infoShare ACADEMY

Invest in Pomerania ACADEMY

INVEST IN POMERANIA ACADEMY

Fundusze Europejskie Program Regionalny

Rzeczpospolita Polska

URZĄD MARSZAŁKOWSKI WOJEWÓDZTWA POMORSKIEGO

Unia Europejska Europejski Fundusz Rozwoju Regionalnego

infoShare ACADEMY

Invest in Pomerania ACADEMY

Strumienie i programowanie funkcyjne

Fundusze Europejskie Program Regionalny

Rzeczpospolita Polska

URZĄD MARSZAŁKOWSKI WOJEWÓDZTWA POMORSKIEGO

Unia Europejska Europejski Fundusz Rozwoju Regionalnego

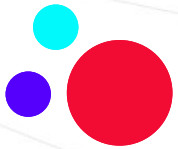
infoShareAcademy.com

HELLO

Tomasz Lisowski

Software developer
Scrum Master
IT trainer

infoShare ACADEMY



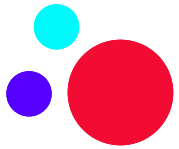
Agenda

- wprowadzenie
- Optional
- interfejs funkcyjny
- wyrażenia lamba
- metoda forEach()
- popularne interfejsy funkcyjne

Ćwiczenie

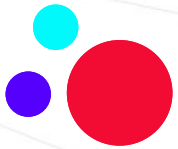
- sklonuj poniższe repozytorium
- zapoznaj się z klasami i strukturą programu

jjdzr11-materialy-Java-Strumienie-i-programowanie-funkcyjne



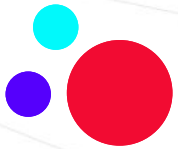
Wprowadzenie

infoShareAcademy.com



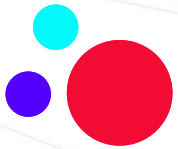
Wprowadzenie

- wersja LTS, wydanie: 03.2014
- wsparcie aż do 03.2025
- **wyrażenia lambda** - programowanie funkcyjnie w Javie
- **interfejsy funkcyjne** - interfejsy z tylko jedną metodą abstrakcyjną
- **Date/Time API** - standaryzacja pracy z czasem i datą
- **strumienie** - wspomagają pracę na kolekcjach
- **Optional** - pomaga unikać NullPointerException



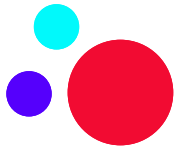
Programowanie funkcyjne

- paradygmat programowania
- programy są kompozycją wielu nałożonych na siebie funkcji
- każda funkcja zwraca wartość (zamiast zmieniać stan obiektu)
- mówimy **“CO”** funkcja ma zrobić, a nie **“JAK”**
- funkcje:
  - mają swój typ
  - mogą być przypisane do zmiennej

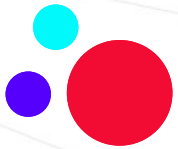


Programowanie funkcyjne

- **zalety**
  - łatwe w testowaniu
  - mniej kodu
  - silna abstrakcja
  - łatwa kompozycja
  - skalowalność
- **wady**
  - większy próg wejścia niż OOP
  - łączenie kodu nie-funkcyjnego z funkcyjnym bywa skomplikowane
  - szybkość

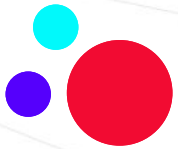


Optional



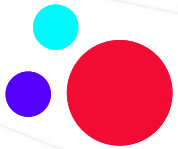
Optional

- kontener na wartości, które mogą być **NULL**
- pomaga unikać wyjątku **NullPointerException**
- gdy wynik metody może być ‘niczym’
- posiada metody obsługi wartości gdy jest “dostępna” lub “niedostępna”
- nie trzeba sprawdzać ***if (obj == null)***



Optional

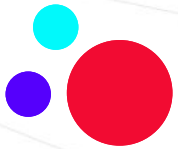
```
String str1 = "isa";
Optional<String> optional1 = Optional.of(str1);
System.out.println(optional1.isPresent()); // true
System.out.println(optional1.get()); // "isa"
System.out.println(optional1.orElse( other: "empty")); // "isa"

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



Optional – tworzenie

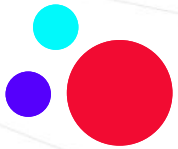
- ***Optional.of(T)*** - kontener na wartość typu T (nie może być NULL)
- ***Optional.ofNullable(T)*** - kontener na wartość typu T (może być NULL)
- ***Optional.empty()*** - pusty kontener

```
Optional.of(findByName("andrzej"));
Optional.ofNullable(findByName("andrzej"));
Optional.empty();
```



Optional – użycie

- ***Optional.get()*** - zwraca wartość (możliwy NoSuchElementException)
- ***Optional.orElse(T)*** - zwraca wartość lub parametr
- ***Optional.isPresent()*** - zwraca *true*, jeśli kontener zawiera wartość
- ***Optional.orElseThrow(Supplier)*** - zwraca wartość lub wyjątek
- ***Optional.ifPresent(Consumer)*** - przekazuje wartość do consumera

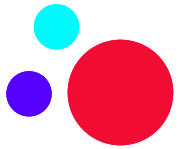


Optional - użycie

```
Optional<String> optional = Optional.of("andrzej");
String value = optional.get();
String orElse = optional.orElse("unkown");
boolean isPresent = optional.isPresent();
String valueOrThrow = optional.orElseThrow(IllegalArgumentException::new);
optional.ifPresent(System.out::println);
```

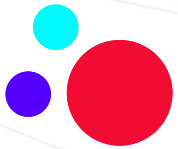
Ćwiczenie 1

- stwórz kilka obiektów typu **Car**
- niech kilka z nich nie ma silnika (pole engine = null)
- dodaj wszystkie obiekty **Car** do listy
- stwórz pętlę, która wypisze wartość mocy silnika dla każdego samochodu
- jeżeli dany obiekt nie ma silnika, to wyświetl komunikat o braku
- skorzystaj z klasy **Optional**
- program nie może rzucić żadnego wyjątku
- nie używaj bloku **try..catch**



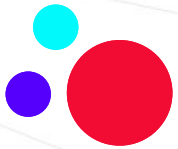
Interfejsy funkcyjne

infoShareAcademy.com



Interfejs funkcyjny

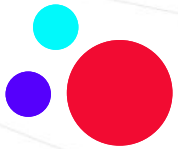
- interfejs, który **posiada tylko jedną metodę abstrakcyjną**
- pozwala na użycie wyrażeń lambda, zamiast jawnych implementacji interfejsu
- może zawierać dowolną liczbę metod domyślnych
- oznaczamy adnotacją **@FuncionalInterface**



Interfejs funkcyjny

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

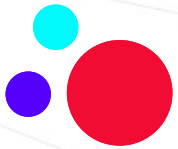
```java
public static void carryOutWork(Task task) {
    task.doWork();
}

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



Interfejs funkcyjny

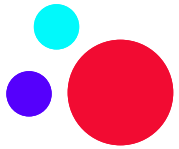
```java
@FunctionalInterface
public interface Task {
    void doWork();
}
```

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

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



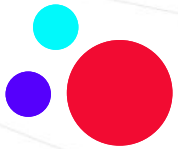
Ćwiczenie 2

- stwórz interfejs
- dodaj w nim jedną, dowolną metodę abstrakcyjną
- oznacz interfejs adnotacją @FunctionalInterface



Wyrażenia lambda

infoShareAcademy.com

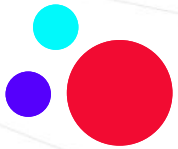


Wyrażenia lambda

- umożliwiają programowanie funkcyjne
- skupienie uwagi na tym CO program robi, a nie JAK
- anonimowa implementacja interfejsu funkcyjnego
- jak metoda, składa się z listy parametrów (o ile występują) i ciała
- operator strzałki “->”
- może być przypisane do zmiennej

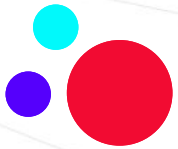
```
() -> wyrażenie
parametr -> wyrażenie
(param1, param2) -> { instrukcja1; instrukcja2; }
```

infoShare ACADEMY



Wyrażenia lambda

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

x -> x * x

() -> “return value”

(Integer a, Integer b) -> System.out.println(a + b)

x -> {
    if (x % 2 == 0) { return true;}
    else { return false;}
}

() -> {}

() -> 1
```



Wyrażenia lambda

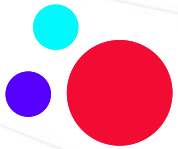
- “metoda”, którą można przypisać do zmiennej
- można ją przekazać jako argument metody
- wyrażenie lambda jest instancją interfejsu funkcyjnego

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

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

Ćwiczenie 3

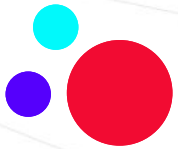
- napisz wyrażenie lamba implementujące interfejs z ćwiczenia 2
- przetestuj działanie
- stwórz kolejną lambdę, realizującą metodę interfejsu w inny sposób

- *przypisz obydwie lambdy do zmiennych
- *stwórz metodę, przyjmującą w parametrze interfejs
- *wywołaj tę metodę dla obydwu wyrażeń



Comparator* – Java 7

```
List<String> names = Arrays.asList("Klaudia",
    "Basia", "Andrzej", "Janusz");


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



Comparator* – Java 8

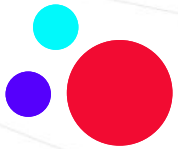
```
List<String> names = Arrays.asList("Klaudia",
    "Basia", "Andrzej", "Janusz");


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);
});
```

Ćwiczenie 4a

- utwórz interfejs **MathOperation** z jedną metodą **calculate**
  - metoda powinna przyjmować parametr - **lista typu Integer**
  - metoda powinna zwracać wartość typu **Integer**
- stwórz klasy **MaxOperation** i **MinOperation**
  - niech obydwie implementują interfejs **MathOperation**
  - niech zwracają odpowiednio wartości **max** i **min** z listy parametrów

Ćwiczenie 4b

- utwórz metodę **printResult**, która przyjmuje 2 parametry:
  - lista liczb typu **Integer**
  - instancja **MathOperation**
- metoda powinna wypisać na ekran wynik danej operacji dla danej listy
- przetestuj program:
  - dla instancji obiektów **MaxOperation**/**MinOperation**
  - z wykorzystaniem lambdy



Metoda forEach()

- Java 8 wprowadziła metodę **forEach()** dla:
  - interfejsu **Iterable** – czyli dla wszystkich kolekcji (wyłączając mapy).
  - interfejsu **Map** – dla uzupełnienia kolekcji również o mapy.
  - interfejsu **Stream** – nowego rozwiązania Javy 8.

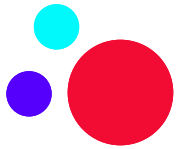
```
values.forEach(n -> System.out.println(n));

values.forEach((k,v)
                    -> System.out.println(k + " : "+ v));
```

Ćwiczenie 5a

- utwórz listę 5 liczb typu Integer
- wywołaj na tej liście metodę forEach():
  - wypisz wszystkie elementy
  - wypisz elementy 2 razy większe niż oryginalne

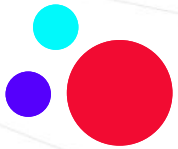
Ćwiczenie 5b

- utwórz listę 5 elementów typu String
- wywołaj na tej liście metodę forEach():
  - wypisz wszystkie elementy
  - wypisz elementy wielkimi literami
  - wypisz tylko pierwszą literę każdego z elementów



Interfejsy funkcyjne v2

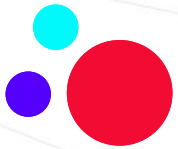
infoShareAcademy.com



Interfejsy funkcyjne

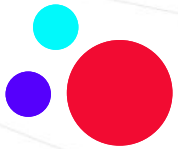
- istnieje zestaw ‘standardowych’ interfejsów funkcyjnych, przygotowanych przez twórców Javy
- w większości przypadków wystarczą nam do pracy
- chociaż zawsze możemy tworzyć własne
- pakiet java.util.function

więcej tutaj: https://docs.oracle.com/



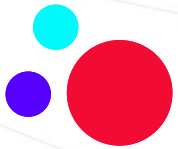
Interfejsy funkcyjne

- *Predicate<T>*
- *Consumer<T>*
- *Function<T, R>*
- *Supplier<T>*
- *UnaryOperator<T>*
- *BinaryOperator<T>*
- *BiPredicate<L, R>*
- *BiConsumer<T, U>*



Interfejsy funkcyjne

	Interfejs		Wersja binarna
Predicate	(x) -> boolean	BiPredicate	(x,y) -> boolean
Consumer	(x) -> void	BiConsumer	(x,y) -> void
Function	(x) -> fun(x)	BiFunction	(x,y) -> fun(x, y)
Supplier	() -> object	-	-



Interfejsy funkcyjne

- ***Predicate<T>*** - test danego warunku, np. filtrowanie

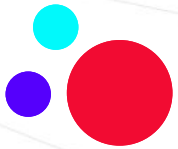
  wyrażenie logiczne obiektu T, zawiera metodę test()

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

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

Ćwiczenie 6a

- napisz własny predykat dla typu Integer
- stwórz listę liczb
- wypisz wynik predykatu dla każdego obiektu z listy



Interfejsy funkcyjne

- ***Consumer<T>*** - wykonanie akcji, np. iteracja przez kolekcje

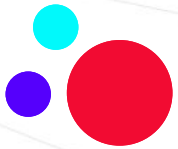
  konsument obiektu T, zawiera metodę accept(), nie zwraca wartości

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

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

Ćwiczenie 6b

- napisz własny consumer dla typu Car
- stwórz listę obiektów typu Car
- wywołaj powyższy consumer na każdym obiekcie z tej listy



Interfejsy funkcyjne

- ***Function<T, R>*** – przetwarza dane

  pobiera obiekt typu T i zwraca obiekt typu R (mapowanie danych)

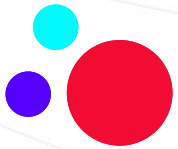
```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

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

infoShare ACADEMY

Ćwiczenie 6c

- napisz własną funkcję dla typu Engine, która zwraca Integer
  - funkcja powinna zwracać moc silnika
- stwórz listę obiektów typu Engine
- wywołaj powyższą funkcję na każdym obiekcie z tej listy



Interfejsy funkcyjne

- ***Supplier<T>*** - odwrotność consumera, np. generator danych

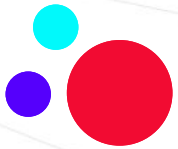
  tworzy obiekt typu T, zawiera metodę get() (dostawca obiektów)

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

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

Ćwiczenie 6d

- napisz własny supplier dla typu Engine
- stwórz obiekt za jego pomocą



Interfejsy funkcyjne

- ***UnaryOperator<T> –*** funkcja pobiera typ T i zwraca typ T (Function<T, T>)

```
UnaryOperator<Integer> square = x -> x * x;
```

- ***BiPredicate<L, R> –*** predykat przyjmujący 2 parametry różnego typu

```
BiPredicate<Car, Integer> biPredicate = (c1, i) -> c1.getMaxSpeed() > i;
```

- ***BiConsumer<T, U> –*** konsument dwóch obiektów, typu T i U

```
BiConsumer<Float, Long> multiplier =
                    (Float x, Long y) -> System.out.println(x * y);
```

Ćwiczenie 7

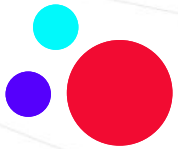
- utwórz listę elementów typu String, w tym kilka pustych
- usuń z listy wszystkie puste elementy (removeIf(..))
- wypisz wszystkie elementy z listy wielkimi literami
- napisz funkcję, która dla zadanej listy zwróci długość jej elementów

  np. {“Andrzej”, “Bob”} zwróci {7, 3}

Ćwiczenie 8

- zmodyfikuj metodę **printResult**
- zastąp interfejs **MathOperation** interfejsem **Function<T, R>**
- do wyświetlania wyników użyj interfejsu **Consumer<T>**

  (dodaj consumer jako parametr metody)

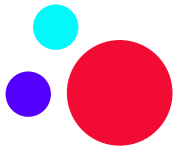


Java 8 – materiały dodatkowe

- https://www.baeldung.com/java-optional
- https://www.samouczekprogramisty.pl/wyrazenia-lambda-w-jezyku-java/

Q&A

infoShareAcademy.com



Thanks!


tomasz.lisowski@proton.me


www.infoshareacademy.com

infoShare ACADEMY

infoShare ACADEMY

Invest in Pomerania ACADEMY

INVEST IN POMERANIA ACADEMY

Fundusze Europejskie Program Regionalny

Rzeczpospolita Polska

URZĄD MARSZAŁKOWSKI WOJEWÓDZTWA POMORSKIEGO

Unia Europejska Europejski Fundusz Rozwoju Regionalnego