

STRUMIENIE I WYRAŻENIA LAMBDA

- Strumienie jako sposób przetwarzania zbiorów danych
- Interfejsy funkcyjne
- Wyrażenia lambda

Wykład częściowo oparty na materiałach A. Jaskot

PRZETWARZANIE ZBIORÓW DANYCH

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() { return name; }  
  
    public int getAge() { return age; }  
}
```

```
List<Person> people = Arrays.asList(  
    new Person("Xxx", 16),  
    new Person("Yyy", 30),  
    new Person("Zzz", 22),  
    new Person("Www", 18),  
    new Person("Vvv", 21)  
);
```

Zadanie:
Policz średnią wieku wszystkich osób starszych
niż 18 lat

PRZETWARZANIE ZBIORÓW DANYCH

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() { return name; }  
  
    public int getAge() { return age; }  
}
```

```
List<Person> people = Arrays.asList(  
    new Person("Xxx", 16),  
    new Person("Yyy", 30),  
    new Person("Zzz", 22),  
    new Person("Www", 18),  
    new Person("Vvv", 21)  
);
```

Zadanie:
Policz średnią wieku wszystkich osób starszych
niż 18 lat

```
int sum = 0;  
int count = 0;  
for (Person person : people) {  
    if (person.getAge() > 18) {  
        sum += person.getAge();  
        count++;  
    }  
}  
  
double averageAge = (count > 0) ?  
    (double) sum / count : 0.0;
```

PRZETWARZANIE ZBIORÓW DANYCH

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() { return name; }  
  
    public int getAge() { return age; }  
}
```

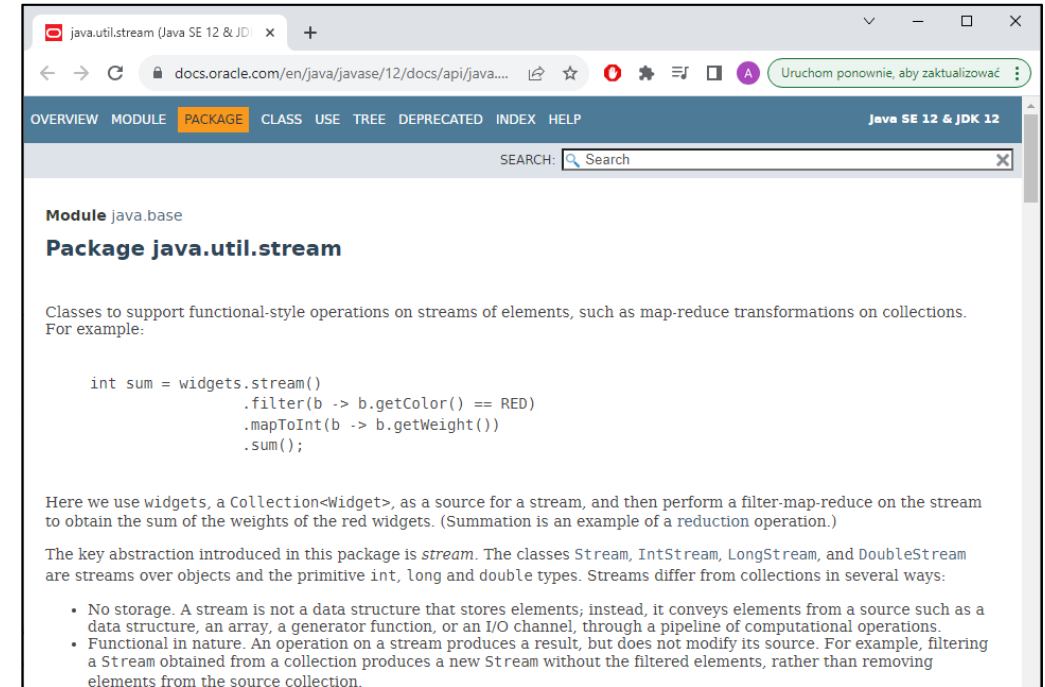
```
List<Person> people = Arrays.asList(  
    new Person("Xxx", 16),  
    new Person("Yyy", 30),  
    new Person("Zzz", 22),  
    new Person("Www", 18),  
    new Person("Vvv", 21)  
);
```

Zadanie:
Policz średnią wieku wszystkich osób starszych
niż 18 lat

```
double averageAge = people.stream()  
    .filter(person -> person.getAge() > 18)  
    .mapToDouble(Person::getAge)  
    .average()  
    .orElse(0.0);
```

STRUMIENIE (ang. Streams)


- dostępne od Javy 8
- są sposobem przetwarzania zbiorów danych pozwalającym na uniknięcie nadmiernego wykorzystania pętli i instrukcji warunkowych
- kładą nacisk na opis operacji do wykonania na danych w formie funkcji



<https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/stream/package-summary.html>

STREAM PIPELINE

```
List<String> words = Arrays.asList  
    ("apple", "banana", "grape", "orange");  
  
words.stream()  
    .filter(word -> !word.startsWith("a"))  
    .map(String::toUpperCase)  
    .forEach(System.out::println);
```



ŹRÓDŁO DANYCH


- źródłem danych może być np. tablica, kolekcja, generator, kanał I/O itd.
- do przekształcenia kolekcji na strumień służy zwykle metoda *stream()*

```
List<String> words = Arrays.asList  
    ("apple", "banana", "grape", "orange");  
Stream<String> wordsStream = words.stream();
```

- innym sposobem generowania strumieni są statyczne metody interfejsu Stream, np. *iterate()*

STREAM PIPELINE

```
List<String> words = Arrays.asList  
    ("apple", "banana", "grape", "orange");  
  
words.stream()  
    .filter(word -> !word.startsWith("a"))  
    .map(String::toUpperCase)  
    .forEach(System.out::println);
```



OPERACJE POŚREDNIE (ang. Intermediate)


- operacje przekształcające strumień w inny strumień, przede wszystkim filtracja, mapowanie, sortowanie, pomijanie części elementów

Najważniejsze metody:

filter(), filterNot(), map(), sorted(), limit(), skip()

STREAM PIPELINE

```
List<String> words = Arrays.asList  
    ("apple", "banana", "grape", "orange");  
  
words.stream()  
    .filter(word -> !word.startsWith("a"))  
    .map(String::toUpperCase)  
    .forEach(System.out::println);
```



OPERACJA KOŃCZĄCA (ang. Terminal)

- operacja zwracająca wynik (lub tzw. efekt uboczny), np. zliczanie elementów, wykonanie operacji na każdym elemencie

Najważniejsze metody:

forEach(), *count()*, *min()*, *max()*, *collect()*,
findAny(), *findFirst()*, *anyMatch()*, *allMatch()*

STREAM PIPELINE

- strumień jest sposobem przetwarzania danych, nie ich przechowywania
- strumień nie modyfikuje źródła danych (operacje pośrednie zwracają nowe strumienie)
- po wykonaniu operacji kończącej strumień jest uznawany za “zużyty” (kolejne przetworzenie danych wymaga stworzenia nowego strumienia)

OPERACJE POŚREDNIE I KOŃCZĄCE

<code>Stream<T></code>	<code>distinct()</code> Returns a stream consisting of the distinct elements (according to <code>Object.equals(Object)</code>) of this stream.
<code>Stream<T></code>	<code>limit(long maxSize)</code> Returns a stream consisting of the elements of this stream, truncated to be no longer than <code>maxSize</code> in length.
<code>Stream<T></code>	<code>skip(long n)</code> Returns a stream consisting of the remaining elements of this stream after discarding the first <code>n</code> elements of the stream.
<code>Stream<T></code>	<code>sorted()</code> Returns a stream consisting of the elements of this stream, sorted according to natural order.

<code>long</code>	<code>count()</code> Returns the count of elements in this stream.
<code>Optional<T></code>	<code>findAny()</code> Returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.
<code>Optional<T></code>	<code>findFirst()</code> Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty.

<code>Stream<T></code>	<code>filter(Predicate<? super T> predicate)</code> Returns a stream consisting of the elements of this stream that match the given predicate.
<code><R> Stream<R></code>	<code>map(Function<? super T, ? extends R> mapper)</code> Returns a stream consisting of the results of applying the given function to the elements of this stream.
<code>DoubleStream</code>	<code>mapToDouble(ToDoubleFunction<? super T> mapper)</code> Returns a <code>DoubleStream</code> consisting of the results of applying the given function to the elements of this stream.
<code>IntStream</code>	<code>mapToInt(ToIntFunction<? super T> mapper)</code> Returns an <code>IntStream</code> consisting of the results of applying the given function to the elements of this stream.

<code>boolean</code>	<code>allMatch(Predicate<? super T> predicate)</code> Returns whether all elements of this stream match the provided predicate.
<code>boolean</code>	<code>anyMatch(Predicate<? super T> predicate)</code> Returns whether any elements of this stream match the provided predicate.
<code>void</code>	<code>forEach(Consumer<? super T> action)</code> Performs an action for each element of this stream.
<code>Optional<T></code>	<code>max(Comparator<? super T> comparator)</code> Returns the maximum element of this stream according to the provided <code>Comparator</code> .
<code>Optional<T></code>	<code>min(Comparator<? super T> comparator)</code> Returns the minimum element of this stream according to the provided <code>Comparator</code> .

INTERFEJSY FUKCYJNE (ang. Functional interfaces)

- interfejsy posiadające tylko jedną metodę abstrakcyjną (SAM, z ang. Single Abstract Method)
- zwykle oznaczone adnotacją *@FunctionalInterface*
- najważniejsze interfejsy funkcyjne pakietu *java.util.function*:
 - *Predicate<T>*
 - *Function<T, R>*
 - *Consumer<T>*
 - *Supplier<T>*
 - *Comparator<T>*

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

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

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

INTERFEJSY FUKCYJNE (ang. Functional interfaces)

- metody służące do przetwarzania strumieni często wymagają przekazania implementacji danego interfejsu

```
List<String> words = Arrays.asList  
    ("apple", "banana", "grape", "orange");  
  
words.stream()  
    .filter(word -> !word.startsWith("a"))  
    .forEach(System.out::println);
```

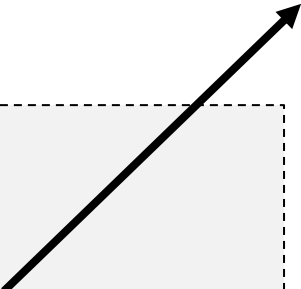
```
@FunctionalInterface  
public interface Predicate<T> {  
    boolean test(T t);  
}
```

```
words.stream()  
    .filter(  
        new Predicate<String>() {  
            @Override  
            public boolean test(String word) {  
                return !word.startsWith("a");  
            }  
        }  
    )  
    .forEach(System.out::println);
```

INTERFEJSY FUKCYJNE (ang. Functional interfaces)

- metody służące do przetwarzania strumieni często wymagają przekazania implementacji danego interfejsu

```
List<String> words = Arrays.asList  
    ("apple", "banana", "grape", "orange");  
  
words.stream()  
    .filter(word -> !word.startsWith("a"))  
    .forEach(System.out::println);
```



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

```
class DoesNotStartWithA implements Predicate<String> {  
    @Override  
    public boolean test(String word) {  
        return !word.startsWith("a");  
    }  
}
```

```
Predicate<String> DoesNotStartWithAPredicate  
    = new DoesNotStartWithA();  
  
words.stream()  
    .filter(DoesNotStartWithAPredicate)  
    .forEach(System.out::println);
```

WYRAŻENIA LAMBDA (ang. Lambda expressions)

- uproszczony sposób zapisu implementacji interfejsów funkcyjnych
- służą przede wszystkim do przekazywania jako argumenty metod (rzadziej przypisywania do zmiennej)
- składają się z dwóch głównych części: listy parametrów oraz ciała wyrażenia (operacji do wykonania) oddzielonych znakiem strzałki ->

```
() -> System.out.println("Lambda without parameters");
```

```
message -> System.out.println("Message: " + message);
```

```
(a, b) -> a + b;
```

```
n -> n % 2 == 0;
```

```
n -> {  
    if (n % 2 == 0) {  
        return "Even";  
    } else {  
        return "Odd";  
    }  
}
```

WYRAŻENIA LAMBDA (ang. Lambda expressions)

```
@FunctionalInterface
public interface Runnable {
    void run();
}
() -> System.out.println("Lambda without parameters");
```

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
message -> System.out.println("Message: " + message);
```

```
@FunctionalInterface
public interface IntPredicate {
    boolean test(int value);
}
n -> n % 2 == 0;
```

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
n -> {
    if (n % 2 == 0) {
        return "Even";
    } else {
        return "Odd";
    }
}
```

```
@FunctionalInterface
public interface BinaryOperator<T> {
    T apply(T t1, T t2);
}
(a, b) -> a + b;
```

WYRAŻENIA LAMBDA (ang. Lambda expressions)

```
people.stream()
    .filter(new Predicate<String>() {
        @Override
        public boolean test(String imie) {
            return imie.startsWith("a");
        }
    })
    .map(new Function<String, Person>() {
        @Override
        public Person apply(String name) {
            return new Person(name);
        }
    })
    .forEach(new Consumer<Person>() {
        @Override
        public void accept(Person person) {
            System.out.println(person);
        }
    });
```

```
people.stream()
    .filter(imie -> {
        return imie.startsWith("a");
    })
    .map(name -> {
        return new Person(name);
    })
    .forEach(person -> {
        System.out.println(person);
    });
```

```
people.stream()
    .filter(imie -> imie.startsWith("a"))
    .map(name -> new Person(name))
    .forEach(person -> System.out.println(person));
```


REFERENCJE DO METOD

```
List<String> words = Arrays.asList  
    ("apple", "banana", "avocado", "grape", "orange");  
  
words.stream()  
    .filter(word -> !word.startsWith("a"))  
    .map(String::toUpperCase)  
    .forEach(System.out::println);
```

```
.map(word -> word.toUpperCase())  
.forEach(word -> System.out.println(word));
```

```
people.stream()  
    .filter(imie -> imie.startsWith("a"))  
    .map(Person::new)  
    .forEach(System.out::println);
```

```
.map(name -> new Person(name))
```

```
double averageAge = people.stream()  
    .mapToDouble(Person::getAge)  
    .average()  
    .orElse(0.0);
```

```
.map(person -> person.getAge())
```

PRZEKSZTAŁCANIE STRUMIENIA W KOLEKCJĘ

```
List<String> names = new ArrayList<>();
names.add("Xxx");
names.add("Yyy");
names.add("Abcd");
names.add("Zzz");
names.add("Aaa");

List<Person> people
    = names.stream()
        .filter(name -> name.startsWith("A"))
        .map(name -> new Person(name))
        .collect(Collectors.toList());

System.out.println(people);
```

```
[Name: Abcd, Name: Aaa]
```

METODA *COLLECT()*

- pozwala na przekształcenie strumienia w kolekcję, np. listę lub zbiór
- jako argument przyjmuje Collector
- podstawowe Collectory są dostępne w klasie Collectors w metodach *toList()* i *toSet()*

METODA FOREACH W KOLEKCJACH

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);  
  
numbers.forEach(number  
    -> System.out.println("Number: " + number));
```

```
Number: 1  
Number: 2  
Number: 3  
Number: 4  
Number: 5
```

METODA *FOREACH()*

- kolekcje implementujące interfejs `Iterable` można przetwarzać też za pomocą metody *forEach()*
- metoda *forEach()* przyjmuje interfejs funkcyjny `Consumer`, reprezentujący akcję do wykonania na każdym elemencie
- metoda *forEach()* daje dużo mniejsze możliwości przetwarzania niż strumień (tylko proste operacje na wszystkich elementach, zwykle drukowanie)