



Java 8

Łukasz Bednarski



Metody domyślne



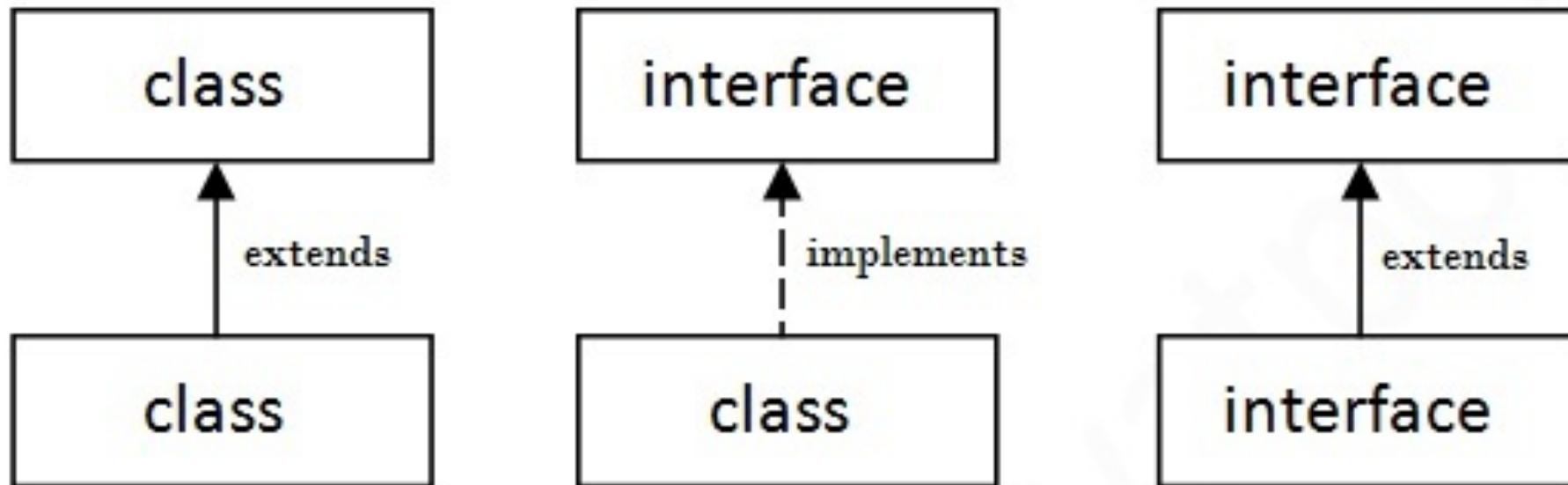
Metody domyślne

- Java 8 umożliwia dodawanie do interfejsów implementacji metod nieabstrakcyjnych
- Służy do tego słowo kluczowe default.

```
interface Formula {  
    double calculate(int a);  
  
    default double sqrt(int a) {  
        return Math.sqrt(a);  
    }  
}
```

```
Formula formula = new Formula() {  
    @Override  
    public double calculate(int a) {  
        return sqrt(a * 100);  
    }  
};  
  
formula.calculate(100);      // 100.0  
formula.sqrt(16);           // 4.0
```

Klasy i interfejsy





Zadanie

- Utwórz interface „MessagePrintable” i dodaj tam jedną metodę „printMessage”. Metoda ta powinna mieć domyślną implementację wyświetlającą w konsoli wiadomość „Hello World”.
- Utwórz 2 implementację interfejsu MessagePrintable: ConsoleMessage oraz LogMessage. W klasie ConsoleMessage nie zmieniaj nic, natomiast w LogMessage nadpisz metodę printMessage tak aby wiadomość była wyświetlona jako LOG (nie ma znaczenia co zostanie wyświetcone).
- Utwórz obiekty i na każdym z nich wywołaj metodę „printMessage”.



java.util.Objects



java.util.Objects

- Klasa dostarczająca kilka statycznych metod do operacji na obiektach. Metody narzędziowe zawierają null-safe lub null-tolerant metody.
- Metody:
 - `isNull(Object)` - true jeśli obiekt jest null, w przeciwnym wypadku false
 - `nonNull(Object)` - false jeśli obiekt jest null, w przeciwnym wypadku true
 - `requireNotNull(Object)` - NPE jeśli przekazany obiekt jest null
 - `requireNotNull(Object, String)` - j/w + komunikat



java.util.Objects

```
class Hosting {  
    private Integer id;  
    private String websites;  
  
    public Hosting(Integer id, String websites) {  
        Objects.requireNonNull(id, message: "Id must not be null");  
        Objects.requireNonNull(websites, message: "Websites must not be null");  
  
        this.id = id;  
        this.websites = websites;  
    }  
}
```

```
public static void main(String[] args) {  
    Integer id = null;  
    Hosting hosting = new Hosting(id, websites: "www.amazon.com");  
}
```

← id == null

The screenshot shows a Java application running in an IDE. At the top, there's a code editor window displaying the `Hosting` class and a `main` method. In the `main` method, the variable `id` is set to `null`, and a new `Hosting` object is created with `id` and the string `"www.amazon.com"`. A blue arrow points from the `id == null` text in the explanatory text above to the `id` parameter in the `new Hosting(id, websites: "www.amazon.com")` line. Below the code editor is a terminal window (Console tab) showing a stack trace for a `NullPointerException`. The stack trace indicates that the exception was thrown at line 228 of `Objects.java` (in the `requireNonNull` method), which was called from line 19 of `DiscoveryarticlesApplication.java` (the constructor of `Hosting`), and finally from line 10 of `DiscoveryarticlesApplication.java` (the `main` method). The terminal also shows standard navigation icons (play, stop, step, etc.) and a tabs bar with `Console` and `Endpoints`.

```
Exception in thread "main" java.lang.NullPointerException: Id must not be null  
at java.util.Objects.requireNonNull(Objects.java:228)  
at pl.lbednarski.discoveryarticles.Hosting.<init>(DiscoveryarticlesApplication.java:19)  
at pl.lbednarski.discoveryarticles.DiscoveryarticlesApplication.main(DiscoveryarticlesApplication.java:10)
```



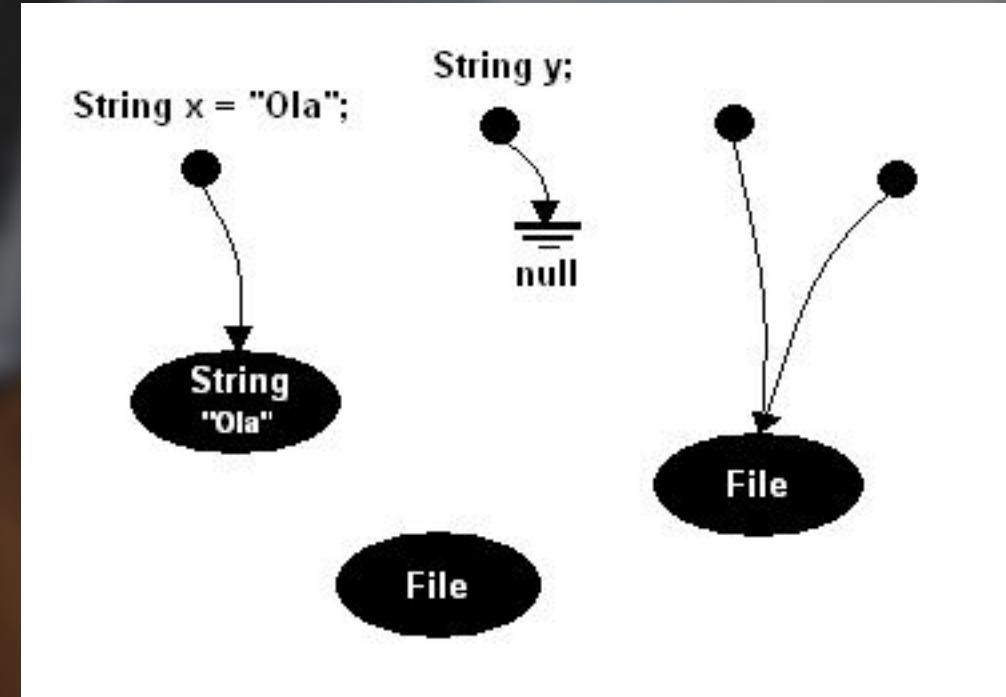
Optional





NullPointerException

- Brak referencji do obiektu - null
- Twórca null-reference jest Tony Hoare
- „I call it my billion-dollar mistake.” - Tony Hoare
- **NullPointerException** jest wyjątkiem z grupy wyjątków nieobsługiwanych
- Najczęściej występujący wyjątek
- Istnieją języki w których null nie występuje





NullPointerException

```
public static void main(String[] args) {  
    Person john = new Person();  
    john.setName("John");  
    System.out.println(john.getName().length());  
  
    Person tom = new Person();  
    System.out.println(tom.getName().length());}  
  
}  
  
class Person{  
  
    private String name;  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```



NPE



Optional

- „Kontener”
- Ułatwia współpracę z null pointerami
- Może posiadać jakąś wartość albo nie
- Zalety Java 8 Optional:
 - 1 Nie trzeba sprawdzać czy obiekt jest NULLem
 - 2 Nie ma problemu z NPE
 - 3 Kod jest bardziej czytelny



Optional

- `empty()` - tworzy pusty Optional z wartością null w środku.
- `of(T value)` - tworzy Optional z podaną wartością. W przypadku przekazania null dostaniemy `nullPointerException`.
 - `ofNullable(T value)` - również tworzy Optional z podaną wartością, ale w przypadku przekazania null nie zostanie zgłoszony wyjątek.
 - `Optional(T value)` - konstruktor rzucający błąd w przypadku przekazania wartości null.
 - `isPresent()` - zwraca boolean mówiący czy w środku znajduje się jakaś wartość czy też null.
 - `get()` - pobranie przechowywanego obiektu. Jeżeli takiego nie dostaniemy: `NoSuchElementException`
 - `ifPresent(function)` - Wykonuje operacje jeśli optional zawiera obiekt a nie null



Optional

```
public static void main(String[] args) {  
    Person john = new Person();  
    john.setName("John");  
    System.out.println(john.getName().length());  
  
    Person tom = new Person();  
    System.out.println(tom.getName().length());  
}  
  
class Person{  
  
    private Optional<String> name = Optional.empty();  
  
    public void setName(String name) {  
        this.name = Optional.ofNullable(name);  
    }  
  
    public String getName() {  
        return name.orElse( other: "");  
    }  
}
```

Wynikiem będzie 0

Utworzenie Optional

Zapisanie jako Optional

Jeśli „name” == null to zwróć
pusty String



Optional

```
public static void main(String[] args) {  
    String name = null; ←  
  
    Person john = new Person();  
    john.setName("John");  
    System.out.println(john.getName().length());  
  
    Person tom = new Person(); ←  
    tom.setName(name);  
    System.out.println(tom.getName().length());  
}  
  
class Person{  
  
    private Optional<String> name = Optional.empty();  
  
    public void setName(String name) {  
        this.name = Optional.ofNullable(name);  
    }  
  
    public String getName() {  
        return name.orElse( other: "");  
    }  
}
```

Utworzenie pola „name” == null

Mimo ustawienia imienia to „name”
nadal pozostaje NULlem



Optional

```
private String saveTrim(final Optional<String> input) {  
    if(input.isPresent()) {  
        return input.get().trim();  
    }  
  
    return "";
```

Sprawdzenie czy Optional zawiera
jakaś wartość
Jeśli tak to wykonuje operacje trim

Jeśli NULL to zwróć pusty String

```
private String saveTrim(final String input) {  
    return input == null ? "" : input.trim();  
}
```

Jaka jest różnica między pierwszym
podejściem a klasycznym ?

```
private String saveTrim(final Optional<String> input) {  
    return Optional.ofNullable(input)  
        .map(String::trim)  
        .orElse("");
```

Poprawnie obsłużony Optional



Zadanie

- Utwórz klasę User zawierające pola firstName oraz lastName. Dodaj także klasę userService która będzie zawierała imitację bazy danych (np. lista). Dodaj także metodę userService oraz findByName.
- Za pomocą addUser dodaj kilka obiektów do „bazy” a następnie za pomocą metody findByName przeszukaj bazę w poszukiwaniu nieistniejącego użytkownika.
- Metoda findByName powinna zwracać Optional<User>.
- Wyświetl imię użytkownika.



Data i czas



LocalDate

- Format ISO (yyyy-MM-dd) bez godziny
- Może reprezentować krótką datę, np. urodzenia
- Przykłady użycia:

```
public static void main(String[] args) {  
    LocalDate localDate = LocalDate.now();  
    LocalDate.of( year: 2015, month: 02, dayOfMonth: 20);  
    LocalDate.parse("2015-02-20");  
  
    LocalDate tomorrow = LocalDate.now().plusDays(1);  
    DayOfWeek sunday = LocalDate.parse("2016-06-12").getDayOfWeek();  
  
    int twelve = LocalDate.parse("2016-06-12").getDayOfMonth();  
    boolean notBefore = LocalDate.parse("2016-06-12")  
        .isBefore(LocalDate.parse("2016-06-11"));  
  
    boolean isAfter = LocalDate.parse("2016-06-12").isAfter(LocalDate.parse("2016-06-11"));  
}
```



LocalTime

- Reprezentuje godzinę bez daty
- Nie zawiera strefy czasowej

```
public static void main(String[] args) {  
    LocalTime now = LocalTime.now();  
    LocalTime sixThirty = LocalTime.parse("06:30");  
    LocalTime fiveThirty = LocalTime.of(hour: 5, minute: 30);  
  
    LocalTime sevenThirty = LocalTime.of(hour: 5, minute: 30).plusHours(2);  
  
    int six = LocalTime.parse("06:30").getHour();  
  
    boolean isbefore = LocalTime.parse("06:30").isBefore(LocalTime.parse("07:30"));  
    LocalTime maxTime = LocalTime.MIN;  
    LocalTime minTime = LocalTime.MAX;  
    LocalTime midnightTime = LocalTime.MIDNIGHT;  
}
```



LocalDateTime

- Reprezentuje godzinę oraz datę
- Nie zawiera strefy czasowej

```
public static void main(String[] args) {  
    LocalDateTime.now();  
    LocalDateTime.of( year: 2015, Month.FEBRUARY, dayOfMonth: 20, hour: 06, minute: 30);  
    LocalDateTime.parse("2015-02-20T06:30:00");  
  
    LocalDateTime.now().plusDays(1);  
    LocalDateTime.now().minusHours(2);  
}
```



ZonedDateTime

- Reprezentuje strefę czasową
- Zawiera 40 stref czasowych
- Pozwala operować na datach dla wielu stref czasowych

```
public static void main(String[] args) {  
    ZoneId zoneId = ZoneId.of("Europe/Paris");  
    Set<String> allZoneIds = ZoneId.getAvailableZoneIds();  
  
    ZonedDateTime zonedDateTime = ZonedDateTime.of(LocalDateTime.now(), zoneId);  
  
    LocalDateTime localDateTime = LocalDateTime.of(year: 2015, Month.FEBRUARY, dayOfMonth: 20, hour: 06, minute: 30);  
    ZoneOffset offset = ZoneOffset.of("+02:00");  
    OffsetDateTime offSetByTwo = OffsetDateTime.of(localDateTime, offset);  
}
```



Zadanie

- Utwórz klasę User (bądź wykorzystaj istniejącą) w której zdefiniujesz pole „data urodzenia”. Dodaj także metodę która zwraca int będący wyliczonym wiekiem użytkownika.
- Dodaj kilku użytkowników do listy a następnie posortuj ich w dowolnej kolejności.



Lambda expression



Lambda

- Dla uproszczenia można powiedzieć, że wyrażenie lambda jest metodą.
- Metodą, którą możesz przypisać do zmiennej.
- Można ją także wywołać czy przekazać jako argument do innej metody.

Podnoszenie liczby do kwadratu:

```
x -> x * x
```

Więc:

```
<lista parametrów> -> <ciało wyrażenia>
```



Lambda

- Lista parametrów zawiera wszystkie parametry przekazane do “ciała” wyrażenia lambda.
- Lista ta może być pusta.
- Wyrażenie lambda poniżej nie przyjmuje żadnych argumentów, zwraca natomiast instancję klasy String:

```
( ) -> "some return value"
```

- Podawanie typów jest opcjonalne:

```
(Integer x, Long y) -> System.out.println(x * y)
```



Lambda

- Jeśli wyrażenie lambda zawiera więcej linii kodu, należy dodać nawiasy {}

```
x -> {  
    if (x != null && x % 2 == 0) {  
        return (long) x * x;  
    }  
    else {  
        return 123L;  
    }  
}
```

Można sobie wyobrazić wyrażenie lambda, które nie przyjmuje żadnych parametrów i nie zwraca żadnych wartości. Najprostsza wersja takiego wyrażenia wygląda następująco:

```
() -> {}
```



Prezydenci

- Jednoargumentowe funkcje o wartości logicznej.
- Ich interfejs zawiera różne metody domyślne służące do łączenia prezydów w złożone operacje logiczne (koniunkcję, alternatywę, negację).

```
Predicate<String> predicate = (s) -> s.length() > 0;  
  
predicate.test("foo");           // true  
predicate.negate().test("foo"); // false  
  
Predicate<Boolean> nonNull = Objects::nonNull;  
Predicate<Boolean>isNull = Objects::isNull;  
  
Predicate<String> isEmpty = String::isEmpty;  
Predicate<String> isNotEmpty = isEmpty.negate();
```



Strumienie (java.util.Stream)

- „Narzędzie” do przetwarzania danych (przekształcanie, filtracja itp) w sposób strumieniowy
- Dzielą się na:
 - pośrednie (ang. intermediate) - nie kończą strumienia, pozwalają na dalsze operowanie na danych (np. map)
 - kończące/terminalne (ang. terminal) - kończą strumień i zwracają wynik (np. forEach)
 - bezstanowe - są wykonane niezależnie od innych danych, co ma znaczenie przy użyciu strumieni równoległych (np. filter)
 - stanowe - ich stan zależy od pozostałych danych i dlatego nie są przetwarzane równolegle (np. sort)
 - redukcyjne - dokonują redukcji danych (np. max)
- Strumienie są wywołane w sposób leniwy, tzn. dane są przetwarzane w momencie wywołania metody kończącej
- Transformacje strumienia nie modyfikują wejściowych danych z których strumień został utworzony



Przykłady operacji

Metoda	Typ	Przeznaczenie	Opis
map(Function...)	pośrednia	transformacja	Zwraca strumień z przekształconymi danymi przy promocji funkcji
filter(Predicate p)	pośrednia	filtrowanie	Zwraca strumień danych dla których warunek będzie spełniony
sorted(...)	pośrednia, stanowa	transformacja	Przekształca strumień do postaci posortowanej
unordered()	pośrednia	transformacja	Przekształca strumień do postaci nieuporządkowanej
allMatch(Predicate p)	kończąca	redukcia	Testuje czy w strumieniu jeden wszystkie obiekty spełniają podany warunek
anyMatch(Predicate p)	kończąca	redukcia	Testuje czy w strumieniu jeden chociaż jeden obiekt spełnia podany warunek



Przykłady operacji

findAny()	kończąca	filtrowanie	Zwraca dowolny element strumienia
findFirst()	kończąca	filtrowanie	Zwraca pierwszy element strumienia
count()	kończąca	redukcia	
forEach(Consumer c)	Kończąca, bezstanowa		Wykonuje akcję z każdym elementem strumienia, zamykając go jednocześnie
collect(...)	kończąca		Grupuje wszystkie elementy pozostające w strumieniu i zwraca je w postaci definiowanej przez podany Collector
max(...)	kończąca	redukcia	Zwraca największą wartość wynikającą z zastosowanego komparatora
min(...)	kończąca	redukcia	Zwraca najmniejszą wartość wynikającą z zastosowanego komparatora
toArray(...)	kończąca		Przekształca strumień do tablicy danych



forEach

```
public static void main(String[] args) {  
    List<String> names = Arrays.asList("Jan", "Tom", "Ryszard", "Mateusz");  
  
    for(String name : names){  
        System.out.println(name);  
    }  
}
```

```
public static void main(String[] args) {  
    List<String> names = Arrays.asList("Jan", "Tom", "Ryszard", "Mateusz");  
  
    names.forEach(name -> System.out.println(name));  
}
```

```
public static void main(String[] args) {  
    List<String> names = Arrays.asList("Jan", "Tom", "Ryszard", "Mateusz");  
  
    names.forEach(System.out::println);  
}
```



filter & Collectors

```
public static void main(String[] args) {  
    List<String> names = Arrays.asList("Jan", "Tom", "Ryszard", "Mateusz");  
  
    List<String> filteredNames = new ArrayList<>();  
    for (String name : names){  
        if (name.contains("a")){  
            filteredNames.add(name);  
        }  
    }  
  
    for (String name : filteredNames){  
        System.out.println(name);  
    }  
}
```

```
public static void main(String[] args) {  
    List<String> names = Arrays.asList("Jan", "Tom", "Ryszard", "Mateusz");  
  
    List<String> filteredList = names.stream()  
        .filter(name -> name.contains("a"))  
        .collect(Collectors.toList());  
  
    filteredList.forEach(name -> System.out.println(name));  
}
```

Odrzuć imiona które nie
zawierają literki „a”
Z elementów które zostały
utwórz i zwróć listę



filter & Collectors

```
public static void main(String[] args) {  
    List<String> names = Arrays.asList("Jan", "Tom", "Ryszard", "Mateusz");  
  
    List<String> filteredList = names.stream()  
        .filter(name -> name.contains("a"))  
        .collect(Collectors.toList());  
  
    filteredList.forEach(name -> System.out.println(name));  
}
```

```
public static void main(String[] args) {  
    List<String> names = Arrays.asList("Jan", "Tom", "Ryszard", "Mateusz");  
  
    names.stream()  
        .filter(name -> name.contains("a"))  
        .collect(Collectors.toList())  
        .forEach(name -> System.out.println(name));  
}
```

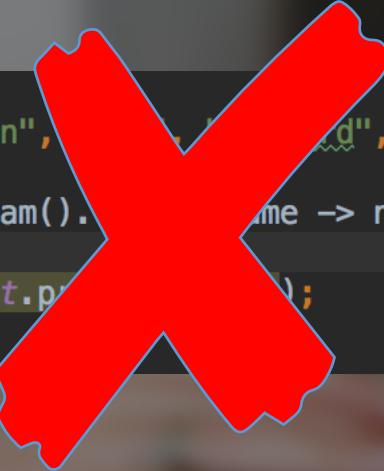


Od razu wyświetla listę imion



Formatowanie

```
public static void main(String[] args) {  
    List<String> names = Arrays.asList("Jan", "Tom", "Ryszard", "Mateusz");  
  
    List<String> filteredList = names.stream()  
        .filter(name -> name.contains("a")).collect(Collectors.toList());  
  
    filteredList.forEach(name -> System.out.println(name));  
}
```



```
public static void main(String[] args) {  
    List<String> names = Arrays.asList("Jan", "Tom", "Ryszard", "Mateusz");  
  
    List<String> filteredList = names.stream()  
        .filter(name -> name.contains("a"))  
        .collect(Collectors.toList());  
  
    filteredList.forEach(name -> System.out.println(name));  
}
```

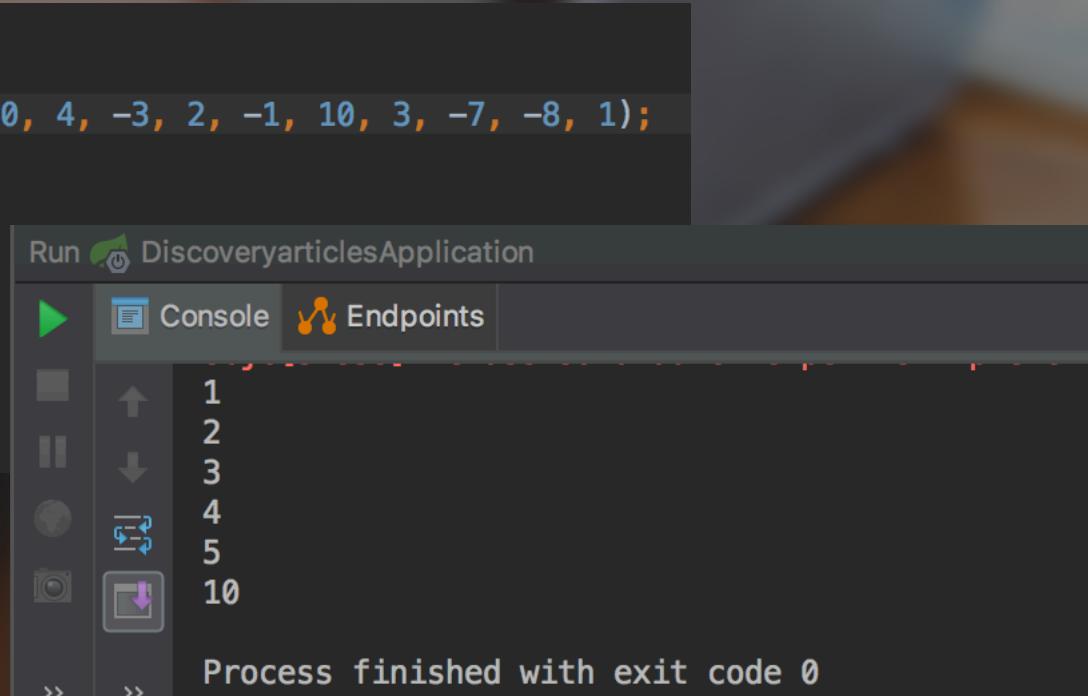




Zadanie

- Utwórz listę obiektów typu Integer
- Dodaj do listy kilka wartości z przedziału <-10;10>
- Utwórz strumień w którym odfiltrujesz wszystkie wartości poniżej 0
- Wyświetl posortowaną listę

```
public static void main(String[] args) {  
  
    List<Integer> values = Arrays.asList(-10, 5, 0, 4, -3, 2, -1, 10, 3, -7, -8, 1);  
  
    values.stream()  
        .filter(value -> value > 0)  
        .sorted()  
        .collect(Collectors.toList())  
        .forEach(System.out::println);  
  
}
```



The screenshot shows the IntelliJ IDEA interface during the execution of a Java application. The title bar indicates the application name is 'DiscoveryarticlesApplication'. The 'Run' tab is selected. In the bottom right corner of the run window, the message 'Process finished with exit code 0' is displayed. The console output is visible in the main pane, showing the sorted list of integers: 1, 2, 3, 4, 5, and 10.

```
Run DiscoveryarticlesApplication  
Console Endpoints  
1  
2  
3  
4  
5  
10  
Process finished with exit code 0
```



Map

- Zwraca strumień z przekształconymi danymi przy pomocy funkcji
- Konwertuje dane i zwraca po przekształceniu

```
public static void main(String[] args) {  
    List<Person> names = Arrays.asList(new Person( name: "Jan"),  
                                         new Person( name: "Tom"),  
                                         new Person( name: "Max"));  
  
    names.stream()  
        .collect(Collectors.toList())  
        .forEach(name -> System.out.println(name));  
  
    names.stream()  
        .map(name -> name.getName())  
        .collect(Collectors.toList())  
        .forEach(name -> System.out.println(name));  
}
```

Przekształcenie strumienia który przechowuje obiekty klasy Person na listę Stringów

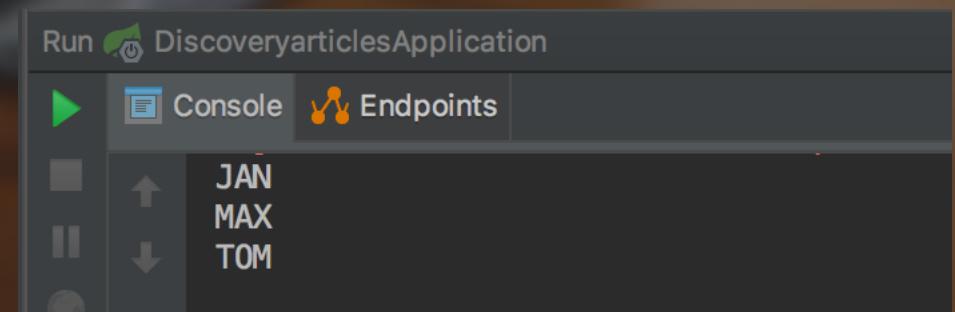
```
pl.lbednarski.discoveryarticles.Person@7d6f77cc  
pl.lbednarski.discoveryarticles.Person@5aaa6d82  
pl.lbednarski.discoveryarticles.Person@73a28541  
Jan  
Tom  
Max
```



Zadanie

- Utwórz listę obiektów typu Person
- Dodaj do listy kilka imion
- Przekształć imiona znajdujące się na liście tak aby były napisane wielkimi literami
- Wyświetl listę posortowanych imion

```
public static void main(String[] args) {  
    List<Person> names = Arrays.asList(new Person( name: "Jan"),  
                                         new Person( name: "Tom"),  
                                         new Person( name: "Max"));  
  
    names.stream()  
        .map(name -> name.getName())  
        .map(name -> name.toUpperCase())  
        .sorted()  
        .collect(Collectors.toList())  
        .forEach(name -> System.out.println(name));  
}
```





Array -> Stream

- Aby łatwiej korzystać z programowania funkcyjnego tablicę należy zamienić na stream.
- Istnieją 2 operacje:
 - Arrays.stream(tablica)
 - Stream.of(tablica)

```
public static void main(String[] args) {  
    String[] array = {"a", "b", "c", "d", "e"};  
  
    Stream<String> stream1 = Arrays.stream(array);  
    stream1.forEach(x -> System.out.println(x));  
  
    Stream<String> stream2 = Stream.of(array);  
    stream2.forEach(x -> System.out.println(x));  
}
```



toMap()

- Pozwala przekonwertować strumień do mapy.

```
class Hosting {  
  
    private int Id;  
    private String name;  
    private long websites;  
  
    public Hosting(int id, String name, long websites) {  
        Id = id;  
        this.name = name;  
        this.websites = websites;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getId() {  
        return Id;  
    }  
  
    public long getWebsites() {  
        return websites;  
    }  
}
```

```
public static void main(String[] args) {  
    List<Hosting> list = new ArrayList<>();  
    list.add(new Hosting(1, "example.com", 80000));  
    list.add(new Hosting(2, "microsoft.com", 90000));  
    list.add(new Hosting(3, "facebook.com", 120000));  
    list.add(new Hosting(4, "amazon.com", 200000));  
    list.add(new Hosting(5, "allegro.com", 1));  
  
    list.stream()  
        .collect(Collectors.toMap(x -> x.getId(), x -> x.getName()))  
        .forEach( (k,v) -> System.out.println("Key: " + k + ", Value: " + v));  
}
```



removelf

- Pozwala usunąć z listy obiekty na podstawie funkcji
- Należy utworzyć predykat i przekazać go jako funkcję do metody `removelf()`

```
public static void main(String[] args) {  
    List<User> johnFamily = Arrays.asList(new User( name: "Mike", lastName: "Kowalski"),  
                                         new User( name: "Tom", lastName: "Nowak"),  
                                         new User( name: "Rick", lastName: "Allan"));  
  
    User john = new User( name: "John", lastName: "Kowalski");  
    john.setFamily(johnFamily);  
  
    List<User> rickFamily = Arrays.asList(new User( name: "Lily", lastName: "Bab"),  
                                         new User( name: "Tom", lastName: "Nowak"),  
                                         new User( name: "Fan", lastName: "Dtro"));  
  
    User rick = new User( name: "Rick", lastName: "Kowalski");  
    rick.setFamily(rickFamily);  
  
    List<User> allUser = new ArrayList<>(Arrays.asList(john, rick));  
  
    allUser.forEach(user -> System.out.println(user));  
  
    Predicate<User> userPredicate = user -> user.getName().equals("John");  
    allUser.removeIf(userPredicate);  
  
    allUser.forEach(user -> System.out.println(user));  
}
```



Utworzenie predykatu
Usuwa elementy które spełniają warunek



Zadanie

- Do klasy User dodaj wiek.
 - Utwórz kilku użytkowników i wyświetl ich pełną listę
 - Utwórz predykat który pozwoli Ci usunąć osoby poniżej 18 roku życia
 - Wyświetl listę użytkowników po redukcji
 - Popraw predykat tak aby usuwać:
 - Mężczyzn poniżej 18 roku życia
 - Kobiety poniżej 21 roku życia
- (Dla naszych potrzeb założmy że wszystkie imiona żeńskie kończą się na „a” ;))



flatMap

- Pozwala „spłaszczyć” kilka kolekcji
- Przykład: Stream<List<User>>

```
public static void main(String[] args) {  
    List<User> johnFamily= Arrays.asList(new User( name: "Mike", lastName: "Kowalski"),  
                                         new User( name: "Tom", lastName: "Nowak"),  
                                         new User( name: "Rick", lastName: "Allan"));  
  
    User john = new User( name: "john", lastName: "Kowalski");  
    john.setFamily(johnFamily);  
  
    List<User> rickFamily = Arrays.asList(new User( name: "Lily", lastName: "Bab"),  
                                         new User( name: "Mike", lastName: "Ras"),  
                                         new User( name: "Fan", lastName: "Dtro"));  
  
    User rick = new User( name: "Rick", lastName: "Kowalski");  
    rick.setFamily(rickFamily);  
  
    List<User> allUser = Arrays.asList(john, rick);  
  
    allUser.stream()  
        .map(user -> user.getFamily())  
        .flatMap(family -> family.stream())  
        .collect(Collectors.toList())  
        .forEach(System.out::println);  
}
```

Konwersja
Spłaszczenie kolekcji



Zadanie

- Utwórz klasę Company w której zdefiniujesz nazwę firmy (typu String) oraz właściciela firmy (typu User). Dodaj także klasę Address w której będzie możliwość zdefiniowania państwa, miasta i ulicę z numerem.
- W klasie Company dodaj listę adresów o nazwie np. „departments” - będzie ona reprezentowała listę oddziałów danej firmy.
- Utwórz kilka firm i dodaj do nich listę oddziałów a następnie:
 1. Wyświetl listę firm wraz z ich właścicielami: Company : Owner
 2. Wyświetl listę wszystkich oddziałów każdej z firm
 3. Wyświetl listę wszystkich oddziałów (wykorzystaj flatMap)
 4. Posortuj listę wszystkich oddziałów alfabetycznie według krajów (możesz spróbować też posortować po kraju i mieście).



Zadanie

- Zmień pole definiujące właściciela na listę (będzie ona reprezentowała listę współwłaścicieli). Zdefiniuj listę kilku z nich a następnie:
 1. Wyświetl listę nazwisk wszystkich właścicieli firm.
 2. Jeśli jakąś osobę występuje więcej niż raz, nie wyświetlaj dodatkowych rekordów (duplikaty możesz usunąć za pomocą metody `.distinct()`)
 3. Dodaj interfejs CompanyService który zawierał będzie:
 1. Dodawanie firm
 2. Usuwanie firm po nazwie
 3. Dodawanie współwłaściciela dla konkretnej firmy
 4. Pobranie właściciela lub współwłaścicieli danej firmy
 5. Dodanie oddziału dla konkretnej firmy
 4. Dodaj implementację interfejsu i zaimplementuj wszystkie powyższe metody (zamiast podłączania bazy możesz wykorzystać listę)



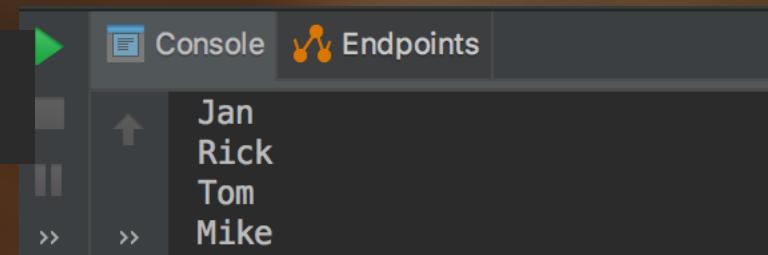
Interfejsy funkcyjne



Referencje do metody

- Odwołanie do istniejącej metody lub konstruktora
- Referencja do metody statycznej:
 - KlasaZawierającaMetode::metodaStatyczna
 - np. Objects::isNull
- Referencja do metody należącej do konkretnej instancji
 - obiektZawierającyMetodę::nazwaMetody
 - np. user::getName
- Referencja do konstruktora
 - NazwaKlasy::new

```
Stream.of("Jan", "Rick", "Tom", "Mike")
    .forEach(System.out::println);
```





Interfejsy funkcyjne

- Interfejs funkcyjny jest interfejsem, który posiada tylko jedną metodę abstrakcyjną.
- Pozwala Javie (od wersji 8) na wykorzystanie wyrażeń lambda w miejscu, gdzie normalnie kompilator oczekuje takiego typu danych.
- Interfejsy funkcyjne opcjonalnie oznaczane są adnotacją **@FunctionalInterface**.
 - Przykładem może być interface Comparable i metoda CompareTo
 - W dużym uproszczeniu pozwala przygotować zachowanie które zostanie później wykorzystane
 - W pakiecie java.util.function znajduje się wiele przydatnych interfejsów



Przykłady interfejsów funkcyjnych

- Function <T, R> - przyjmuje dowolny obiekt i zwraca dowolny obiekt (T, R),
- Consumer <T> - przyjmuje dowolny obiekt, ale nic nie zwraca (T, void),
- Supplier <T> - nic nie przyjmuje, ale zwraca dowolny obiekt (void, T),
- Predicate <T> - przyjmuje dowolny obiekt, ale zwraca boolean (T, boolean),
- UnaryOperator <T, T> - przyjmuje oraz zwraca obiekt dowolnego typu, ale muszą być takie same (T, T) szczególny przypadek Function,

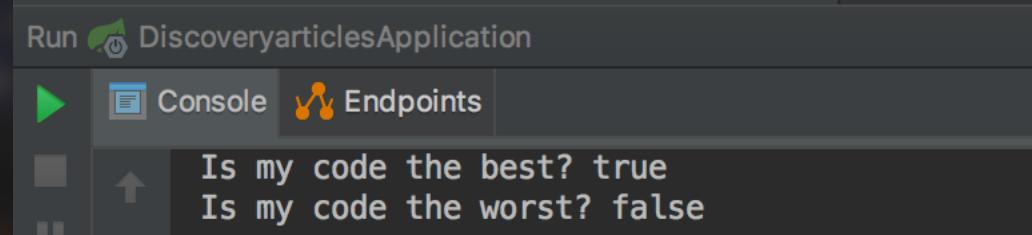


Predicate <T>

- **Predicate <T>** - przyjmuje dowolny obiekt, ale zwraca boolean (T, boolean),
- Najważniejsza jest metoda test

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

```
Predicate<String> textContainsBest = text -> text.contains("best");  
  
System.out.println("Is my code the best? " + textContainsBest.test( t: "My Code is the best"));  
System.out.println("Is my code the worst? " + textContainsBest.test( t: "My Code is the worst"));
```





Predicate <T>

```
Predicate<String> containsTom = textToCheck -> textToCheck.contains("Tom");  
  
Stream.of("Jan", "Rick", "Tom")  
    .filter(containsTom)  
    .forEach(System.out::println);
```

```
Predicate<String> containsTom = textToCheck -> textToCheck.contains("Tom");  
  
Stream.of("Jan", "Rick", "Tom")  
    .filter(containsTom.negate())  
    .forEach(System.out::println);
```

```
Predicate<String> containsMike = textToCheck -> textToCheck.contains("Mike");  
Predicate<String> containsTom = textToCheck -> textToCheck.contains("Tom");  
  
Stream.of("Jan", "Rick", "Tom")  
    .filter(containsTom.or(containsMike))  
    .forEach(System.out::println);
```

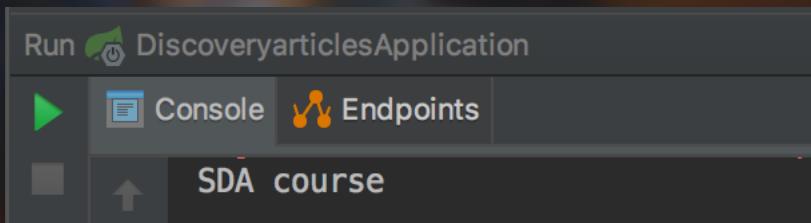


Consumer <T> 1

- Consumer <T> - przyjmuje dowolny obiekt, ale nic nie zwraca (T, void),
- Najważniejszą metodą jest metoda accept(),

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

```
Consumer<String> showSDAtext = text -> System.out.println(text);  
  
showSDAtext.accept( t: "SDA course");
```





Consumer <T>

```
int x = 99;

Consumer<Integer> myConsumer = (y) ->
{
    System.out.println("x = " + x);
    System.out.println("y = " + y);
};

myConsumer.accept(x);
```

```
Consumer<String> showSDAtext = text -> showText3Times(text);

showSDAtext.accept( t: "SDA course");

}

private static void showText3Times(String text){
    for (int i = 0 ; i < 3 ; i ++){
        System.out.println(text);
    }
}
```



Consumer <T>

- Przygotuj prostą metodę która przyjmuje obiekt klasy User a następnie wyświetla informację o nim.
- Dodaj do klasy User pole typu LocalDate które będzie przechowywało datę urodzenia użytkownika.
- Dodaj także metodę zwracającą wiek użytkownika.
- Wyświetl informację czy osoba jest pełnoletnia (w metodzie utwórz predykat).
- Utwórz instancję interfejsu Consumer do którego przekażesz użytkownika i po wywołaniu metody accept wyświetcone zostaną parametry.

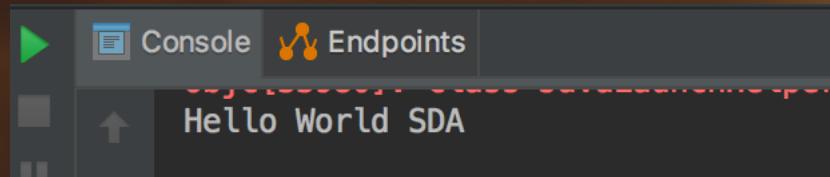


Supplier <T>

- Supplier <T> - nic nie przyjmuje, ale zwraca dowolny obiekt (void, T),
- Najważniejszą metodą jest metoda get(),

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

```
Supplier<String> returnSDAText = () -> "Hello World SDA";  
System.out.println(returnSDAText.get());
```





Supplier <T>

- Utwórz dostawcę obiektów klasy User z losowo wypełnionymi polami.
- Zdefiniuj listę np. 5 imion i nazwisk.
- Dodaj metodę która będzie tworzyła i zwracała użytkownika z losowymi danymi (w tym także wiek).
- Po wywołaniu funkcji get() dostawca powinien zwracać użytkownika z losowymi polami.
- Utwórz kilku użytkowników i wyświetl ich.



Function <T, R>

- Function <T, R> - przyjmuje dowolny obiekt i zwraca dowolny obiekt (T, R),

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

- Przyjmuje oraz zwraca dowolny typ.
- Najważniejszą metodą w tym interfejsie jest metoda R apply(T t). Wywołuje ona pożądaną dla nas akcję.



Zadanie

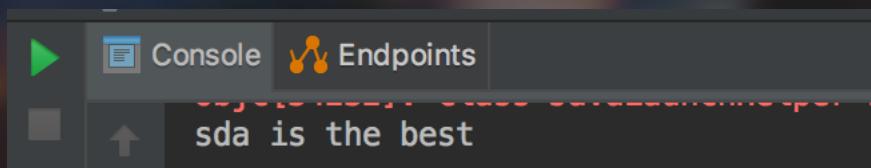
- Utwórz instancję interfejsu Function.
- Jako parametr do funkcji przekaż String, natomiast zwracany powinien być obiekt typu User.
- Funkcja powinna parsować przekazany String i na jego podstawie tworzyć i zwracać obiekt typu User.
- Np.
- String text = „Jan,Nowak,18” zwróci Ci użytkownika z ustalonymi wartościami dla poszczególnych pól.



UnaryOperator <T, T>

- UnaryOperator <T, T> - przyjmuje oraz zwraca obiekt dowolnego typu, ale muszą być takie same (T, T) szczególny przypadek Function,
- Przyjmuje oraz zwraca dowolny typ jednakże oba muszą być tej samej klasy.
- Rozszerza interface Function
- Najważniejszą metodą w tym interfejsie jest metoda T apply(T t).

```
UnaryOperator<String> sdaText = text -> text + " is the best";
System.out.println(sdaText.apply( t: "sda" ));
```





UnaryOperator <T, T>

```
public static void main(String[] args) {
    User john = new User(name: "John", lastName: "Kowalski");
    System.out.println(john);

    UnaryOperator<User> fillDefault = defaultJohn -> fillUserDefaultValue(defaultJohn);
    System.out.println(fillDefault.apply(john));
}

private static User fillUserDefaultValue(User user){
    user.setName("Default John");
    user.setLastName("Default Kowalski");

    return user;
}
```



Własny interfejs funkcyjny

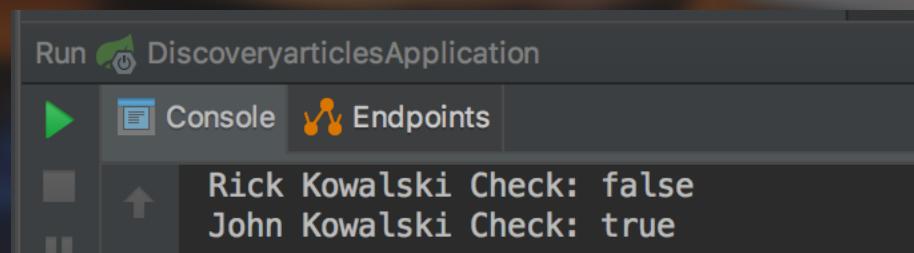
- Oprócz dostarczonych już interfejsów funkcyjnych z pakietu `java.util.function` możesz także tworzyć swoje
- Twój interfejs może mieć tylko 1 metodę (drobnym odstępstwem od reguły jest to że może mieć więcej metod, jednakże muszą to być metody typu `default` lub `static`).

```
@FunctionalInterface  
public interface UserChecker {  
  
    boolean checkNameContainsJan(User user);  
  
}
```



Własny interfejs funkcyjny

```
public static void main(String[] args) {  
    User rick = new User( name: "Rick", lastName: "Kowalski");  
    User john = new User( name: "John", lastName: "Kowalski");  
  
    UserChecker personChecker = (p) -> p.getName().contains("John");  
  
    System.out.println(rick.toString() + " Check: " + personChecker.checkNameContainsJan(rick));  
    System.out.println(john.toString() + " Check: " + personChecker.checkNameContainsJan(john));  
}
```





Zadanie

- Utwórz swój własny interface który jako argument będzie przyjmował:
 - Obiekt klasy User
 - Obiekt typu String
 - Metoda powinna zwracać obiekt typu Company
-
- Twoja metoda będzie wykonywała rejestrację firmy. Przekazując dane o użytkowników oraz nazwę firmy, wywołanie metody np. registerCompany zwróci obiekt klasy Company z wypełnionymi wszystkimi polami (w tym z unikatowym ID firmy).
 - Metoda powinna sprawdzać czy użytkownik jest pełnoletni a pola nie są puste (możesz wykorzystać predykaty).



Zadanie

- Utwórz klasę która będzie reprezentowała walidację dla parametrów firm. Zdefiniuj w niej listę słów kluczowych których nie wolno używać w nazwie firmy (np. słowa powszechnie uważane za obraźliwe).
- Rozbuduj klasę User o pole PESEL a następnie utwórz w validatorze listę PESELi które z jakiegoś powodu nie mogą utworzyć firmy. Możesz to zrobić wykorzystując HashMapę <PESEL;Powód>
- Jeśli okaże się że użytkownik chce zarejestrować firmę niezgodną z prawem - możesz rzucić wyjątkiem z komunikatem dlaczego nie udało się zarejestrować działalności.