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

JAVA FOUNDATIONS: EXAM NUMBER: 1Z0-811

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

ZAKRES SZKOLENIA – DZIEŃ 2

- **Typy generyczne w Javie**

  (klasy i interfejsy generyczne)

- **Kolekcje w Javie**

  ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap

- **Programowanie funkcyjne w Javie**

  (interfejsy funkcyjne, klasa Stream, klasa Optional

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

KOLEKCJE I TYPY GENERYCZNE

TYPY GENERYCZNE

Typy generyczne są nazywane inaczej **szablonami**. Umożliwiają one parametryzowanie **klasy/metody/interfejsu**, które realizowane jest poprzez przekazanie typów argumentów w **momencie rzeczywistego wykorzystania ich w kodzie**. Wykorzystując taki mechanizm, możemy wielokrotnie wykorzystywać te same fragmenty kodu, ponieważ nie są na sztywno powiązane z żadną konkretną implementacją. **Dzięki typom generycznym dodatkowo jesteśmy w stanie pozbyć się zbędnego rzutowania**.

TYPY GENERYCZNE - PRZYKŁAD

W przypadku przedstawionego fragmentu kodu jesteśmy w stanie przekazać dowolny obiekt do obiektu klasy Box (ze względu na korzystanie z typu Object). Niemniej jednak, **nie jesteśmy w żaden sposób chronieni na poziomie kompilacji** przed omyłkowym przekazywaniem różnego typu obiektów do tego samego obiektu klasy Box, co **może prowadzić do błędów w trakcie działania aplikacji**. Dzięki możliwości wykorzystania typów generycznych, kompilator jest w stanie sprawdzić poprawność przekazywanych typów.

```
public class Box {

    private Object item;

    public Object getItem() {
        return item;
    }

    public void setItem(Object item) {
        this.item = item;
    }
}
```

```
public class Box<T> {
    private T item;

    public T getItem() {
        return item;
    }

    public void setItem(T item) {
        this.item = item;
    }
}
```

TYPY GENERYCZNE KONWENCJA NAZEWNICZA

Zgodnie z konwencją, **nazwy typów generycznych są najczęściej zapisywane za pomocą pojedynczej wielkiej litery**. Sugeruje się wykorzystywać poniższe formy nazewnicze:

- **E** - Element (używany m.in. dla Java Collection API)
- **K** – Klucz
- **N** – Liczba
- **T** – Typ
- **V** - Wartość

KLASY GENERYCZNE - KREACJA

W celu stworzenia obiektu, do którego musimy przekazać wartość typu generycznego, **musimy podczas jego tworzenia podać konkretny typ, który zastępuje parametr, np. T**:

```
new Box<Integer>();
```

Tak stworzony obiekt możemy przypisać do odpowiedniej referencji, która również powinna mieć informacje, jaki typ generyczny został wyspecyfikowany, tzn.:

```
Box<Integer> numberBox = new Box<Integer>(); // T zastąpione przez Integer
```

KLASY GENERYCZNE - KREACJA

W Javie, jeżeli referencja, do której przypisujemy tworzony obiekt generyczny, zawiera informacje o typie, **typu tego nie musimy powtarzać tworząc obiekt, ale wykorzystanie <> jest ciągle obowiązkowe**:

```
Box<Integer> numberBox = new Box<>(); // <> wymagane
```

Z kolei typ ten musimy wyspecyfikować podczas tworzenia obiektu, jeżeli korzystamy ze słowa kluczowego var, np.:

```
var intList = new ArrayList<Integer>();
```

ILOŚĆ TYPÓW GENERYCZNYCH

W ramach jednej klasy można zadeklarować wiele typów generycznych, które będą częścią klasy. Każdy parametr typu powinien być deklarowany jako unikalny znak zgodnie z konwencją.

Kolejny przykład definiuje parę dwóch generycznych obiektów. Zarówno pole **key** jak i pole **value** może **mieć dowolny typ**.

Tworzenie instancji w przypadku powyższej klasy będzie wyglądać następująco:

```
Pair<String, Float> pair = new Pair<String, Float>();
```

```
public class Pair<K, V> {
    private K key;
    private V value;

    public K getKey() {
        return key;
    }

    public void setKey(K key) {
        this.key = key;
    }

    public V getValue() {
        return value;
    }

    public void setValue(V value) {
        this.value = value;
    }
}
```

ROZSZERZANIE KLASY GENERYCZNEJ

Klasę generyczną możemy bez problemu rozszerzyć. **Klasa dziedzicząca klasę generyczną musi wyspecyfikować typ generyczny lub dalej pozostać generyczna**. Poniższe definicje klas pokazują te możliwości:

```
public class BaseClass<T, V> {
}

// wyspecyfikowano typy generyczne
public class NoLongerGenericClass extends BaseClass<String, Integer> {
}

// wyspecyfikowano jeden typ generyczny - V.
// Klasa jest ciągle generyczna i wymaga wyspecyfikowania parametru T.
public class StillGenericClass<T> extends BaseClass<T, Integer> {
}
```

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

METODY GENERYCZNE

Nie tylko klasy mogą być generyczne. Również metody umożliwiają **deklarowanie własnych typów parametrycznych**. Widoczność tych typów jest ograniczona do konkretnej metody (sygnatury i ciała). Dozwolone są zarówno **statyczne, jak i niestatyczne metody generyczne**. Składnia metod generycznych rozszerza deklarację metod o typy parametryczne, umieszczone przed typem zwracanym. Podczas korzystania z takich metod, nie musimy, ale **możemy wyspecyfikować typy generyczne**. Robimy to za pomocą nawiasów <>, w których podajemy wartości typów generycznych.

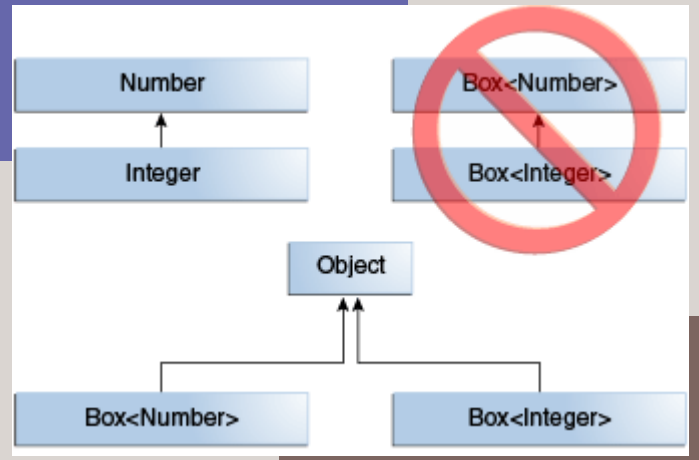
```
public class PairGenerator {
  public static <K, V> Pair<K, V> generatePair(K key, V value) {
    Pair<K, V> pair = new Pair<K, V>();
    pair.setKey(key);
    pair.setValue(value);
    return pair;
  }

  public static void main(String[] args) {
    final Pair<Integer, String> firstPair = PairGenerator.generatePair(1, "value1");
    final Pair<Long, String> secondPair = PairGenerator
        .<Long, String>generatePair(2L, "value2");
  }
}
```

PODTYPY

W przypadku typów generycznych, **błędem jest traktowanie parametrów typów w ten sam sposób, co klasy generyczne**, np. jeśli **Integer jest podtypem Number**, to **nie znaczy to, że typ Box<Integer> jest podtypem Box<Number>**.

W celu uzyskania oczekiwanej relacji powinno się korzystać z tzw. Wildcards.



KOLEKCJE

Kolekcje, często nazywane również **kontenerami**, są obiektami, które agregują elementy. Kolekcje wykorzystywane są do **przechowywania obiektów**, uzyskiwania przechowywanych danych lub manipulowania danymi. W Javie wbudowane w język kolekcje opierają się o poniższe mechanizmy:

- **Interfejs**: abstrakcyjny typ danych reprezentujący kolekcje.
- **Implementacja**: konkretna realizacja interfejsów kolekcji.
- **Algorytm**: przydatne operacje na strukturach danych

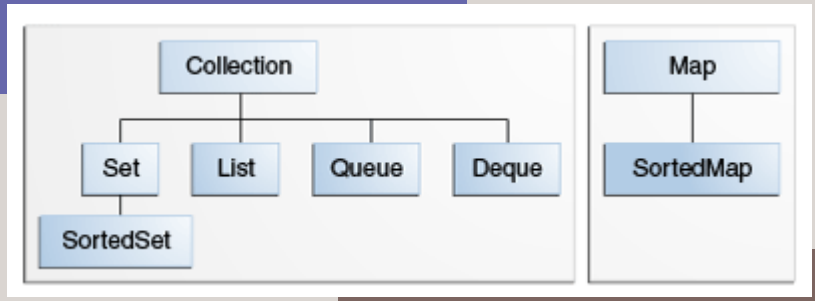
KOLEKCJE

Kolekcje w Javie dają programistom następujące korzyści:

- zwalniają programistę z **obowiązku implementacji struktur danych od zera**, co redukuje czas potrzebny na realizację konkretnej funkcjonalności,
- zaimplementowane struktury danych wykorzystują **najbardziej efektywne mechanizmy, a ich implementacja jest optymalna**,
- w przypadku projektowania własnych struktur danych, nie ma potrzeby „wymyślania koła na nowo”. **Można ponownie użyć już istniejące**.

INTERFEJSY

Interfejsy kolekcji reprezentują kontenery różnego typu. Interfejsy te umożliwiają manipulowanie kolekcjami bez wnikania w szczegóły implementacyjne. Rdzeniem wszystkich kolekcji jest interfejs generyczny: **public interface Collection<E>**



INTERFEJSY

Poniżej znajdują się wszystkie bazowe interfejsy **Collection API**:

- **Set** - kolekcja, która nie może przechowywać duplikatów.
- **List** - kolekcja uporządkowana, może zawierać duplikaty.
- **Queue** - kolekcja realizująca mechanizm FIFO (first in, first out) lub LIFO (last in, first out).
- **Deque** - rodzaj kolejki, gdzie można dodawać i usuwać elementy, zarówno z początku, jak i końca.
- **Map** - kolekcja służąca do przechowywania par klucz-wartość.

SET

java.util.Set jest generyczną strukturą danych odpowiedzialną za przechowywanie elementów unikalnych. Set korzysta tylko i wyłącznie z metod interfejsu Collection. Dodatkowo wprowadza silne połączenie pomiędzy metodami: **equals** i **hashCode**. Istnieją trzy podstawowe implementacje interfejsu Set: **HashSet**, **TreeSet**, **LinkedHashSet**.

SET

- **HashSet**
  - kolejność elementów nie jest zachowana
  - przechowuje informacje w tablicy hashującej
- **TreeSet**
  - kolejność elementów jest zachowana według tzw. kolejności naturalnej lub wg pewnego Comparatora
  - przechowuje dane w drzewie czerwono-czarnym
- **LinkedHashSet**
  - zachowuje informacje o kolejności dodawania poszczególnych elementów
  - implementacja jest oparta o tablicę hashującą wraz z obsługą linked list

```java
final Set<Integer> numbersSet = new HashSet<>(); // stworzenie instancji HashSet
System.out.println(numbersSet.isEmpty()); // true, Set nie zawiera elementów
numbersSet.add(1);
numbersSet.add(17);
numbersSet.add(3);
numbersSet.add(2);
numbersSet.add(1); // Dodanie elementu o wartości,
// który już istnieje - element NIE jest ponownie dodany
numbersSet.forEach(System.out::println);

/* przykładowa kolejność, w jakiej elementy mogą być wypisane:
 1 17 2 3
*/
```

```java
final Set<Integer> numbersSet = new TreeSet<>();
numbersSet.add(1);
numbersSet.add(3);
numbersSet.add(2);
numbersSet.add(1); // Dodanie elementu o wartości,
// który już istnieje - element NIE jest ponownie dodany
numbersSet.forEach(System.out::println);
/* Kolejność elementów ZAWSZE będzie taka sama
   (posortowana wg naturalnej kolejności):
 1 2 3
*/
```

```java
final Set<Integer> numbersSet = new LinkedHashSet<>();
numbersSet.add(1);
numbersSet.add(3);
numbersSet.add(2);
numbersSet.add(1); // Dodanie elementu o wartości,
// który już istnieje - element NIE jest ponownie dodany
numbersSet.forEach(System.out::println);
/* Kolejność elementów ZAWSZE będzie taka sama
   (wg kolejności dodawania elementów)
 1 3 2
*/
```

LIST

java.util.List jest interfejsem, który reprezentuje kolekcje uporządkowane. Charakteryzuje się tym, że:

- może zawierać **zduplikowane elementy** (tzn. o takiej samej wartości)
- element możemy **pobrać na podstawie jego pozycji** w liście (na podstawie indeksu)
- **element możemy wyszukać**.

Najczęściej wykorzystywanymi implementacjami interfejsu List są:

- **ArrayList**, która oparta jest o strukturę tablicową
- **LinkedList**, która zaimplementowana jest na zasadzie węzłów.

```
final List<String> names = new ArrayList<>();
names.add("Andrzej");  // dodanie elementu na koniec listy
names.add("Grzegorz"); // dodanie elementu na koniec listy
for (final String name: names) {
  System.out.println(name);
  // na ekran zostanie wypisane Andrzej, Grzegorz, z zachowaniem kolejności
}
```

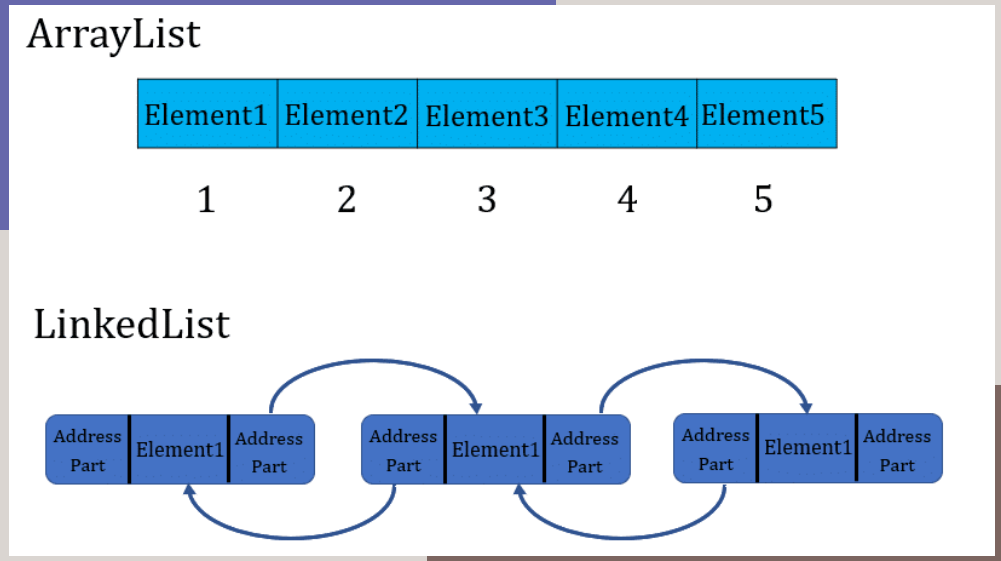
```
final List<String> names = new LinkedList<>();
names.add(0, "Andrzej");  // dodanie elementu na początek listy
names.add(0, "Grzegorz"); // dodanie elementu na początek listy
for (final String name: names) {
  System.out.println(name);
  // na ekran zostanie wypisane Grzegorz, Andrzej, z zachowaniem kolejności
}
```

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

ARRAYLIST VS LINKEDLIST

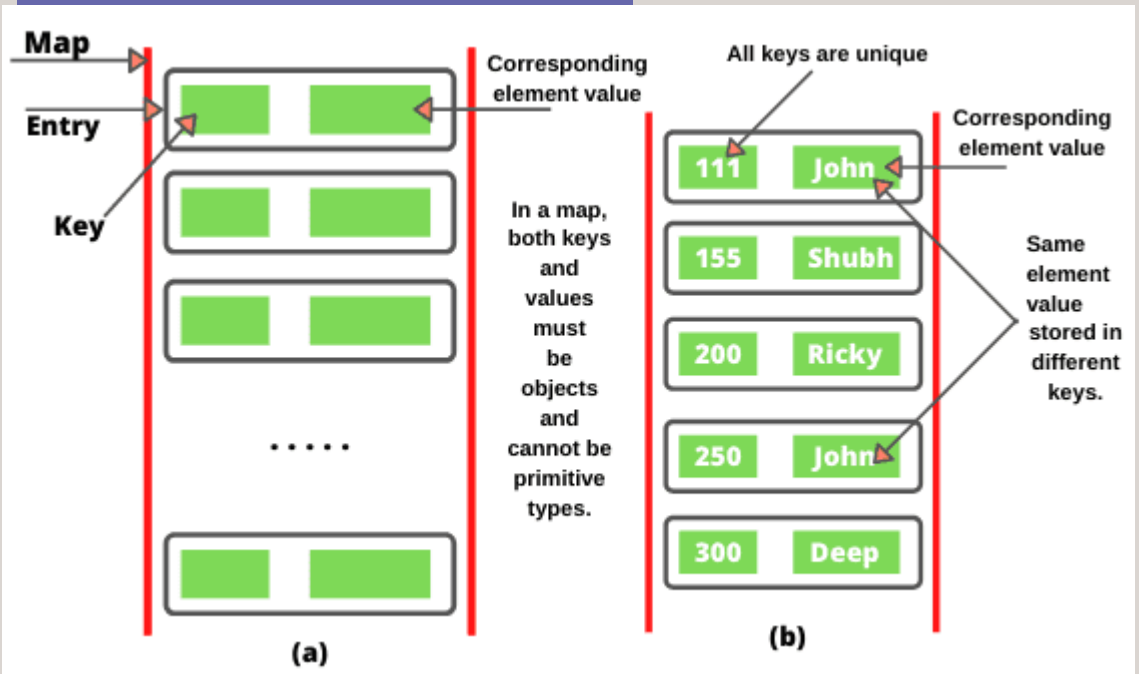
Kiedy powinniśmy korzystać z **ArrayList**, a kiedy z **LinkedList**?

- pobieranie elementu na podstawie jego indeksu z **ArrayList** jest szybsze (**O(1)**) niż z **LinkedList** (**O(n)**)
- dodawanie elementu za pomocą metody **add(E someELement)** ma taką samą złożoność obliczeniową dla obu implementacji, ale w przypadku przepełnienia wewnętrznej tablicy w ArrayList, ta operacja jest wolniejsza (**O(n)**)
- dodawanie elementu na konkretny indeks, tzn. za pomocą metody **add(int index, E someElement)**, jest szybsze w przypadku LinkedList.



MAP

Interfejs **java.util.Map** jest strukturą danych, umożliwiającą operowanie na danych w postaci **klucz-wartość**. Każdy **klucz** w takim obiekcie **musi być unikalny**, tzn. jeden klucz może zawierać dokładnie jedną wartość.



MAP

Metody mapy, które służą do wykonywania podstawowych operacji, to m.in.:

- **put** - służy do dodania odpowiedniej pary do kolekcji lub zastąpienia starej wartości nową dla konkretnego klucza
- **get** - służy do pobierania wartości na podstawie klucza
- **remove** - usuwa element na podstawie klucza (lub dodatkowo wartości)
- **containsKey** - zwraca informację, czy istnieje wartość w mapie dla danego klucza
- **containsValue** - zwraca informację, czy istnieje klucz w mapie dla danej wartości

MAP

- **size** - zwraca ilość par (tzw. **entry**) znajdujących się w kolekcji
- **isEmpty** - zwraca informację, czy mapa jest pusta

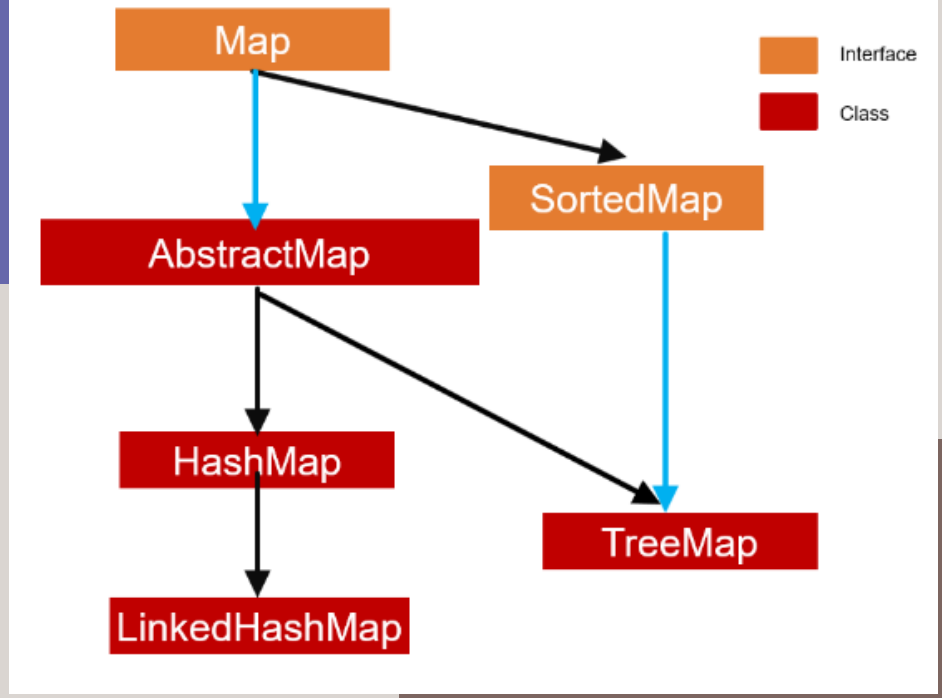
UWAGA: **Wartość null może być kluczem w mapie.**

Metoda zwracające elementy w formie innej kolekcji:

- **keySet** – zwraca zbiór kluczy jako Set
- **values** – zwraca wszystkie elementy jako Collection
- **entrySet** – zwraca Set obiektów klucz-wartość

MAP - IMPLEMENTACJE

- **HashMap**
  - kolejność par nie jest zachowana
  - przechowuje informacje w tablicy hashującej
- **TreeMap**
  - kolejność par jest zachowana według tzw. kolejności naturalnej kluczy lub wg pewnego Comparatora kluczy
  - przechowuje dane w drzewie czerwono-czarnym
- **LinkedHashMap**
  - zachowuje informację o kolejności dodawania poszczególnych par
  - implementacja jest oparta o tablicę hashującą, wraz z obsługą linked list



MAP – IMPLEMENTACJE - PRZYKŁADY

```
final Map<Integer, String> ageToNames = new HashMap<>(); // tworzenie HashMapy
ageToNames.put(11, "Andrzej"); // dodawanie elementów
ageToNames.put(22, "Michal");  // dodawanie kolejnej pary
ageToNames.put(33, "Janusz");  // dodanie trzeciej pary do mapy
ageToNames.remove(22);     // usunięcie elementu na podstawie klucza
// wyświetlenie wartości na podstawie klucza 11 -> Andrzej
System.out.println(ageToNames.get(11));
```

```
final Map<Integer, Integer> numberToDigitsSum = new TreeMap<>();
numberToDigitsSum.put(33, 6);
numberToDigitsSum.put(19, 10);
numberToDigitsSum.put(24, 6);
numberToDigitsSum.forEach((key, value) -> System.out.println(key + " " + value));

/* Elementy zawsze zostaną wypisane w tej samej kolejności:
  19 10
  24 6
  33 6
*/
```

```
// stworzenie LinkedHashMap
final Map<Integer, String> ageToNames = new LinkedHashMap<>();
ageToNames.put(20, "Gosia");
ageToNames.put(40, "Kasia");
ageToNames.put(30, "Ania");

// iteracja po kluczach z wykorzystaniem keySet()
for (final Integer key : ageToNames.keySet()) {
  // kolejność kluczy zawsze taka sama -> 20, 40, 30
  System.out.println("Key is map: " + key);
}

// iteracja po wartościach z wykorzystaniem values()
for (final String value : ageToNames.values()) {
  // kolejność wartości zawsze taka sama -> Gosia, Ania, Kasia
  System.out.println("Value in map is: " + value);
}

// iteracja po parach za pomocą entrySet()
for (final Map.Entry<Integer, String> ageToName : ageToNames.entrySet()) {
  System.out.println("Value for key  " + ageToName.getKey()
    + " is " + ageToName.getValue());
  /* wynikiem zawsze będą poniższe 3 linijki, w tej dokładnie kolejności
     (wynika z użycia LinkedHashMap)
     Value for key  20 is Gosia
     Value for key  40 is Kasia
     Value for key  30 is Ania
   */
}
```

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

KLASA ARRAYS

Klasa **Arrays** jest klasą pomocniczą, która pozwala w łatwy sposób wykonywać pewne operacje na tablicach.

- metoda **binarySearch** pozwala na znalezieniu indeksu, na którym znajduje się dany element w posortowanej tablicy

```
int result = Arrays.binarySearch(new int[]{1, 2, 4, 5 ,6}, 5);
System.out.println(result); // 3
```

- metoda **asList** pozwala z dowolnej ilości elementów lub tablicy stworzyć listę

```
List<Integer> ints = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> ints2 = Arrays.asList(5, 2, 7, 4);
```

- w celu porównania dwóch tablic możemy wykorzystać metodę **compare**.

```
int result1 = Arrays.compare(new int[]{1, 2, 3}, new int[]{1, 2, 3});
System.out.println(result1); // 0

int result2 = Arrays.compare(new int[]{1, 2}, new int[]{1, 2, 3});
System.out.println(result2); // -1

int result3 = Arrays.compare(new int[]{3, 1}, new int[]{1, 3});
System.out.println(result3); // 1
```

KLASA ARRAYS

- aby posortować tablicę (lub jej część) wykorzystujemy metodę **sort**, np.:

```
int[] ints = {3, 1, 5, 4, 2};
Arrays.sort(ints);
System.out.println(Arrays.toString(ints)); //[1, 2, 3, 4, 5]
```

- tablicę (lub jej część) możemy skopiować wykorzystując metodę **copyOf**

```
int[] original = new int[]{1, 2, 3, 4};
int[] copiedResult = Arrays.copyOf(original, 3);
System.out.println(Arrays.toString(copiedResult)); // [1, 2, 3]
```

- metoda **equals** porównuje zawartość tablicy

```
boolean result = Arrays.equals(new int[]{3, 1}, new int[]{1, 3});
System.out.println(result); // false
```

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

KLASA COLLECTIONS

Klasa **Collections**, podobnie jak klasa Arrays, **daje dostęp zestawu metod statycznych, które możemy wykorzystywać do wykonywania operacji na kolekcjach**.

- w celu uzyskania pustych (**niemutowalnych**) kolekcji wykorzystujemy metody, których nazwy zaczynają się od słowa **empty**, np.:

```
List<String> list = Collections.emptyList();
Map<String, Integer> map = Collections.emptyMap();
Set<Object> set = Collections.emptySet();

list.add("2"); // UnsupportedOperationException
```

- metoda **replaceAll** pozwala zastąpić wszystkie wystąpienia danego elementu w kolekcji wartością zastępczą.

```
List<Integer> ints = new ArrayList<>();
ints.add(1);
ints.add(2);
ints.add(2);
ints.add(3);
Collections.replaceAll(ints, 2, 4);
ints.forEach(System.out::println); // 1 4 4 3
```

KLASA COLLECTIONS

- operację sortowania na kolekcji (**mutowalnej**) możemy wykonać przy pomocy metody **sort**, opcjonalnie wskazując odpowiedni **Comparator**, np.:

```
List<String> words = new LinkedList<>();
words.add("hi");
words.add("welcome");
words.add("hello!");
Collections.sort(words, Collections.reverseOrder());

System.out.println(words); // [welcome, hi, hello!]
```

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

INTERFEJSY FUNKCYJNE

Java jest językiem obiektowym, który opiera się na tworzeniu obiektów i komunikacji między nimi. W tym dziale poznamy **paradygmat programowania funkcyjnego**. Programowanie funkcyjne **opiera się jedynie na funkcjach**. Główny program jest funkcją, której podajemy argumenty, a w odpowiedzi otrzymujemy wyznaczony rezultat. Główna funkcja programu składa się tylko i wyłącznie z innych funkcji, które z kolei agregują inne.

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

INTERFEJSY FUNKCYJNE

Wiele interfejsów w Javie składa się tylko i wyłącznie z jednej abstrakcyjnej metody, np. Runnable, Callable, Comparator. Interfejsy takie nazywane są **Single Abstract Method**. W Javie 8 wprowadzono nazwę **FunctionalInterface** dla tego typu komponentów. Zdefiniowany na slajdzie interfejs Action jest interfejsem funkcyjnym.

```
1 public interface Action {
2     void execute(int x, int y);
3 }
```

INTERFEJSY FUNKCYJNE

Poniższy interfejs nie może zostać zakwalifikowany jako interfejs typu SAM, ponieważ posiada **dwie metody abstrakcyjne**.

```
public interface Presenter {
    void present(String text);
    void present(String text, int size);
}
```

@FUNCTIONALINTERFACE

W celu łatwiejszej identyfikacji interfejsów funkcyjnych, w Javie wprowadzona została adnotacja **@FunctionalInterface**, która informuje programistę, że wskazany interfejs **ma być z założenia interfejsem funkcyjnym**. Adnotacja ta powinna zostać umieszczona nad definicją interfejsu. Próba umieszczenia takiej informacji nad interfejsem, który ma np. zero lub dwie metody abstrakcyjne, **skończy się błędem kompilacji**.

UWAGA: Interfejs funkcyjny **NIE musi być oznaczony adnotacją @FunctionalInterface.**

```
@FunctionalInterface
public interface Executor {
    void executor(int x);
}
```

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

DEFAULT

W Javie 8 został wprowadzony mechanizm **metod domyślnych**, określony za pomocą słowa kluczowego **default**. Metoda domyślna jest to metoda interfejsu, która zawiera domyślną implementację (tzn. metoda ta ma ciało). Dzięki tej nowej składni, w ramach interfejsu funkcyjnego, możemy deklarować **więcej niż jedną metodę, z zachowaniem zasady pojedynczej metody abstrakcyjnej**.

Metody typu default mogą być nadpisywane w klasach implementujących, bądź też realizować domyślną funkcjonalność.

```
@FunctionalInterface
public interface Executor {
    void executor(int x);

    default void executor(int x, int y) {
      // I have a body I am a default method
    }
}
```

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

WYRAŻENIE LAMBDA

Dużym problemem klas anonimowych jest to, że nawet **jeśli implementujemy prosty interfejs z jedną metodą, to forma zapisu potrafi być nieczytelna**. Przykładem takiej sytuacji jest przekazanie funkcji do innej metody, np. do obsługi kliknięcia przycisku.

```
Thread thread = new Thread(new Runnable() {
  @Override
  public void run() {
    System.out.println("
        Implementacja interfejsu Runnable jako implementacja klasy anonimowej!");
  }
});
thread.start();
```

WYRAŻENIA LAMBDA

W Javie 8 zostały wprowadzone **wyrażenia lambda, które pozwalają traktować klasę anonimową jak zwykłą funkcję**, znacząco skracając przy tym składnię samego zapisu.

```
Thread thread = new Thread(() -> {
  System.out.println("Runnable example using lambda!");
});
thread.start();
```

Jak widać wyrażenie lambda **zwalnia nas z pisania słowa kluczowego new** i **wykorzystywania adnotacji Override**.

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

SKŁADNIA WYRAŻENIA LAMBDA

Wyrażenie lambda składa się z trzech części:

- listy argumentów, która:
  - musi znaleźć się **w nawiasie, jeżeli ilość argumentów jest różna od 1**
  - **może, ale nie musi**, podawać **typy argumentów**
- operatora **->**, który jest używany po liście argumentów
- ciała implementowanej metody, które znajduje się **za operatorem ->**
  - jeżeli ciało składa się z **jednego wyrażenia** to:
    - ciało to **nie musi** być wewnątrz klamer, tzn. **{** i **}**
    - w przypadku, gdy to wyrażenie **zwraca obiekt**, możemy **pominąć słowo kluczowe return**

SKŁADNIA WYRAŻENIA LAMBDA

- jeżeli ciało składa się z **wielu wyrażeń**, to:
  - ciało to musi znaleźć się **wewnątrz klamer**, tzn. { i }.

UWAGA: jeżeli zadaniem lambdy jest **wyrzucenie wyjątku**, to musi to zostać zrobione **wewnątrz klamer**, pomimo tego, że jest to pojedyncze wyrażenie.

UWAGA: **nazwy argumentów mogą różnić się od tych zdefiniowanych w interfejsie**.

UWAGA: definiując lambdę zupełnie **nie interesuje nas nazwa metody abstrakcyjnej w interfejsie**.

LAMBDA – PRZYKŁAD 1

```
public interface Action {
    String execute(int x, int y);
}
```

```
Action action = (int x, int y) -> {
    return x + "-" + y;
};
```

```
Action action = (x, y) -> x + "-" + y;
```

LAMBDA – PRZYKŁAD 2

```
@FunctionalInterface
public interface Runnable {
  public abstract void run();
}
```

```
Runnable runnableExample = () -> {
  System.out.println("Hello from runnable");
  System.out.println("{ and } cannot be omitted");
};
```

LAMBDA – PRZYKŁAD 3

```
@FunctionalInterface
public interface FruitEater<T> {
    void consume(T t);
}
```

```
FruitEater<String> fruitEater = fruit ->
    System.out.println(String.format("eating %s... omnomnom", fruit);
```

REFERENCJE METOD

Niektóre proste wyrażenia lambda, **możemy zapisać za pomocą referencji metody**. Zamiast pisać wywołanie metody, możemy podać jedynie jej nazwę. W takim przypadku **nazwę klasy i metody musimy oddzielić od siebie znakami '::'**.

Z referencji do metody możemy korzystać, gdy **lambda posiada jeden argument oraz jest spełniony jeden z warunków**:

- argument wejściowy lambdy jest argumentem metody z pewnej klasy
- na argumencie wejściowym wywoływana jest bezargumentowa metoda.

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

REFERENCJE METOD

```
// wykorzystanie lambdy
Consumer<String > consumerExample = someString -> System.out.println(someString);
// identyczny zapis, jak w linijce powyżej, użycie referencji
Consumer<String > consumerExampleReference = System.out::println;

// wykorzystanie lambdy w metodzie map
List.of("someString").stream().map(str -> str.toUpperCase());
// wykorzystanie referencji do metody, zapis równoważny w linijce kodu wyżej
List.of("someString").stream().map(String::toUpperCase);
```

INTERFEJSY FUNKCYJNE - PODSUMOWANIE

```
6:   ____<List> ex1 = x -> "".equals(x.get(0));
7:   ____<Long> ex2 = (Long l) -> System.out.println(l);
8:   ____ <String, String> ex3 = (s1, s2) -> false;
```

```
6:   Function<List<String>> ex1 = x -> x.get(0); // DOES NOT COMPILE
7:   UnaryOperator<Long> ex2 = (Long l) -> 3.14; // DOES NOT COMIPLE
8:   Predicate ex4 = String::isEmpty;  // DOES NOT COMPILE
```

OPTIONAL

Optional, znajdujący się w pakiecie **java.util** jest obiektem, który **opakowuje obiekt docelowy**, tzn., jest pewnego rodzaju pudełkiem, które taki obiekt może zawierać lub nie.

Klasa Optional daje nam dostęp do wielu metod:

- **of**
- **ofNullable**
- **isPresent**
- **ifPresent**
- **orElse**
- **orElseGet**

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

OPTIONAL - KREACJA

Za tworzenie obiektów odpowiedzialna jest metoda statyczna **of** lub **ofNullable**. Różnica między pierwszą, a drugą metodą polega m.in. na tym, że pierwsza z metod **nie pozwala na przyjmowanie wartości typu null**.

```
public class CreatingOptionals {
  public static void main(String[] args) {
    // stworzenie pudełka z obiektem typu String
    final Optional<String> stringOptional = Optional.of("This is Java course!!!");

    String value = null;
    if ((Integer.parseInt(args[0]) % 2 == 0)) {
      value = "I am even";
    }
    // wykorzystanie metody ofNullable, ponieważ value może być nullem.
    // W takim przypadku Optional będzie pustym pudełkiem
    final Optional<String> optionalThatCanBeEmpty = Optional.ofNullable(value);
  }
}
```

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

OPTIONAL – SPRAWDZANIE WARTOŚCI

Klasa **Optional** udostępnia metody, które pozwalają na **warunkowe wykonanie operacji**, jeśli faktycznie w jego wnętrzu znajduje się referencja do obiektu.

- metoda **isPresent** zwraca wartość true/false w zależności od tego, czy Optional przechowuje jakiś obiekt
- metoda **ifPresent** przyjmuje argument w postaci interfejsu funkcyjnego Consumer. Consumer ten **wywoła się tylko i wyłącznie w przypadku, gdy Optional przechowuje referencję do obiektu**.

```java
public class OptionalsPresenceExample {
  public static void main(String[] args) {
    final Optional<String> optional = getStringForEvenNumber(3);
    if (optional.isPresent()) {
      System.out.println("I am optional with a value, I am non empty box");
    } else if (optional.isEmpty()) { // warunek zawsze prawdziwy w tym momencie
      System.out.println("I am an empty optional");
    }

    // wypisanie wartości w pudełku na ekranie, tylko jeżeli jest dostępna
    optional.ifPresent(System.out::println);
  }

  private static Optional<String> getStringForEvenNumber(final int number) {
    if (number % 2 == 0) {
      return Optional.of("even");
    }
    return Optional.empty();
  }
}
```

POBIERANIE WARTOŚCI

Klasa Optional udostępnia kilka metod umożliwiających pobieranie wartości:

- metoda **get** zwraca wartość przechowywaną w obiekcie, bądź też **w przypadku nulla wyrzuca wyjątek**: **NoSuchElementException**
- metoda **orElse** zwraca wartość przechowywaną w Optionalu **lub wartość wskazaną jako argument metody w przypadku pustego Optionala**
- metoda **orElseGet** zwraca wartość przechowywaną w Optionalu **lub wartość wskazaną przez Supplier**, który jest argumentem wejściowym.

```
public class OptionalOrElseExample {
  public static void main(String[] args) {
    String object = null;
    String name = Optional.ofNullable(object).orElse("john");
    System.out.println(name); // na ekranie zostanie wypisana wartość john
  }
}

public class OptionalOrElseGetExample {
  public static void main(String[] args) {
    String object = null;
    String name = Optional.ofNullable(object).orElseGet(() -> "john");
    System.out.println(name); // na ekranie ponownie zostanie wyświetlony String john
  }
}
```

POBIERANIE WARTOŚCI

UWAGA: Chcąc **wykorzystać metodę get**, powinniśmy najpierw **sprawdzić dostępność obiektu za pomocą metody isPresent**.

UWAGA: Metoda **orElseGet w przeciwieństwie do orElse** wyliczy wartość zastępczą **tylko, gdy Optional jest pusty**. Oznacza to, że wydajność metody orElseGet jest lepsza, niż metody orElse.

POBIERANIE WARTOŚCI

Method	When Optional Is Empty	When Optional Contains a Value
`get()`	Throws an exception	Returns value
`ifPresent(Consumer c)`	Does nothing	Calls `Consumer c` with value
`isPresent()`	Returns `false`	Returns `true`
`orElse(T other)`	Returns `other` parameter	Returns value
`orElseGet(Supplier s)`	Returns result of calling `Supplier`	Returns value
`orElseThrow(Supplier s)`	Throws exception created by calling `Supplier`	Returns value

JAVA STREAM API

Klasy dające dostęp do interfejsu **Stream** z pakietu **java.util.stream** umożliwiają funkcyjny sposób przetwarzania danych. Strumienie (tzw. **streamy**) **reprezentują** sekwencyjny zestaw elementów i pozwalają na **wykonywanie** różnych **operacji na tych elementach**.

STRUMIENIE A KOLEKCJE

Strumienie są pewnego **rodzaju reprezentacją kolekcji**, jednak są między nimi pewne różnice:

- **Strumień nie jest strukturą danych**, która przechowuje elementy. Przekazuje on tylko referencje elementów ze źródła, którym może być np.: struktura danych, tablica, kolekcja, metoda generatora
- Operacje na strumieniu **zwracają wynik**, ale **nie modyfikują źródła** strumienia, np. **filtrowanie** strumienia uzyskanego z kolekcji powoduje utworzenie nowego strumienia **bez przefiltrowanych elementów**, natomiast **żaden element nie jest usuwany z oryginalnej kolekcji**.

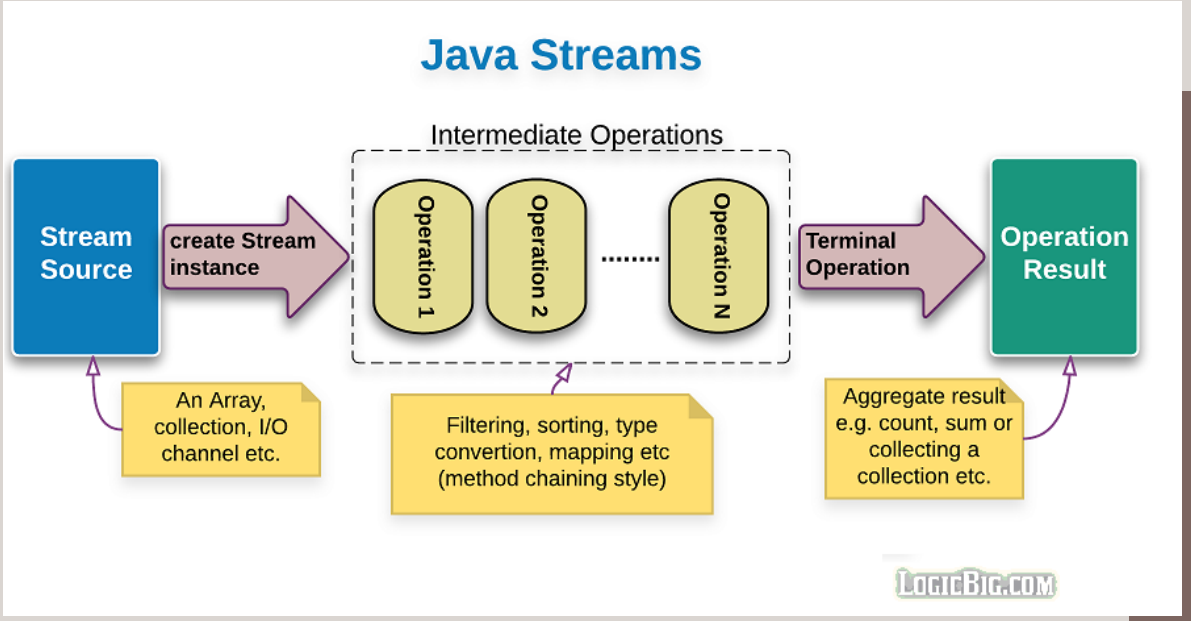
Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

STRUMIENIE A KOLEKCJE

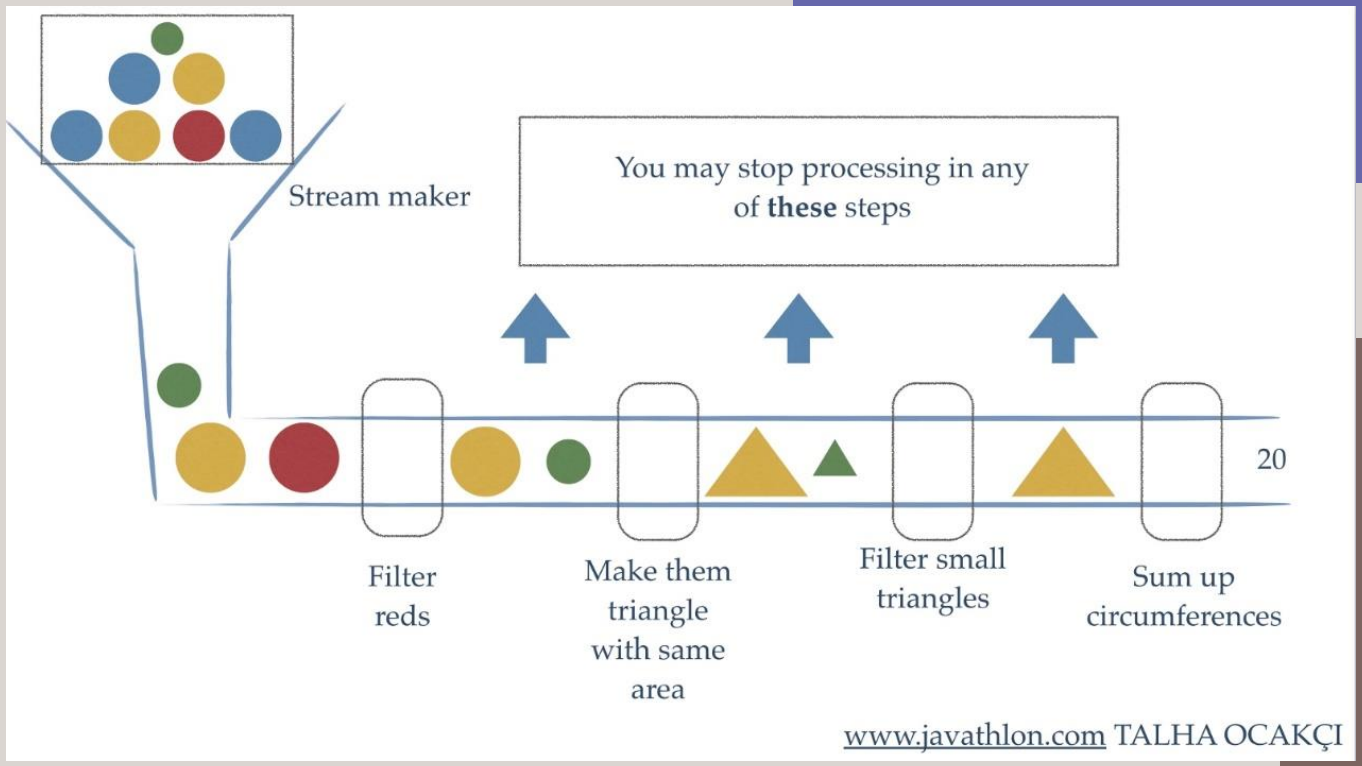
- **Kolekcje mają skończony rozmiar**, strumienie **nie muszą posiadać skończonego rozmiaru**. Operacje takie jak **limit(n)** lub **findFirst()** pozwalają na zakończenie obliczeń w strumieniach w skończonym czasie.
- Elementy w strumieniach **odwiedzane są tylko raz podczas jego trwania**. Podobnie jak Iterator, **niezbędne jest wygenerowanie nowego strumienia**, aby ponownie odwiedzić te same elementy źródła, np. kolekcji.

STRUMIENIE A KOLEKCJE

- **Strumienie są przetwarzane w sposób leniwy** (lazy), to znaczy żadne z metod pośrednich nie są wywoływane, **aż do momentu wywołania jednej z metod kończących**.



STRUMIENIE A KOLEKCJE - PRZYKŁAD



STRUMIENIE - KREACJA

W Javie istnieje wiele sposobów otrzymywania strumieni na podstawie wskazanych kolekcji, tablic, obiektów:

- metoda **stream()** zwraca strumień dla klas dostępnych w Collection API
- metoda **Arrays.stream(Object[])** umożliwia tworzenie strumieni na podstawie tablic
- metoda statyczna **Stream.of(T ...values)** umożliwia tworzenie strumieni na podstawie tablic, obiektów
- metod statyczna **Stream.generate()** tworzy Stream elementów na podstawie wejściowego Suppliera
- strumienie plików mogą być zwracane na podstawie klasy Files.

STRUMIENIE - KREACJA

```
Stream<Integer> streamOfInts = Arrays.asList(1, 2, 3).stream();
Stream<String> streamOfStrings = Set.of("one", "two", "three").stream();
Stream<Map.Entry<String, Integer>> stream = Map.of("someKeyA", 1, "someKeyB", 2).entrySet().stream();
IntStream arraysStream = Arrays.stream(new int[]{1, 2, 3});
Stream<Double> ofStream = Stream.of(1.1, 2.2, 3.3);
Stream<Integer> generateStream = Stream.generate(() -> new Random().nextInt());
Stream<String> fileLinesStream = Files.lines(Path.of("/tmp/1.txt"));
```

```
1: Stream<String> empty = Stream.empty();    // count = 0
2: Stream<Integer> singleElement = Stream.of(1);    // count = 1
3: Stream<Integer> fromArray = Stream.of(1, 2, 3);    // count = 2
```

```
4: List<String> list = Arrays.asList("a", "b", "c");
5: Stream<String> fromList = list.stream();
6: Stream<String> fromListParallel = list.parallelStream();
```

```
7: Stream<Double> randoms = Stream.generate(Math::random);
8: Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);
```

OPERACJE NA STRUMIENIACH

Operacje na strumieniach dzielą się na dwa rodzaje - **pośrednie** oraz **kończące**. Operacje pośrednie zawsze zwracają nowy strumień, są one też przetwarzane w sposób lazy (leniwy). Zaliczamy do nich m.in.:

- **filter**
- **map**
- **flatMap**
- **peek**
- **distinct**
- **sorted**
- **limit**

OPERACJE NA STRUMIENIACH

Operacje kończące są operacjami, które zwracają końcowy wynik. Ich wywołanie powoduje **wykonanie się wszystkich poprzedzających funkcji pośrednich**. Do funkcji kończących zaliczamy:

- **toArray**
- **collect**
- **count**
- **reduce**
- **forEach**, **forEachOrdered**
- **min**, **max**
- **anyMatch**, **allMatch**, **noneMatch**
- **findAny**, **findFirst**

OPERACJE NA STRUMIENIACH

Scenario	For Intermediate Operations?	For Terminal Operations?
Required part of a useful pipeline?	No	Yes
Can exist multiple times in a pipeline?	Yes	No
Return type is a stream type?	Yes	No
Executed upon method call?	No	Yes
Stream valid after call?	Yes	No

OPERACJE POŚREDNIE - MAP

Metoda **map** oczekuje na wejście obiektu **Function<T, R>**. Jej zadaniem jest **przekonwertowanie elementu** strumienia na nowym element, który dodatkowo może mieć inny typ.

```
// stworzenie streamu i przetworzenie elementów wejściowych typu Integer
// na wartość trzy razy większą o typie Double
List.of(1, 2, 3).stream()
    .map(streamElem -> streamElem * 3.0);
```

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
s.map(String::length)
```

OPERACJE POŚREDNIE -FILTER

Operacja **filter** umożliwia **wyrzucenie ze strumienia tych elementów**, które nie spełniają pewnego predykatu, który jest argumentem wejściowym metody. Przykład poniżej pokazuje w jaki sposób usunąć liczby nieparzyste ze strumienia liczb.

```
final int[] idx = { 0 };
Stream.generate(() -> idx[0]++)
    .limit(10)
    .filter(elem -> elem % 2 == 0); //w strumieniu zostają: 0, 2, 4, 6, 8
```

OPERACJE POŚREDNIE -DISTINCT

Operacja **distinct** umożliwia stworzenie strumienia, w którym **wszystkie elementy są unikalne**, tzn. pozbywamy się powtórzeń, np.:

```
Arrays.asList(3, 6, 6, 20, 21, 21).stream()
    .distinct(); // w streamie zostaną elementy: 3, 6, 20, 21
```

OPERACJE TERMINALNE -FOREACH

Operacja **forEach** reprezentuje **funkcyjną wersję pętli for**. Funkcja ta wywołuje dowolną operację zaimplementowaną za pomocą interfejsu funkcyjnego **Consumer<T> na każdym elemencie strumienia**.

```
List.of(1, 2, 3, 4, 5).stream()
    .forEach(System.out::println);
```

```
void forEach(Consumer<? super T> action)
```

```
Stream<String> s = Stream.of("Monkey", "Gorilla", "Bonobo");
s.forEach(System.out::print);    // MonkeyGorillaBonobo
```

```
Stream s = Stream.of(1);
for (Integer i: s) {} // DOES NOT COMPILE
```

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

OPERACJE TERMINALNE - COLLECT

Metoda **collect** pozwala **zebrać elementy strmienia do pewnego obiektu docelowego**. W celu zebrania elementów musimy wykorzystać interfejs **Collector**, który **nie jest interfejsem funkcyjnym**. Z pomocą przychodzi klasa **Collectors**, która zawiera statyczne metody odpowiedzialne za kumulację elementów strumienia do wskazanej struktury, np. **List**, bądź **Set**.

```
final List<Integer> listCreatedFromCollectMethod
    = Stream.generate(() -> new Random().nextInt())
        .limit(10)
        .distinct()
        .filter(elem -> Math.abs(elem) < 1000)
        .collect(Collectors.toUnmodifiableList());

final String sentence = Stream.of("This", "will", "be", "single",
        "sentence", "but", "without", "some", "words")
    .filter(word -> word.length() > 2)
    .collect(Collectors.joining(" "));
System.out.println(sentence);
// outputem będzie "This will single sentence but without some words"


final Map<String, String> wordToUppercasedVersion
    = Stream.of("Hello", "from", "Stream", "api")
        .collect(Collectors.toMap(Function.identity(), String::toUpperCase));
```

Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska
Europejski Fundusz Społeczny

DZIĘKUJĘ ZA UWAGĘ