe

Własny Interfejs funkcyjny - przykład

Mamy możliwość skorzystania z gotowych interfejsów, ale równie łatwo możemy tworzyć swoje własne.

```
package pl.coderslab.lambda;

@FunctionalInterface
public interface PrintMessage {

        void log(String message);

}
```

Własny Interfejs funkcyjny - przykład

Mamy możliwość skorzystania z gotowych interfejsów, ale równie łatwo możemy tworzyć swoje własne.

```
package pl.coderslab.lambda;

@FunctionalInterface
public interface PrintMessage {

        void log(String message);

}
```

Interfejs taki możemy dodatkowo oznaczyć adnotacją **@FunctionalInterface**

Własny Interfejs funkcyjny - przykład

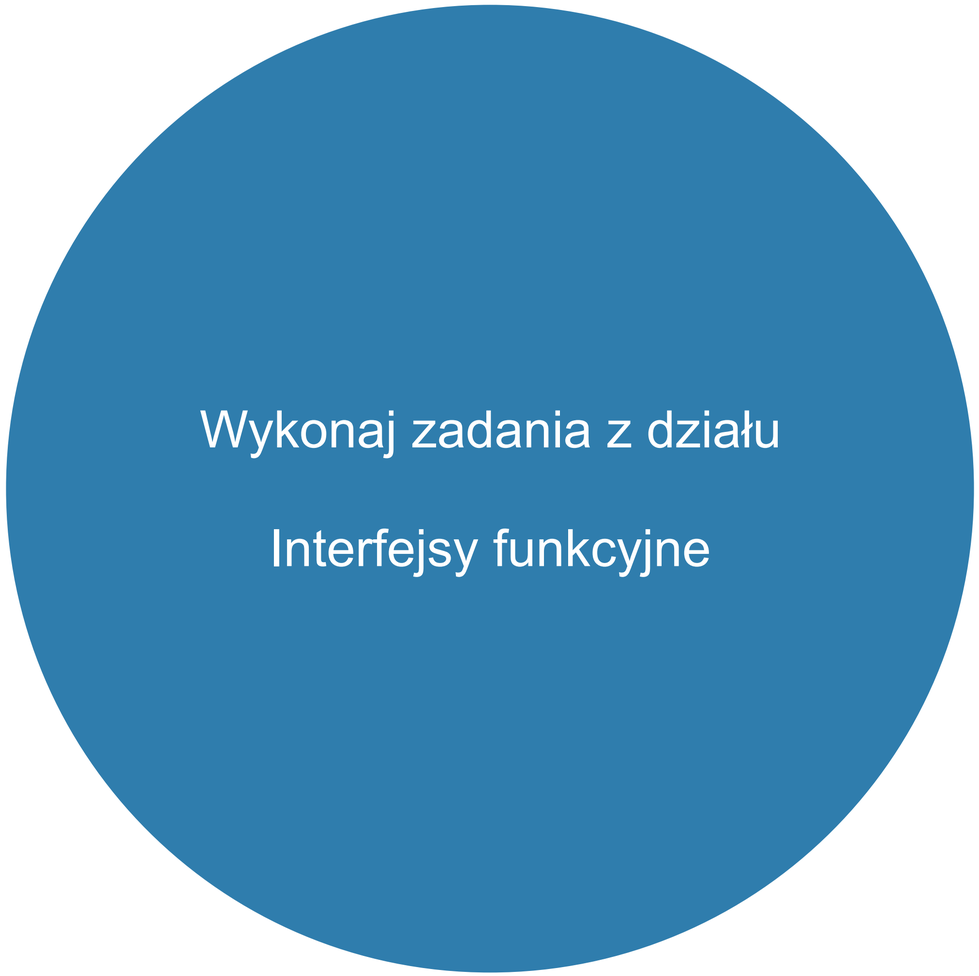
Mamy możliwość skorzystania z gotowych interfejsów, ale równie łatwo możemy tworzyć swoje własne.

```
package pl.coderslab.lambda;

@FunctionalInterface
public interface PrintMessage {

        void log(String message);

}
```

Interfejs taki możemy dodatkowo oznaczyć adnotacją **`@FunctionalInterface`**

Określamy **jedną** metodę abstrakcyjną - która w naszym przypadku przyjmuje parametr typu **String** i jest typu **void** .

Zadania



Wyrażenia lambda

Ogólna składnia wyrażenia lambda przedstawia się następująco:

```
(Type1 arg2, Type2 arg2, …) -> {
        ciało wyrażenia
}
```

arg1, **arg2**, **…** to parametry przekazywane do ciała lambdy.

W wyrażeniu lambda nie musimy podawać typu parametrów

```
(arg2, arg2, …) -> {
        ciało wyrażenia
}
```

Wyrażenia lambda

Wyrażenie lambda może nie przyjmować żadnych parametrów :

```
() -> {
        ciało wyrażenia
}
```

Jeśli ciało wyrażenia lambda składa się z tylko jednej linii kodu, można pominąć nawiasy klamrowe:

```
() -> polecenie
```

A jeśli jest tylko jeden parametr, można pominąć nawiasy okrągłe:

```
parametr -> polecenie
```

Wyrażenia lambda

Wyrażenie lambda może nie przyjmować żadnych parametrów :

```
() -> {
        ciało wyrażenia
}
```

Wyrażenie lambda może zwracać wartość:

```
() -> { return 2 + 2; }
```

Można także pominąć słowo kluczowe return oraz nawiasy:

```
() -> 2 + 2;
```

Wyrażenia lambda

Wyrażenie lambda można przypisać do zmiennej.

Przykład wyrażenia lambda gdzie jako typu używamy wcześniej utworzonego interfejsu:

```
PrintMessage pm = (s) -> System.out.println("Message to print: " + s);
```

Poniższe przykłady korzystają z interfejsów znajdujących się w pakiecie **java.util.function**:

```
Supplier<Integer> sup = () -> 12;
```

```
Predicate<String> predicate = s -> s.length() > 300;
```

Interfejsy te omówimy w dalszej części wykładu.

Przykład zastosowania:

Wyrażenie lambda można przekazać bezpośrednio jako argument metody.

Zastosowanie wyrażeń lambda najlepiej zobrazować to przykładem.

Załóżmy, że mamy za zadanie posortować alfabetycznie pewną kolekcję.

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

W tym celu można zastosować statyczną metodę **`sort()`** klasy **Collections**, która jako argumenty przyjmuje kolekcję do posortowania i komparator pozwalający ustalić właściwy porządek.

Przykład zastosowania:

Poniższy kod wykorzystuje klasę anonimową **Comparator**, która jest przekazywana do metody **sort**. Interfejs **Comparator** jest interfejsem funkcyjnym - posiada tylko jedną metodę **`compare(o1, o2)`**, można więc go sobie wyobrazić jako funkcję, która przyjmuje dwa argumenty i zwraca wartość typu int.

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

Przykład zastosowania:

Poniższy kod wykorzystuje klasę anonimową **Comparator**, która jest przekazywana do metody **sort**. Interfejs **Comparator** jest interfejsem funkcyjnym - posiada tylko jedną metodę **`compare(o1, o2)`**, można więc go sobie wyobrazić jako funkcję, która przyjmuje dwa argumenty i zwraca wartość typu int.

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

Implementujemy wymaganą przez interfejs metodę o nazwie **compare**.

Przykład zastosowania:

Poniższy kod wykorzystuje klasę anonimową **Comparator**, która jest przekazywana do metody **sort**. Interfejs **Comparator** jest interfejsem funkcyjnym - posiada tylko jedną metodę **`compare(o1, o2)`**, można więc go sobie wyobrazić jako funkcję, która przyjmuje dwa argumenty i zwraca wartość typu int.

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

Implementujemy wymaganą przez interfejs metodę o nazwie **compare**.

Wykorzystujemy w tym celu metodę **compareToIgnoreCase** klasy **String** : https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#compareToIgnoreCase-java.lang.String- .

Przykład zastosowania:

W miejsce klasy anonimowej możemy wstawić wyrażenie lambda:

```
Collections.sort(names, (String s1, String s2) ->{
        return s1.compareToIgnoreCase(s2);
});
```

Jak widać, dzięki zastosowaniu wyrażenia lambda, kod stał się o wiele krótszy, a można go skrócić jeszcze bardziej:

```
Collections.sort(names, (String s1, String s2) -> s1.compareToIgnoreCase(s2));
```

W przypadku funkcji, których ciało mieści się w jednej linii można pominąć nawiasy klamrowe oraz słowo kluczowe **return**.

Ponieważ lista zawiera metodę **sort**, można także napisać:

```
names.sort((s1, s2) -> s1.compareToIgnoreCase(s2));
```

Interfejsy funkcyjne - przykład

Formę interfejsu funkcyjnego może przyjąć każdy interfejs, pod warunkiem że posiada tylko jedną metodę abstrakcyjną. Podczas tworzenia własnych interfejsów funkcyjnych warto opatrzyć je adnotacją **@FunctionalInterface**, np.:

```
@FunctionalInterface
public interface MathOperation {
        int operate(int i1, int i2);
}
```

Interfejsy funkcyjne - przykład

Dzięki takiemu podejściu jesteśmy w stanie tworzyć kod bardziej uniwersalny:

```
public static void main(String[] args) {
    MathOperation addition = (i1, i2) -> i1 + i2;
    MathOperation subtraction = (i1, i2) -> i1 - i2;
    MathOperation multiplication = (i1, i2) -> i1 * i2;
    MathOperation division = (i1, i2) -> i1 / i2;

    System.out.println("2 + 9 = " + addition.operate(2, 9));
    System.out.println("7 - 5 = " + subtraction.operate(7, 5));
    System.out.println("3 * 7 = " + multiplication.operate(3, 7));
    System.out.println("15 / 5 = " + division.operate(15, 5));
}
```

Interfejsy funkcyjne - przykład

Dzięki takiemu podejściu jesteśmy w stanie tworzyć kod bardziej uniwersalny:

```
public static void main(String[] args) {
    MathOperation addition = (i1, i2) -> i1 + i2;
    MathOperation subtraction = (i1, i2) -> i1 - i2;
    MathOperation multiplication = (i1, i2) -> i1 * i2;
    MathOperation division = (i1, i2) -> i1 / i2;

    System.out.println("2 + 9 = " + addition.operate(2, 9));
    System.out.println("7 - 5 = " + subtraction.operate(7, 5));
    System.out.println("3 * 7 = " + multiplication.operate(3, 7));
    System.out.println("15 / 5 = " + division.operate(15, 5));
}
```

Tworzymy różne operacje.

Interfejsy funkcyjne - przykład

Dzięki takiemu podejściu jesteśmy w stanie tworzyć kod bardziej uniwersalny:

```
public static void main(String[] args) {
    MathOperation addition = (i1, i2) -> i1 + i2;
    MathOperation subtraction = (i1, i2) -> i1 - i2;
    MathOperation multiplication = (i1, i2) -> i1 * i2;
    MathOperation division = (i1, i2) -> i1 / i2;

    System.out.println("2 + 9 = " + addition.operate(2, 9));
    System.out.println("7 - 5 = " + subtraction.operate(7, 5));
    System.out.println("3 * 7 = " + multiplication.operate(3, 7));
    System.out.println("15 / 5 = " + division.operate(15, 5));
}
```

Wywołujemy utworzone metody.

Interfejsy funkcyjne

W wielu wypadkach nie musimy tworzyć własnych interfejsów funkcyjnych aby skorzystać z wyrażeń lambda.

Java posiada sporą ilość predefiniowanych interfejsów funkcyjnych, do których można dopasować wyrażenie lambda uwzględniając ilość i rodzaj argumentów oraz zadeklarowanych w ich metodach abstrakcyjnych zwracanych wartości.

Wiele z tych interfejsów znaleźć można w dokumentacji pakietu **java.util.function**.

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Interfejsy z java.util.function

Pakiet **java.util.function** zawiera np. interfejs **Consumer<T>** , który deklaruje metodę abstrakcyjną **`accept(Object)`** i nie zwraca żadnej wartości.

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

Interfejsy z java.util.function

Innym często wykorzystywanymi interfejsem funkcyjnym, które możemy znaleźć w **pakiecie java.util.function** jest:

Supplier<T>

Nie przyjmuje żadnego argumentu, zwraca dowolny obiekt. Posiada tylko jedną metodę **`get()`**.

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

Interfejsy z java.util.function

Function<T,R>

Przyjmuje jeden argument dowolnego typu i zwraca obiekt dowolnego typu. Interfejs ten posiada abstrakcyjną metodę **`apply()`**, która może posłużyć do wykonania pewnej akcji i zwrócenia jej wyniku.

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

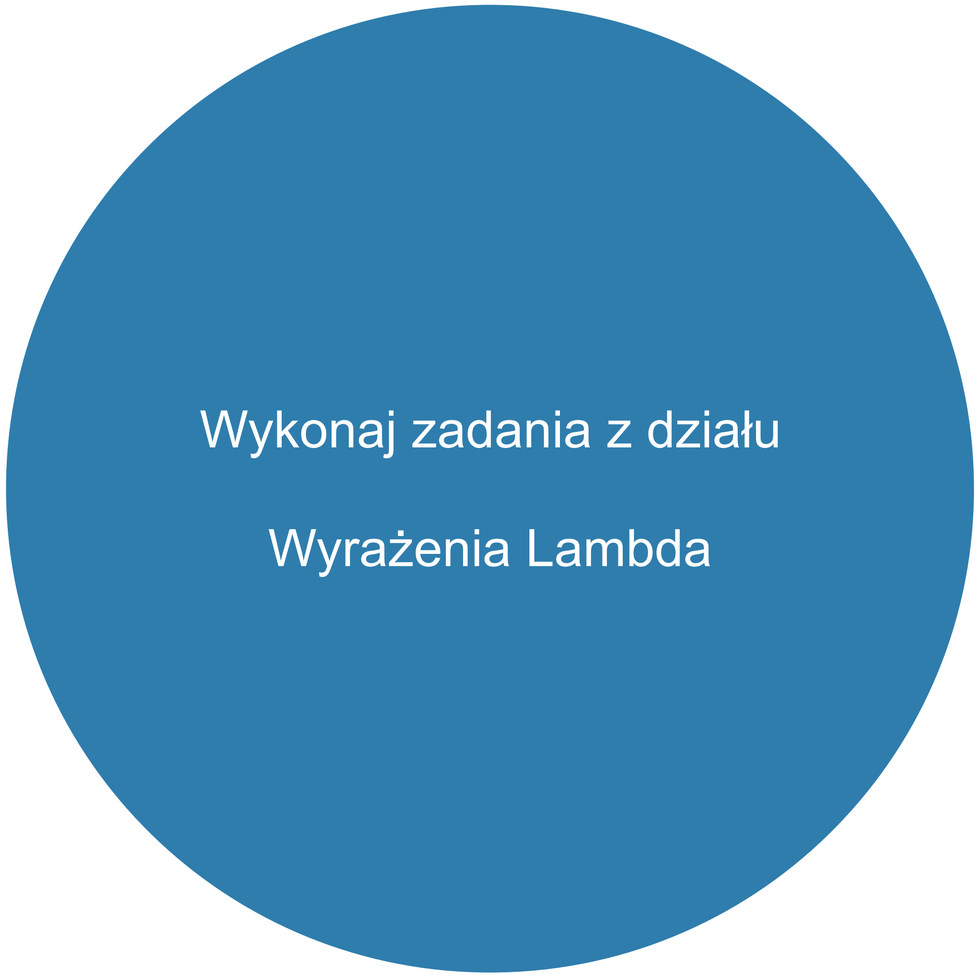
Interfejsy z java.util.function

Predicate<T>

Przyjmuje dowolny obiekt, zwraca true lub false. Posiada metodę test(). Używany jest często, gdy chcemy sprawdzić jakiś warunek.

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

Zadania



Referencje do metod

Referencje do metod

Wraz w pojawieniem się wyrażeń lambda, Java 8 wzbogacona została o możliwość stosowania referencji do metod.

Odbywa się to poprzez użycie symbolu **::** (podwójnego dwukropka), np.:

```
Object::methodName
```

Dzięki takiemu podejściu można przypisać metodę do zmiennej bez jej wykonania, lub podobnie jak to miało miejsce w przypadku wyrażeń lambda, przekazać ją w parametrze do innej metody.

Ponieważ konstruktor jest również specyficznym rodzajem metody, możliwe jest także przekazanie go jako argumentu (referencja na konstruktor).

Referencje do metod - przykład

Do zobrazowania wykorzystania **referencji do metody** skorzystamy z interfejsu **Consumer** w celu wyświetlenia elementów listy.

Rozważmy poniższy kod:

```
public static void printNames(List<String> list){
        for (String string : list) {
                System.out.println(string);
        }
}
```

Utworzyliśmy metodę, która w doskonale nam już znany sposób wyświetla elementy listy przekazanej jako parametr do naszej metody.

Referencje do metod - przykład

W kolejnym kroku tworzymy listę i wypełniamy ją elementami.

```
public static void main(String[] args) {
    List<String> names = new ArrayList<String>();

    names.add("Wojtek"); names.add("Kasia");
    names.add("Ania");  names.add("Maciek");

    Consumer<List<String>> listConsumer = (l) -> printNames(l);
    listConsumer.accept(names);
}
```

Referencje do metod - przykład

W kolejnym kroku tworzymy listę i wypełniamy ją elementami.

```
public static void main(String[] args) {
    List<String> names = new ArrayList<String>();

    names.add("Wojtek"); names.add("Kasia");
    names.add("Ania");  names.add("Maciek");

    Consumer<List<String>> listConsumer = (l) -> printNames(l);
    listConsumer.accept(names);
}
```

Wykorzystując interfejs **Consumer** tworzymy **lambda wyrażenie**, które korzysta z naszej metody.

Referencje do metod - przykład

Następnie uprościmy naszą metodę do postaci:

```
public static void printNames(List<String> list){
        list.forEach((item) -> System.out.println(item));
}
```

Dzięki referencji, powyższy kod realizujący drukujący elementy listy możemy zapisać w jeszcze krótszej postaci:

```
list.forEach(System.out::println);
```

Instrukcja **`forEach()`** została także dodana w Javie 8 i zostanie szerzej omówiona w dalszej części.

Metody domyślne i statyczne w interfejsach

Metody domyślne i statyczne

Ósma wersja Javy przyniosła też drobne zmiany w interfejsach.

Do wersji 7 mogły one zawierać wyłącznie sygnatury metod bez ich implementacji.

Od wersji 8 istnieje możliwość deklaracji ciała metody w interfejsie, przy zachowaniu jednocześnie wielokrotnego dziedziczenia.

Metody takie nazywa się metodami domyślnymi i deklaruje się je z użyciem słowa kluczowego **default**.

```
public interface User {

        public int getId();

        default int getAdminId() {
                return 1;
        }
}
```

Metody domyślne i statyczne

Metody domyślne mogą, ale nie muszą być definiowane w klasach implementujących dany interfejs.

Pozwala to uniknąć powielania kodu w przypadku metod, których ciało wygląda identycznie w każdej z implementujących je klas.

Jeśli metoda domyślna będzie zdefiniowana w klasie implementującej interfejs, jej implementacja domyślna zostanie nadpisana.

Metody domyślne nie są abstrakcyjne, więc mogą być umieszczane w interfejsach funkcyjnych, nie mogą być jednak użyte w wyrażeniach lambda.

Metody domyślne i statyczne

Można zablokować możliwość nadpisania metody domyślnej w klasie implementującej interfejs deklarując ją z użyciem słowa kluczowego **static**.

```
public interface User {

    public int getId();
    default int getAdminId() {
        return 1;
    }
    static void printMsg(String msg) {
        System.out.println(msg);
    }
}
```

Tworząc interfejsy zawierające metody domyślne należy pamiętać, że nie należy ich stosować do implementacji wielokrotnego dziedziczenia. Może to spowodować błędy kompilacji w przypadku konfliktu – gdy wiele interfejsów deklaruje tą samą metodę domyślną.

Stream API

Stream API

Nowością zaprezentowaną w Javie 8 są także **strumienie danych** (nie mylić z InputSteam, OutputStream czy Java I/O).

Strumień (ang. Stream) reprezentuje sekwencję elementów, na których może być wykonany wiele operacji w sposób równoległy lub sekwencyjny bez konieczności przechowywania wartości pośrednich.

Strumień tworzy zatem potok różnych operacji na elementach.

Potok operacji ma swoje źródło (można to nazwać źródłem strumienia), którym może być np. kolekcja, tablica czy napis.

Stream API

Stream API zdefiniowany został w pakiecie **java.util.stream**, a jego centralnym elementem jest interfejs **`Stream<T>`**.

Operacje wykonywane na strumieniach dzielą się na:

- **pośrednie** (ang. intermediate)
- **i kończące/terminalne** (ang. terminal).

Dodatkowo operacje mogą być:

- **bezstanowe** – wykonywane niezależnie od innych danych (np. filter), mogą być wykonywane równolegle
- **stanowe** – ich stan zależy od wcześniejszych danych (np. sort), nie mogą być wykonywane równolegle
- **redukcyjne** – redukują dane (np. max)

Stream API

Operacje pośrednie zwracają w wyniku przetworzony strumień, na którym można wykonać kolejne operacje.

Można w ten sposób tworzyć łańcuch operacji (ang. chain).

Operacje terminalne natomiast zamykają strumień i zwracają wynik lub nie. W strumieniu może wystąpić tylko jedna metoda kończąca.

W ten sposób budowany jest swoisty rurociąg (ang. pipeline) posiadający swoje wejście, łańcuch operacji i wyjście.

Listę wszystkich operacji na strumieniach można znaleźć w dokumentacji interfejsu Stream:

http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Stream API

Operacje pośrednie są wykonywane leniwie. Oznacza to, że nie są one wykonywane aż do momentu wywołania metody terminalnej.

Transformacje strumienia nie modyfikują danych wejściowych, z których został utworzony strumień.

Domyślnie operacje na strumieniu wykonywane są sekwencyjnie (w jednym wątku), można jednak „przełączyć” strumień do wykonywania równoległego (za pomocą metody **`parallel()`**), czyli w wielu wątkach.

Stream API

W operacjach strumieniowych mogą być podawane wyrażenia lambda, dzięki czemu kod jest bardziej zwięzły i czytelny.

Ogólnie pracę ze strumieniami można podzielić na 3 etapy:

1. Tworzenie strumienia.
2. Przetwarzanie strumienia, czyli wykonywanie operacji pośrednich.
3. Zamykanie strumienia poprzez użycie metody kończącej (terminalnej).

Tworzenie strumieni

Strumienie można tworzyć na wiele sposobów, m.in.:

- z kolekcji – za pomocą metody **`stream()`** lub **`parallelStream()`** interfejsu Collection, np.: **`list.stream()`**. Metody te zostały dodane do interfejsu **Collection** jako metody domyślne.
- z tablicy – za pomocą statycznej metody stream z klasy **Arrays**, np. **`Arrays.stream(array)`**
- z napisów – za pomocą metody **`chars()`**
- z podanych wartości – za pomocą metody **`Stream.of(zestaw wartości)`**
- z plików – za pomocą statycznej metody **`lines()`** klasy **Files**
- z katalogów (strumień reprezentujący drzewo katalogowe obiektów plikowych) za pomocą metody **`walk()`** lub **`find()`** klasy **Files**

Przykład zastosowania strumieni

```
List<String> fruits =
Arrays.asList("orange", "lemon", "peach", "banana", "plum",
    "cherry", "apple", "pomelo");

fruits.stream()
    .filter(s -> s.startsWith("p"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Przykład zastosowania strumieni

```
List<String> fruits =
Arrays.asList("orange", "lemon", "peach", "banana", "plum",
    "cherry", "apple", "pomelo");

fruits.stream()
    .filter(s -> s.startsWith("p"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Za pomocą metody **`stream()`** tworzymy strumień.

Przykład zastosowania strumieni

```
List<String> fruits =
Arrays.asList("orange", "lemon", "peach", "banana", "plum",
    "cherry", "apple", "pomelo");

fruits.stream()
    .filter(s -> s.startsWith("p"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Przy pomocy operacji **`filter()`** z kolekcji wybrane zostały tylko te elementy, które zaczynają się na literę **p**. Wykorzystujemy poznane wcześniej wyrażenie lambda - wykorzystany interfejs to **Predicate<T>**.

Przykład zastosowania strumieni

```
List<String> fruits =
Arrays.asList("orange", "lemon", "peach", "banana", "plum",
    "cherry", "apple", "pomelo");

fruits.stream()
    .filter(s -> s.startsWith("p"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Za pomocą operacji **`map()`** wszystkie znaki napisów dostały zamienione na wielkie litery.

Przykład zastosowania strumieni

```
List<String> fruits =
Arrays.asList("orange", "lemon", "peach", "banana", "plum",
    "cherry", "apple", "pomelo");

fruits.stream()
    .filter(s -> s.startsWith("p"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Następnie posortowane alfabetycznie za pomocą metody **`sorted()`**.

Przykład zastosowania strumieni

```
List<String> fruits =
Arrays.asList("orange", "lemon", "peach", "banana", "plum",
    "cherry", "apple", "pomelo");

fruits.stream()
    .filter(s -> s.startsWith("p"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Ostatecznie za pomocą terminalnej metody **`forEach()`** wynik wszystkich operacji pośrednich został przekazany na konsolę.

Przykład zastosowania strumieni

W wyniku wykonania przykładowego kodu na standardowym wyjściu otrzymamy:

```
PEACH
PLUM
POMELO
```

Kolekcja wejściowa **fruits** z powyższego przykładu nie została zmodyfikowana w wyniku przetworzenia strumienia. Aby zapisać wynik działania strumienia w nowej zmiennej możemy skorzystać z metody kończącej **`collect()`**:

```
List<String> newList = fruits.stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .collect(Collectors.toList());
```

Przykłady operacji pośrednich i kończących

Metoda	Typ	Opis
distinct()	pośrednia	Zwraca strumień zawierający dane unikalne
sorted()	pośrednia	Zwraca strumień w formie posortowanej
unordered()	pośrednia	Przekształca strumień do postaci nieuporządkowanej
limit(int n)	pośrednia	Zwraca strumień ograniczony do n pierwszych elementów
map(Function…)	pośrednia	Zwraca przekształcony przy pomocy funkcji strumień
parallel()	pośrednia	Strumień równoległy
sequential()	pośrednia	Strumień sekwencyjny

Przykłady operacji pośrednich i kończących

Metoda	Typ	Opis
noneMatch(Predicate p)	kończąca	Sprawdza czy żaden z elementów nie spełnia warunków
findAny()	kończąca	Zwraca dowolny element strumienia
findFirst()	kończąca	Zwraca pierwszy element strumienia
count()	kończąca	Zwraca ilość elementów strumienia
forEach(Consumer c)	kończąca	Wykonuje akcje na każdym elemencie strumienia

Przykłady operacji pośrednich i kończących

Metoda	Typ	Opis
collect(…)	kończąca	Grupuje wszystkie elementy strumienia i zwraca w postaci zdefiniowanej przez dany Collector
max(…)	kończąca	Zwraca wartość największą wynikającą z zastosowanego komparatora
min(…)	kończąca	Zwraca wartość najmniejszą wynikającą z zastosowanego komparatora
toArray(…)	kończąca	Przekształca strumień do tablicy

Zadania