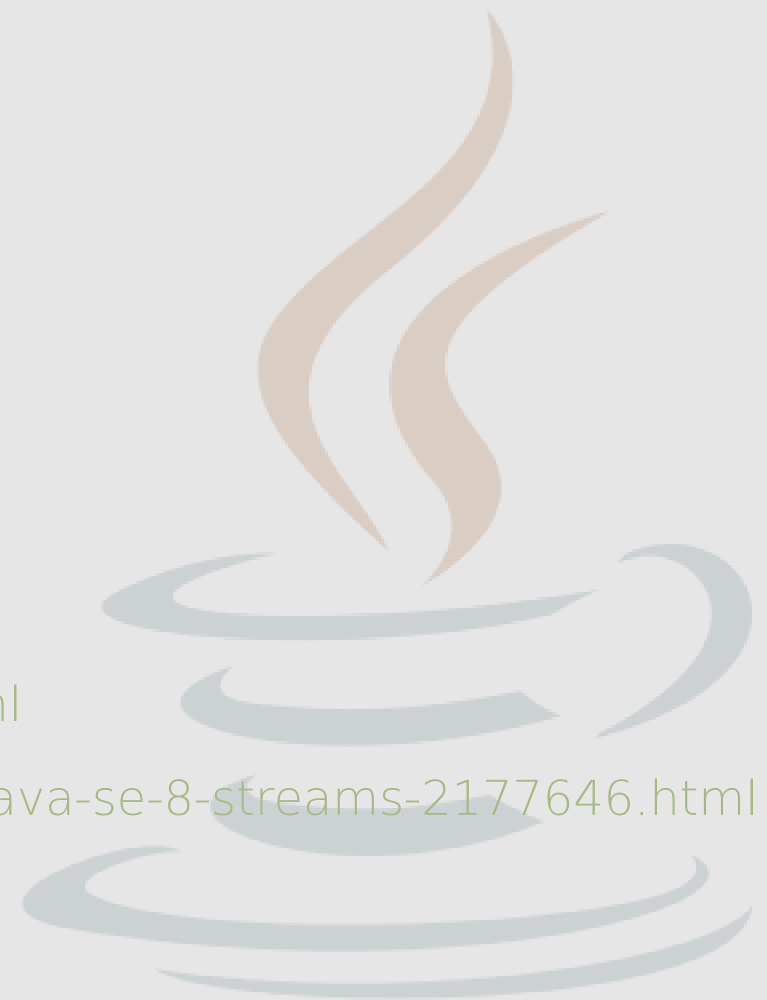
PROGRAMOWANIE FUNKCYJNE

ZAGADNIENIA:

- Funkcje
- Wyrażenia lambda
- Strumienie



MATERIAŁY:

https://flyingbytes.github.io/.../Java8-Part0.html

http://www.oracle.com/technetwork/.../ma14-java-se-8-streams-2177646.html

FUNKCJE

```
...

Thread t = new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello");
    }
});
t.start();
...

Thread t = new Thread(()->System.out.println("Hello"));
t.start();

...

Runnable r = ()->System.out.println("Hello");
Thread t = new Thread(r);
t.start();

...
```

FUNKCJE

Przykłady definicji funkcji:

```
() -> {System.out.println("Hello");}// java.lang.Runnable
x  -> {System.out.println(x);}         // java.util.function.Consumer<T>
() -> 7                                // java.util.function.Supplier<T>
```

```
Runnable r = ()->{System.out.println("Hello");};
Consumer<Integer> cons = x -> {System.out.println(x);};
Supplier<Integer> sup = () -> 7;
```

FUNKCJE

Inne przykłady:

```
Function<Integer, Integer> inc = x -> x+1;
BiFunction<Integer, Integer, Integer> sum = (x,y) -> x+y;
```

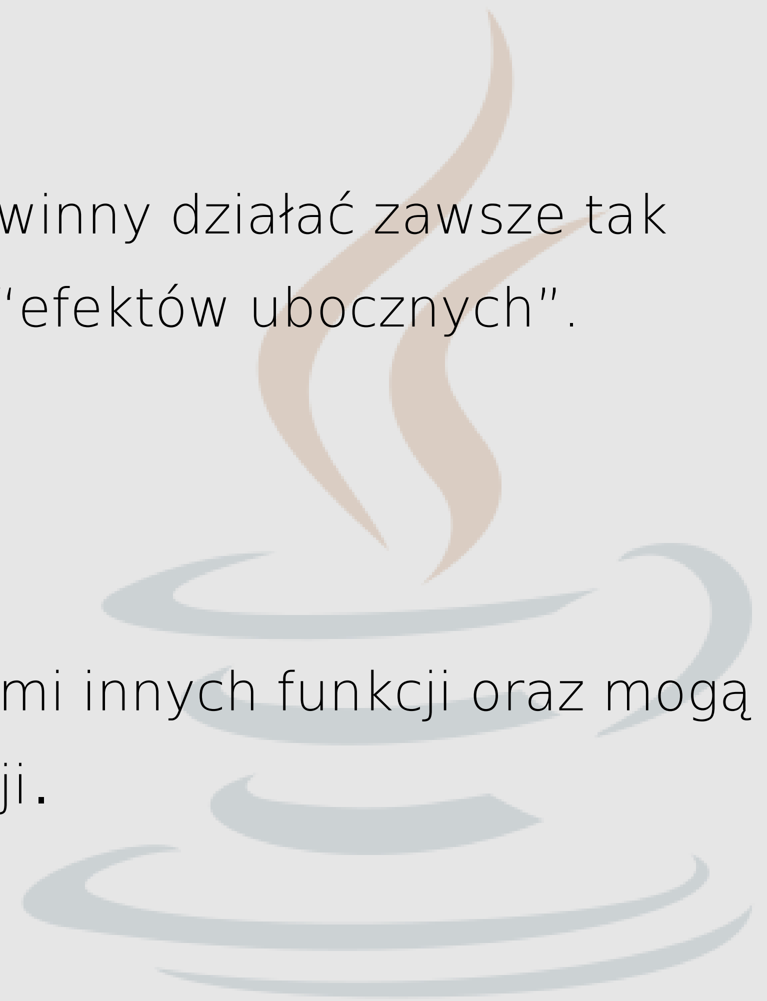
Każdą funkcję wieloargumentową można zapisać jako złożenie funkcji jednoargumentowych:

```
Function<Integer, Function<Integer, Integer>> sum1 = x->y->x+y;
```

Funkcja może być argumentem innej funkcji:

```
BiFunction<Function<Integer, Integer>, Integer, Integer> fx = (f, x) -> f.apply(x);
```

FUNKCJE



W programowaniu funkcyjnym funkcje powinny działać zawsze tak samo i nie powinny powodować żednych "efektów ubocznych".

Podstwowe zasady:

- wszystkie zmienne są stałe (**final**),
- nie ma zmiennych globalnych,
- funkcje mogą być zarówno argumentami innych funkcji oraz mogą być zwracane jako wynik innych funkcji.

java.util.function

```
Function<T, R>
    R apply(T t);
    Function<T, V> andThen(Function<? super R, ? extends V> after);
    Function<V, R> compose(Function<? super V, ? extends T> before);
```

metoda **apply()** zwraca wartość funkcji, natomiast **andThen()** oraz **compose()** zwracają funkcję będącą złożeniem dwóch funkcji.

```
Consumer<T>
    void accept(T t);
    Consumer<T> andThen(Consumer<? super T> after);
```

metoda **accept()** wykonuje funkcję, natomiast **andThen()** zwracaja funkcję będącą złożeniem dwóch funkcji.

```
Supplier<R>
    R get();
```

java.util.function

Inne (wybrane) interfejsy:

```
Predicate<T>
    boolean test(T t);
    Predicate<T> and(Predicate<? super T>, other);
    Predicate<T> or(Predicate<? super T>, other);
    Predicate<T> negate();
```

Predicate to funkcja, której wynik jest typu boolean

```
public interface UnaryOperator<T> extends Function<T,T>
```

UnaryOperator to funkcja, której wynik jest tego samego typu co argument

STRUMIENIE

Strumienie java.util.stream.Stream reprezentuje strumień danych (obiektów) i jest odpowiednikiem kolekcji (java.util.Collection) w programowaniu obiektowym

```
Stream<String> objectStream = Stream.of("Ala", "Ola");


String[] array = {"Ala", "Ola"};
Stream<String> arrayStream = Arrays.stream(array);
```

W interfejsie Collection istnieje metoda stream() lub parallelStream() przekształcająca kolekcję w strumień.

STRUMIENIE

Przetwarzanie danych w strumieniach opiera się na przekształcaniu strumieni I stosowaniu odpowiednich funkcji

```
Stream.of(1, 2, 3)  // tworzymy strumień zawierający 1 2 3
        .map(num -> num * num) // nowy strumień, w którym element
       // zastępujemy jego drugą potęgą (1 4 9)
        .forEach(System.out::println); // dla każdego elementu strumienia
           // wywołujemy funkcję która                                         // wypisuje element na
ekran
```

System.*out*::println – “wskaźnik” do funkcji. Równoważnie:

```
num -> System.out.println(num)
```

STRUMIENIE

```
List<Integer> list1 = Arrays.asList(1, 2, 3);
List<Integer> list2 = Arrays.asList(4, 5, 6);

Stream.of(list1, list2)              // Stream<List<Integer>>
    .flatMap(List::stream)           // Stream<Integer>
    .filter(num -> num % 2 == 0)     // zostają liczby parzyste
    .forEach(System.out::println);   // 2 4 6
```

STRUMIENIE

```
String sentence = Stream.of("Hello", "world")
                        .collect(Collectors.joining(" "));

System.out.println(sentence); // Hello world
```

```
Integer sum = Stream.of(1, 2, 3)
                    .reduce(0, Integer::sum);

    System.out.println(sum); // 6
```

DZIĘKUJĘ ZA UWAGĘ