

Java SE 8



Witam!

Rafał Kurt

Senior Software Developer

rafalkurt@gmail.com

<https://github.com/rkurt/solwit-java8.git>

Skąd ten rozwój?

Java od lat pozostaje w czołówce najpopularniejszych języków programowania.

Kompetencje związane z tą technologią są stale w cenie



Technologię Java wykorzystuje już:

ok. 9 000 000

programistów

ok. 3 000 000 000

urządzeń

Droga do wydania Java SE w wersji 8

2010 r.

Zatwierdzenie
specyfikacji
JSR-337

2012 r.

Planowany debiut

18 marca 2014 r.

Ostateczny debiut

Rewolucja czy ewolucja?

Środowisko Java SE w wersji 8 to:

- nowe funkcje pozwalające zwiększyć efektywność prac programistycznych
- usprawnienia wydajnościowe
- mniejsze wymagania zasobów pamięci

Przemyślana ewolucja

Java SE w wersji 8 pozwala zmienić sposób, w jaki programiści piszą swoje programy.

Aby uzyskać szybkie porównanie, zobaczmy program sortujący napisany zarówno z użyciem składni Java SE w wersji 7 i Java SE w wersji 8.

Porównajmy...

// Java SE 7

```
public static void main(String[] args) {  
    List<String> names = new ArrayList<>();  
    names.add("Tomasz");  
    names.add("Wioletta");  
    names.add("Anna");  
  
    names.sort(new Comparator<String>() {  
        @Override  
        public int compare(String s1, String s2) {  
            return s1.compareTo(s2);  
        }  
    });  
  
    System.out.println(names);  
}
```

// Java SE 8

```
public static void main(String[] args) {  
    List<String> names = new ArrayList<>();  
    names.add("Tomasz");  
    names.add("Wioletta");  
    names.add("Anna");  
  
    names.sort((s1, s2) -> s1.compareTo(s2));  
  
    System.out.println(names);  
}
```



1. Wyrażenia lambda

Zastosowanie wyrażenia lambda

Wprowadzenie wyrażen lambda uznawane jest za największą wartość Java SE w wersji 8:

- umożliwia programowanie funkcjonalne, tj. skupienie uwagi bardziej na tym co chce się zrobić niż zajmowanie się obiektami
- „metoda anonimowa”, tj. posiada parametry i ciało metody, ale nie posiada nazwy
- funkcjonalność przekazywana jako argument metody
- wykorzystują interfejsy funkcyjne, tj. z 1 metodą abstrakcyjną
- znacznie upraszcza pisanie kodu, powstaje mniej klas

Składnia wyrażenia lambda

parameter -> expression body

Przykłady wyrażenia:

- *a -> a.methodName()*
 - 1 parametr
 - 1 instrukcja w ciele wyrażenia
- *(int a, int b) -> { int x = 1; return a * b + x; }*
 - więcej parametrów
 - więcej instrukcji w ciele wyrażenia

Składnia wyrażenia lambda

Elementy opcjonalne:

- typy parametrów – nie ma potrzeby deklarować, kompilator wnioskuje typ po wartości parametru
- nawias wokół parametrów – nie jest wymagany tylko wtedy, gdy jest 1 parametr
- nawias klamrowy wokół ciała wyrażenia, słowo kluczowe *return* oraz średnik – nie są wymagane tylko wtedy, gdy ciało zawiera 1 instrukcję

Składnia wyrażenia lambda

Błędne przykłady:

- `print(a, b -> a.startsWith("test"));`
- `print(a -> { a.startsWith("test"); });`
- `print(a -> { return a.startsWith("test") });`

Czy jesteś w stanie wskazać błędy zawarte w każdym z przykładów?

Przykład użycia wyrażenia lambda

```
public class Animal {  
  
    private String species;  
    private boolean canJump;  
    private boolean canSwim;  
  
    public Animal(String species, boolean canJump, boolean canSwim) {  
        this.species = species;  
        this.canJump = canJump;  
        this.canSwim = canSwim;  
    }  
  
    public String getSpecies() {  
        return species;  
    }  
  
    public boolean canJump() {  
        return canJump;  
    }  
  
    public boolean canSwim() {  
        return canSwim;  
    }  
}
```

Przykład użycia wyrażenia lambda

```
public class CheckIfJumper implements CheckSkill {  
  
    @Override  
    public boolean test(Animal a) {  
        return a.canJump();  
    }  
}
```

```
public interface CheckSkill {  
    boolean test(Animal a);  
}
```

```
public class CheckIfSwimmer implements CheckSkill {  
  
    @Override  
    public boolean test(Animal a) {  
        return a.canSwim();  
    }  
}
```

Przykład użycia wyrażenia lambda

```
// Java SE 7
public class AnimalDemo {

    private static void print(List<Animal> animals, CheckSkill checker) {
        for (Animal animal : animals) {
            if (checker.test(animal)) {
                System.out.print(animal.getSpecies() + " ");
            }
        }
        System.out.println();
    }

    public static void main(String[] args) {
        List<Animal> animals = new ArrayList<>();
        animals.add(new Animal("fish", false, true));
        animals.add(new Animal("kangaroo", true, false));
        animals.add(new Animal("rabbit", true, false));
        animals.add(new Animal("turtle", false, true));

        print(animals, new CheckIfJumper()); // kangaroo rabbit
        print(animals, new CheckIfSwimmer()); // fish turtle
    }
}
```


Przykład użycia wyrażenia lambda

```
// Java SE 8
public class AnimalDemo {

    private static void print(List<Animal> animals, CheckSkill checker) {
        for (Animal animal : animals) {
            if (checker.test(animal)) {
                System.out.print(animal.getSpecies() + " ");
            }
        }
        System.out.println();
    }

    public static void main(String[] args) {
        List<Animal> animals = new ArrayList<>();
        animals.add(new Animal("fish", false, true));
        animals.add(new Animal("kangaroo", true, false));
        animals.add(new Animal("rabbit", true, false));
        animals.add(new Animal("turtle", false, true));

        print(animals, a -> a.canJump()); // kangaroo rabbit
        print(animals, a -> a.canSwim()); // fish turtle
    }
}
```

Jakie zmiany ponadto nastąpiły w kodzie?

Przykład użycia wyrażenia lambda

```
// Java SE 8
public class AnimalDemo {

    private static void print(List<Animal> animals, CheckSkill checker) {
        for (Animal animal : animals) {
            if (checker.test(animal)) {
                System.out.print(animal.getSpecies() + " ");
            }
        }
        System.out.println();
    }

    public static void main(String[] args) {
        List<Animal> animals = new ArrayList<>();
        animals.add(new Animal("fish", false, true));
        animals.add(new Animal("kangaroo", true, false));
        animals.add(new Animal("rabbit", true, false));
        animals.add(new Animal("turtle", false, true));

        print(animals, a -> a.canJump()); // kangaroo rabbit
        print(animals, a -> a.canSwim()); // fish turtle
    }
}
```

*Klasy **CheckIfJumper** i **CheckIfSwimmer** trafiły właśnie do kosza.*

Wyrażenia lambda - dostęp do zmiennych

Wewnątrz wyrażenia lambda można odwoływać się do zmiennych lokalnych, które są zadeklarowane poza wyrażeniem:

- zmienna musi być przynajmniej niejawnie *final*
- przypisanie wartości do zmiennej tylko raz
- wyrażenie lambda generuje błąd kompilacji, jeśli wartość do zmiennej zostanie przypisana po raz drugi

Pola klasy oraz zmienne statyczne można zarówno odczytać, jak i nadpisać.

Wyrażenia lambda - dostęp do zmiennych

```
static int s = 1;
```

```
public static void main(String[] args) {  
    List<Animal> animals = new ArrayList<>();  
    animals.add(new Animal("fish", false, true));  
    animals.add(new Animal("kangaroo", true, false));  
    animals.add(new Animal("rabbit", true, false));  
    animals.add(new Animal("turtle", false, true));
```

```
    print(animals, a -> a.canJump()); // kangaroo rabbit
```

```
    print(animals, a -> {s = 2; return a.canSwim();}); // compilation ok
```

```
    int k = 1;
```

```
    print(animals, a -> {int z = k + 1; return a.canSwim();}); // compilation ok
```

```
    int i = 1;
```

```
    print(animals, a -> {i = 2; return a.canSwim();}); // compilation error
```

```
    int m = 1;
```

```
    print(animals, a -> {int m = 2; return a.canSwim();}); // compilation error
```

```
}
```

Czy jesteś w stanie wskazać przyczynę błędu kompilacji?

Wyrażenia lambda – ćwiczenie (1a)

Utwórz interfejs "MathOperation" z 1 metodą "calculate" do obliczeń matematycznych. Parametrem wejściowym powyższej metody jest lista elementów typu Integer. Metoda zwraca wartość typu Integer.

Przygotuj 2 klasy, które implementują ten interfejs.

Pierwsza "MaxOperation" niech zwraca największą liczbę z podanej listy.

Druga "MinOperation" niech zwraca najmniejszą liczbę z podanej listy.

(warto użyć metod z klasy java.util.Collections)

Utwórz klasę "Java8Demo" z metodą "main".

Przygotuj prywatną metodę „getResult”, która przyjmuje listę elementów typu Integer oraz instancję "MathOperation".

Powyższa metoda ma nie zwracać żadnej wartości (void). Będzie ona wykonywać obliczenia oraz wyświetlać wynik typu "Wynik działania metody = X".

Przygotuj dowolne dane testowe w metodzie "main". Wykonaj obliczenia używając utworzonych obiektów klasy "MinOperation" i "MaxOperation".

Wyrażenia lambda – ćwiczenie (1b)

W klasie "Java8Demo" użyj wyrażeń lambda tak, aby nie trzeba było tworzyć instancji obiektów klasy MaxOperation i MinOperation.

Klasę MaxOperation i MinOperation można usunąć. 

2. Interfejsy funkcyjne

Zastosowanie interfejsów funkcyjnych

Interfejs funkcyjny jest to interfejs, który:

- eksponuje tylko 1 funkcjonalność
- zawiera dokładnie 1 metodę abstrakcyjną
- jest wykorzystywany przez wyrażenia lambda
- może zawierać dowolną liczbę metod *static* i *default*
- opcjonalna adnotacja *@FunctionalInterface* - jej obecność powoduje, że kompilator będzie kontrolował liczbę metod abstrakcyjnych zawartych w interfejsie

Zastosowanie interfejsów funkcyjnych

Przykład stworzenia i użycia interfejsu funkcyjnego:

```
@FunctionalInterface
```

```
public interface Converter<F, T> {  
    T convert(F from);  
}
```

```
public static void main(String[] args) {  
    Converter<String, Integer> converter = (String from) -> Integer.valueOf(from);  
    Integer converted = converter.convert("1000");  
    System.out.println(converted); // 1000  
}
```

Czy za każdym razem trzeba tworzyć taki interfejs?

Wbudowane interfejsy funkcyjne

Java SE w wersji 8 zawiera wiele wbudowanych interfejsów funkcyjnych:

- pakiet *java.util.function*
- mają za zadanie ułatwić nam pracę
- zmniejszyć ilość wytwarzanego kodu
- niektóre z nich jak, np. *Comparator* czy *Runnable*, wywodzą się ze starszych wersji Java
- niektóre z nich wywodzą się z biblioteki *Google Guava*

Predicate<T>

Predykaty to jednoargumentowe funkcje, które zwracają wartość logiczną dla podanej wartości typu T. Ich interfejs zawiera różne metody domyślne służące do łączenia predykatów w złożone operacje logiczne.

@FunctionalInterface

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

```
public static void main(String[] args) {  
    Predicate<String> isEmpty = (s) -> s.isEmpty();  
    Predicate<String> isEmpty = isEmpty.negate();  
  
    boolean b1 = isEmpty.test("isa"); // false  
    boolean b2 = isEmpty.test("isa"); // true  
    boolean b3 = isEmpty.and(isEmpty).test("isa"); // false  
}
```

Function<T, R>

Funkcje przyjmują jeden argument i zwracają wynik. Dokonują przekształcenia wartości typu T na wartość typu R. Ich interfejs zawiera różne metody domyślne służące do łączenia funkcji w łańcuch.

@FunctionalInterface

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

```
public static void main(String[] args) {  
    Function<String, Integer> toInteger = (s) -> Integer.valueOf(s);  
    Function<Integer, String> toString = (i) -> String.valueOf(i);  
    Function<String, String> fromStringToString = toInteger.andThen(toString);  
  
    Integer i = toInteger.apply("69"); // 69  
    String s1 = toString.apply(69); // 69  
    String s2 = fromStringToString.apply("69"); // 69  
}
```

Supplier<T>

Dostawcy nie przyjmują argumentów oraz zwracają wynik o wskazanym typie T.

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

public static void main(String[] args) {
    Supplier<Car> carSupplier = Car::new;
    Car newCar = carSupplier.get(); // new Car object
}
```

Consumer<T>

Konsumenci przyjmują jeden argument typu T i reprezentują operacje, jakie mają zostać na nim wykonane.

@FunctionalInterface

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

```
public static void main(String[] args) {  
    Consumer<String> mailFotterPrinter = (s) -> System.out.println("Pozdrawiam, " + s);  
    mailFotterPrinter.accept("Joanna"); // Pozdrawiam, Joanna  
}
```

Comparator<T>

Komparatory przyjmują dwa argumenty typu T i zwracają wartość liczbową.

@FunctionalInterface

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

```
public static void main(String[] args) {  
    Comparator<String> indexComparator = (s1, s2) -> s1.indexOf(s2);  
    Comparator<String> indexReverseComparator = indexComparator.reversed();  
  
    String string1 = "abcd";  
    String string2 = "bc";  
  
    indexComparator.compare(string1, string2); // 1  
    indexReverseComparator.compare(string1, string2); // -1  
}
```

Interfejsy funkcyjne – ćwiczenie (2)

W klasie "Java8Demo" użyj wbudowanego interfejsu funkcyjnego "Function<T, R>" zamiast "MathOperation".

Interfejs MathOperation można usunąć. 

Użyj także wbudowanego interfejsu funkcyjnego "Consumer<T>" do wyświetlania wyników (jako kolejny parametr prywatnej metody „getResult”).

3. Referencje do metod

Zastosowanie referencji

W Java SE w wersji 8 referencje do metod i konstruktorów mogą być przekazywane za pomocą słowa kluczowego ::.

Służą one do wskazywania metod według ich nazw.

Referencje mogą być użyte do następujących typów metod:

- metoda statyczna *Integer::valueOf*
- metoda instancji (obiekt) *objectName::toString*
- metoda instancji (klasa) *Object::toString*
- konstruktor *Object::new*

Referencja do metody statycznej

```
public class Person {  
  
    private Integer id;  
    private String name;  
  
    public Person(Integer id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
  
    private static int compareByName(Person a, Person b) {  
        return a.name.compareTo(b.name);  
    }  
  
    public static void main(String[] args) {  
        Person[] persons = new Person[2];  
        persons[0] = new Person(1, "Jan");  
        persons[1] = new Person(2, "Agnieszka");  
  
        Arrays.sort(persons, Person::compareByName);  
  
        System.out.println(persons[0].id + " " + persons[0].name); // 2 Agnieszka  
        System.out.println(persons[1].id + " " + persons[1].name); // 1 Jan  
    }  
}
```


Referencja do metody instancji (obiekt)

```
public class Person {  
    private Integer id;  
    private String name;  
  
    public Person(Integer id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public static void main(String[] args) {  
        Person[] persons = new Person[2];  
        persons[0] = new Person(1, "Jan");  
        persons[1] = new Person(2, "Agnieszka");  
  
        ComparisonProvider cp = new ComparisonProvider();  
        Arrays.sort(persons, cp::compareByName);  
  
        System.out.println(persons[0].id + " " + persons[0].name); // 2 Agnieszka  
        System.out.println(persons[1].id + " " + persons[1].name); // 1 Jan  
    }  
}
```

```
public class ComparisonProvider {  
  
    public int compareByName(Person a, Person b) {  
        return a.getName().compareTo(b.getName());  
    }  
}
```



Referencja do metody instancji (klasa)

```
public class SortString {  
    public static void main(String[] args) {  
        String[] persons = { "Jan", "Agnieszka"};  
  
        Arrays.sort(persons, String::compareTo);   
  
        System.out.println(persons[0]); // Agnieszka  
        System.out.println(persons[1]); // Jan  
    }  
}
```

```
// Metoda z klasy String  
public int compareTo(String anotherString) {  
    int len1 = value.length;  
    int len2 = anotherString.value.length;  
    int lim = Math.min(len1, len2);  
    char v1[] = value;  
    char v2[] = anotherString.value;  
  
    int k = 0;  
    while (k < lim) {  
        char c1 = v1[k];  
        char c2 = v2[k];  
        if (c1 != c2) {  
            return c1 - c2;  
        }  
        k++;  
    }  
    return len1 - len2;  
}
```

Referencja do konstruktora

```
public class Person {  
  
    private Integer id;  
    private String name;
```

```
    public Person() {}
```

```
    public Person(Integer id, String name) {  
        this.id = id;  
        this.name = name;  
    }
```

```
    public static void main(String[] args) {  
        PersonFactory<Person> personFactory = Person::new;  
        Person person = personFactory.create(1, "Martyna");  
  
        System.out.println(person.id + " " + person.name); // 1 Martyna  
    }  
}
```

```
public interface PersonFactory<P extends Person> {  
    P create(Integer id, String name);  
}
```



Referencje do metod – ćwiczenie (3)

W klasie "Java8Demo" zastosuj referencję do metody max() i min().

W klasie tej utwórz także 2 prywatne i statyczne metody "printMax" i "printMin".
Obie przyjmują parametr typu Integer i nie zwracają żadnej wartości (void).
Przenieś do tych metod instrukcję, która wyświetla "Wynik działania metody = X".
Zmodyfikuj w każdej z nich zwracany napis tak, aby się różnił.
Np. "Wynik działania metody max = X", "Wynik działania metody min = X".

Na koniec zastosuj referencję także do metody printMax() i printMin().

4. Metody default i static w interfejsach


Zastosowanie metod default

Od Java SE w wersji 8 istnieje możliwość definiowania metod *default* w interfejsach:

- słowo kluczowe *default*
- umożliwiają one dodanie nowych funkcjonalności do interfejsów przy jednoczesnym zachowaniu kompatybilności z kodem napisanym przy życiu poprzedniej wersji interfejsu
- klasa implementująca interfejs nie musi implementować metody *default*
- klasa implementująca interfejs może użyć metodę *default* lub ją nadpisać

Użycie metody default

```
public interface Report {  
    String getName();  
  
    default String getReportCode() {  
        return "ISA";  
    }  
}  
  
public class ReportDemo implements Report {  
  
    @Override  
    public String getName() {  
        return "Report for demo: " + getReportCode();  
    }  
  
    public static void main(String[] args) {  
        ReportDemo reportDemo = new ReportDemo();  
        System.out.println(reportDemo.getName()); // Report for demo: ISA  
    }  
}
```



Wiele metod default

Jeżeli klasa implementuje kilka interfejsów, które mają metodę *default* o tej samej nazwie:

- błąd podczas kompilacji tej klasy, gdy metoda *default* zostanie wykorzystana

Możliwe rozwiązania problemu:

- nadpisanie w klasie problematycznej metody
- użycie metody *default* z wybranego interfejsu przy pomocy słowa kluczowego *super*

Nadpisanie metody default

```
public interface Report {  
    String getName();
```

```
    default String getReportCode() {  
        return "ISA";  
    }  
}
```

```
public interface MobileReport {  
    String getName();
```

```
    default String getReportCode() {  
        return "ISA-Mobile";  
    }  
}
```

```
public class MultiReportDemo implements Report, MobileReport {
```

```
    @Override
```

```
    public String getName() {  
        return "Report for demo: " + getReportCode();  
    }
```

```
    @Override
```

```
    public String getReportCode() {  
        return "ISA-Multi";  
    }
```



```
    public static void main(String[] args) {  
        MultiReportDemo reportDemo = new MultiReportDemo();  
        System.out.println(reportDemo.getName()); // Report for demo: ISA-Multi  
    }  
}
```

Metoda default z wybranego interfejsu

```
public interface Report {  
    String getName();
```

```
    default String getReportCode() {  
        return "ISA";  
    }  
}
```

```
public interface MobileReport {  
    String getName();
```

```
    default String getReportCode() {  
        return "ISA-Mobile";  
    }  
}
```

```
public class MultiMobileReportDemo implements Report, MobileReport {
```

```
    @Override
```

```
    public String getName() {  
        return "Report for demo: " + getReportCode();  
    }
```

```
    @Override
```

```
    public String getReportCode() {  
        return MobileReport.super.getReportCode();  
    }
```



```
    public static void main(String[] args) {  
        MultiMobileReportDemo reportDemo = new MultiMobileReportDemo();  
        System.out.println(reportDemo.getName()); // Report for demo: ISA-Mobile  
    }  
}
```

Zastosowanie metod static

Od Java SE w wersji 8 istnieje możliwość definiowania także metod *static* w interfejsach:

- słowo kluczowe *static*
- metoda statyczna jest powiązana z interfejsem, w którym jest zdefiniowana, a nie z obiektem klasy implementującym ten interfejs
- pozwalają rezygnować z tzw. klas narzędziowych
- każda instancja klasy implementująca dany interfejs dzieli między sobą metodę statyczną z interfejsu
- klasa implementująca interfejs może użyć metodę statyczną

Użycie metody static

```
public interface Report {
    String getName();

    default String getReportCode() {
        return "ISA";
    }

    static void generateReportFile() {
        System.out.println("Raport został wygenerowany!");
    }
}

public class ReportDemo implements Report {

    @Override
    public String getName() {
        return "Report for demo: " + getReportCode();
    }

    public static void main(String[] args) {
        ReportDemo reportDemo = new ReportDemo();
        System.out.println(reportDemo.getName()); // Report for demo: ISA
        Report.generateReportFile(); // Raport został wygenerowany!
    }
}
```

Metody default i static w interfejsach – ćwiczenie (4)

Obok klasy "Java8Demo" stwórz interfejs "MathResults".
Przenieś do niego statyczne metody "printMin" i "printMax" z klasy "Java8Demo".
W klasie "Java8Demo" użyj powyższych metod z interfejsu.

Ponadto, dodaj w tym interfejsie nową metodę "getIntegersForDemo" i oznacz ją jako "default".
Metoda ta nie przyjmuje żadnych parametrów.
Niech zwraca ona dane testowe (liczby), które zadeklarowane były w metodzie "main".

W klasie "Java8Demo" użyj powyższej metody i usuń deklarację danych testowych.
Zwróć uwagę na to, że nie jest to metoda statyczna.

5. Klasa Optional<T>

Zastosowanie klasy `Optional<T>`

Optional jest to klasa, która pomaga uniknąć wystąpienia wyjątku `NullPointerException`:

- `java.util.Optional<T>`
- jest używana, gdy ma być zwracany wynik inny niż *null*, a czasami wynik ma nie zwracać niczego
- posiada różne metody do obsługi wartości, które są „dostępne” lub „niedostępne”
- odejście od sprawdzania warunku czy wartość jest *null*
- podobieństwo z *Optional* w *Guava*

Użycie klasy Optional<T>

```
public static void main(String[] args) {  
    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("empty")); // "isa"  
  
    String str2 = null;  
    Optional<String> optional2 = Optional.ofNullable(str2);  
    System.out.println(optional2.isPresent()); // false  
    System.out.println(optional2.orElse("empty")); // "empty"  
}
```

Metody klasy `Optional<T>`

- *boolean **isPresent()*** - zwraca *true* jeżeli wartość jest „dostępna”
- *T **get()*** - zwraca wartość jeżeli jest ona „dostępna”, w przeciwnym wypadku jest wyjątek *NoSuchElementException*
- *static <T> Optional<T> **of**(T value)* – zwraca *Optional* z podaną wartością, która jest różna od *null*
- *static <T> Optional<T> **ofNullable**(T value)* – zwraca *Optional* z podaną wartością lub pusty *Optional* jeżeli wartość jest *null*

Metody klasy `Optional<T>`

- `T orElse(T other)` – zwraca wartość jeżeli jest ona „dostępna”, w przeciwnym wypadku zwraca wartość *other*
- `static <T> Optional<T> empty()` - zwraca pustą instancję *Optional*
- `<U>Optional<U> map(Function<? super T,? extends U> mapper)` – stosuje podaną metodę *mapper* do wartości jeżeli jest ona „dostępna”
- `Optional<T> filter(Predicate<? super <T> predicate)` – zwraca *Optional* z wartością jeżeli jest ona „dostępna” oraz pasuje ona do podanej metody *predicate*

Klasa Optional<T> - ćwiczenie (5)

W interfejsie "MathResults" ustaw wartość zwracaną "null" dla metody "getIntegersForDemo".

Zwróć uwagę, że kompilator przy uruchamianiu zgłasza wyjątek "NullPointerException".

W klasie "Java8Demo" użyj metody „ofNullable” z klasy "Optional" dla zmiennej z danymi testowymi.

Wykorzystaj metodę "isPresent()" z klasy "Optional" tak, aby przy uruchamianiu nie pojawiał się wyjątek.

Na koniec można nadpisać metodę "getIntegersForDemo" w klasie "Java8Demo" tak, aby zwracała jakieś liczby.

6. API Date/Time

Zastosowanie API Date/Time

Java SE w wersji 8 zawiera zupełnie nowy interfejs API do obsługi daty i czasu:

- pakiet *java.time*
- wzorowane na popularnej bibliotece *Joda-Time*
- zawiera metody, które ułatwiają manipulowanie datą i czasem
- łatwiejsze operowanie strefami czasowymi
- jest *Immutable* – *final*, nie posiada metod typu *setter*

LocalDate

- zawiera tylko datę, np. 2017-01-02
- nie zawiera strefy czasowej

// Data bieżąca

```
LocalDate today = LocalDate.now(); // 2017-04-09
```

```
LocalDate tomorrow = today.plusDays(1); // 2017-04-10
```

```
LocalDate twoYearsAgo = today.minusYears(2); // 2015-04-09
```

```
LocalDate threeMonthsLater = today.plus(3, ChronoUnit.MONTHS); // 2017-07-09
```

// Data zdefiniowana

```
LocalDate definedDate = LocalDate.of(2017, Month.MARCH, 1); // 2017-03-01
```

```
Month month = definedDate.getMonth(); // MARCH
```

```
int year = definedDate.getYear(); // 2017
```

```
LocalDate newDate = definedDate.withDayOfYear(1); // 2017-01-01
```

// Formatowanie i parsowanie

```
DateTimeFormatter shortFormatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
```

```
String formattedDate = definedDate.format(shortFormatter); // 01.03.17
```

```
LocalDate thatDate = LocalDate.parse("24.12.17", shortFormatter); // 2017-12-24
```

LocalTime

- zawiera tylko czas, np. 12:45:59
- nie zawiera strefy czasowej

// Czas bieżący

```
LocalTime now = LocalTime.now(); // 13:42:07.592
```

```
LocalTime oneHourLater = now.plusHours(1); // 14:42:07.592
```

```
LocalTime tenMinutesAgo = now.minusMinutes(10); // 13:32:07.592
```

```
LocalTime fiveSecondsLater = now.plus(5, ChronoUnit.SECONDS); // 13:42:12.592
```

// Czas zdefiniowany

```
LocalTime definedTime = LocalTime.of(12, 45, 59); // 12:45:59
```

```
int hour = definedTime.getHour(); // 12
```

```
int seconds = definedTime.getSecond(); // 59
```

```
LocalTime newTime = definedTime.withHour(11); // 11:45:59
```

// Formatowanie i parsowanie

```
DateTimeFormatter shortFormatter = DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
```

```
String formattedTime = definedTime.format(shortFormatter); // 12:45
```

```
LocalTime thatTime = LocalTime.parse("05:55", shortFormatter); // 05:55
```

LocalDateTime

- zawiera datę i czas, np. 2017-01-02 12:45:59
- nie zawiera strefy czasowej

// Data i czas bieżący

```
LocalDateTime now = LocalDateTime.now(); // 2017-04-09T13:43:58.579
LocalDateTime oneHourLater = now.plusHours(1); // 2017-04-09T14:43:58.579
LocalDateTime twoYearsAgo = now.minusYears(2); // 2015-04-09T13:43:58.579
LocalDateTime fiveMonthsLater = now.plus(5, ChronoUnit.MONTHS); // 2017-09-09T13:43:58.579
```

// Data i czas zdefiniowany

```
LocalDateTime definedDateTime = LocalDateTime.of(2017, Month.MARCH, 1, 12, 45, 59); // 2017-03-01T12:45:59
int hour = definedDateTime.getHour(); // 12
Month month = definedDateTime.getMonth(); // MARCH
LocalDateTime newDateTime = definedDateTime.withDayOfYear(1); // 2017-01-01T12:45:59
```

// Formatowanie i parsowanie

```
DateTimeFormatter myFormatter = DateTimeFormatter.ofPattern("MM dd, yyyy - HH:mm");
String formattedDateTime = definedDateTime.format(myFormatter); // 03 01, 2017 - 12:45
LocalDateTime thatDateTime = LocalDateTime.parse("10 30, 2016 - 05:55", myFormatter); // 2016-10-30T05:55
```

Clock

- dostęp do aktualnej daty i czasu
- uwzględnia strefę czasową
- stosowany zamiast *System.currentTimeMillis()*

```
Clock clock = Clock.systemDefaultZone();  
long millis = clock.millis(); // 1491734943778
```

```
ZoneId zone1 = clock.getZone(); // Europe/Warsaw  
ZoneId zone2 = ZoneId.of("Europe/Warsaw"); // Europe/Warsaw
```

```
Instant instant = clock.instant();  
ZoneOffset zoneOffset = zone1.getRules().getStandardOffset(instant);  
System.out.println(zoneOffset); // +01:00
```

```
System.out.println(ZoneId.getAvailableZoneIds()); // zwraca wszystkie strefy czasowe
```

ZonedDateTime

- zawiera datę i czas, np. 2017-01-02 12:45:59
- zawiera strefę czasową

// Data i czas bieżący

```
ZoneId myZone = ZoneId.systemDefault(); // Europe/Warsaw
```

```
ZonedDateTime now = ZonedDateTime.now(myZone); // 2017-04-09T13:23:48.613+02:00[Europe/Warsaw]
```

```
ZonedDateTime oneHourLater = now.plusHours(1); // 2017-04-09T14:23:48.613+02:00[Europe/Warsaw]
```

```
ZonedDateTime twoYearsAgo = now.minusYears(2); // 2015-04-09T13:23:48.613+02:00[Europe/Warsaw]
```

// Data, czas i strefa zdefiniowana

```
ZoneId zone2 = ZoneId.of("America/El_Salvador"); // America/El_Salvador
```

```
ZonedDateTime definedDateTime =
```

```
    ZonedDateTime.of(2017, 3, 1, 12, 45, 59, 0, zone2); // 2017-03-01T12:45:59-06:00[America/El_Salvador]
```

```
int hour = definedDateTime.getHour(); //12
```

```
Month month = definedDateTime.getMonth(); // MARCH
```

ChronoUnit

- jest to enum *java.time.temporal.ChronoUnit*
- używany do reprezentacji dni, miesięcy, itp.

```
LocalDateTime now = LocalDateTime.now();
```

```
System.out.println(now.plus(60, ChronoUnit.SECONDS)); // + 60 sekund  
System.out.println(now.minus(12, ChronoUnit.MONTHS)); // - 12 miesięcy  
System.out.println(now.plus(2, ChronoUnit.DECADES)); // + 20 lat  
System.out.println(now.plus(1, ChronoUnit.CENTURIES)); // + 100 lat
```

Period & Duration

Klasy do obsługi określonej ilości czasu

- *Period* – stosowana do okresu czasu, określonego datą
- *Duration* – stosowana do okresu czasu, określonego czasem

```
LocalDate today = LocalDate.now(); // 2017-04-09
Period period1 = Period.of(1, 1, 1);
System.out.println(today.plus(period1)); // 2018-05-10
```

```
LocalDate newDate = today.plusYears(5); // 2022-04-09
Period period2 = Period.between(today, newDate);
System.out.println(period2.getYears()); // 5 lat
```

```
LocalTime now = LocalTime.now(); // 12:45
Duration duration1 = Duration.of(10, ChronoUnit.MINUTES);
System.out.println(now.plus(duration1)); // 12:55
```

```
LocalTime newTime = now.plusHours(2); // 14:45
Duration duration2 = Duration.between(now, newTime);
System.out.println(duration2.getSeconds()); // 7200 sekund
```

TemporalAdjusters

- klasa używana do wykonywania obliczeń na datach
- np. wylicz "drugi piątek miesiąca"

```
LocalDate now = LocalDate.now(); // 2017-04-09
```

```
LocalDate previousSaturday = now.with(TemporalAdjusters.previous(DayOfWeek.SATURDAY));  
System.out.println(previousSaturday); // 2017-04-08
```

```
LocalDate firstFriday = now.with(TemporalAdjusters.firstInMonth(DayOfWeek.FRIDAY));  
System.out.println(firstFriday); // 2017-04-07
```

```
LocalDate secondFriday = now.with(TemporalAdjusters.firstInMonth(DayOfWeek.FRIDAY))  
    .with(TemporalAdjusters.next(DayOfWeek.FRIDAY));  
System.out.println(secondFriday); // 2017-04-14
```


API Date/Time – ćwiczenie (6)

W klasie "Java8Demo" zmodyfikuj metodę "getIntegersForDemo" tak, aby zwracała np. 10_000_000 dowolnych liczb.

Do interfejsu "MathResults" dodaj statyczną metodę "printStartTime", która przyjmuje parametr typu "ZonedDateTime" i nie zwraca żadnej wartości.

Metoda ta ma za zadanie wyświetlić "Data uruchomienia operacji = X", gdzie data będzie wyświetlona przy użyciu "DateTimeFormatter" (FormatStyle.FULL).

Do interfejsu dodaj także statyczną metodę "printDuration", która przyjmuje parametr typu "Duration" i nie zwraca żadnej wartości. Metoda ma za zadanie wyświetlić "Czas wykonania operacji (sekund) = X", gdzie podana będzie liczba sekund.

Przy użyciu klasy "Clock" wylicz czas od uruchomienia do zakończenia działania metody "main". Powyższy rezultat wyświetl używając metody "printDuration" z interfejsu. Ponadto, używając metody z interfejsu "printStartTime" na początku metody "main", wyświetl datę uruchomienia aplikacji.

7. Strumienie

Zastosowanie strumieni

Strumień jest nową warstwą w Java SE 8 do przetwarzania danych:

- strumień stanowi sekwencję obiektów danego typu, na których mogą zostać wykonane operacje takie, jak np. filtrowanie, mapowanie, ograniczanie, zmniejszanie, znajdowanie, itp.
- strumień jest tworzony na podstawie źródła kolekcji takich, jak np. *List*, *Set*, *Arrays*, *I/O*
- umożliwia przetwarzanie danych w sposób deklaratywny
- pozwala odejść od pętli i ciągłego sprawdzania warunków
- wykorzystuje architekturę procesorów wielordzeniowych

Operacje strumieniowe

Operacje strumieniowe wykonują iteracje wewnętrznie na podanej kolekcji źródłowej.

Operacje strumieniowe dzielą się na:

- operacje kończące zwracają wynik określonego typu
- operacje pośrednie zwracają sam strumień, co pozwala połączyć w łańcuch kilka metod

Operacje strumieniowe mogą być wykonywane:

- sekwencyjnie: *stream()*
- równolegle: *parallelStream()*

Przykład użycia

Posortowanie listy z imionami oraz przefiltrowanie uwzględniając jedynie „poprawne” imiona.

```
List<String> names = Arrays.asList("Jan", "", "Wioletta", null, "Maria");  
List<String> filtered =  
    names.stream()  
        .filter(s -> s != null && !s.isEmpty())  
        .sorted()  
        .collect(Collectors.toList());  
System.out.println(filtered); // Jan, Maria, Wioletta
```

Collectors

Kolektory są używane do łączenia i zwrócenia wyniku po przetworzeniu elementów strumienia. Kolektory mogą być użyte do zwracania wyników o typie *List* lub *String*.

```
String[] names1 = {"Wioletta", "Maria"};  
Stream<String> stream = Stream.of(names1);  
List<String> list = stream.collect(Collectors.toList());  
System.out.println(list); // [Wioletta, Maria]  
  
List<String> names2 = Arrays.asList("Jan", "Tomasz");  
String str = names2.stream().collect(Collectors.joining("; "));  
System.out.println(str); // Jan; Tomasz
```

Metoda forEach

Metoda wykonuje iterację po każdym elemencie strumienia i wywołuje wskazaną instrukcję.

```
List<String> names = Arrays.asList("Jan", "Wioletta", "Maria");  
names.forEach(s -> {String dots = "..."; System.out.println(s.concat(dots));});  
// Jan...  
// Wioletta...  
// Maria...
```

Metoda map

Metoda jest używana do mapowania każdego elementu do odpowiedniego wyniku.

```
List<Integer> numbers = Arrays.asList(8, 3, 5);  
List<Integer> mapped =  
    numbers.stream()  
        .map(i -> 2 * i)  
        .collect(Collectors.toList());  
System.out.println(mapped); // 16, 6, 10
```


Metoda filter

Metoda jest używana do usunięcia elementów ze strumienia na podstawie podanego kryterium.

```
List<String> names = Arrays.asList("Jan", "Wioletta", "Maria");  
List<String> filtered =  
    names.stream()  
        .filter(s -> s.contains("i"))  
        .collect(Collectors.toList());  
System.out.println(filtered); // Wioletta, Maria
```

Metoda match

Metoda jest używana do sprawdzenia czy dany predykat odpowiada strumieniowi. Metoda zwraca wartość logiczną.

```
List<String> names = Arrays.asList("Jan", "Wioletta", "Maria");
```

```
boolean allMatched =  
    names.stream()  
        .allMatch(s -> s.contains("i"));  
System.out.println(allMatched); //false
```

```
boolean anyMatched =  
    names.stream()  
        .anyMatch(s -> s.contains("i"));  
System.out.println(anyMatched); //true
```

Metoda limit

Metoda jest używana do zredukowania liczby elementów zawartych w strumieniu.

```
List<String> names = Arrays.asList("Jan", "Wioletta", "Maria");  
List<String> filtered =  
    names.stream()  
        .limit(2)  
        .collect(Collectors.toList());  
System.out.println(filtered); // Jan, Wioletta
```

Metoda count

Metoda jest używana do wyliczenia ilości elementów w strumieniu.

```
List<String> names = Arrays.asList("Jan", "Wioletta", "Maria");  
long count =  
    names.stream()  
        .filter(s -> s.contains("i"))  
        .count();  
System.out.println(count); // 2
```

Statystyki

Umożliwiają wyliczanie różnych statystyk po tym, jak przetwarzanie strumienia dobiegło końca.

```
List<Integer> numbers = Arrays.asList(8, 3, 5);
IntSummaryStatistics stats =
    numbers.stream()
        .mapToInt(x -> 2 * x)
        .summaryStatistics();

System.out.println(stats.getMax()); // 16
System.out.println(stats.getMin()); // 6
System.out.println(stats.getSum()); // 32
System.out.println(stats.getAverage()); // 10.66666666
```

Strumienie równoległe

Operacje na strumieniach równoległych wykonywane są w kilku wątkach jednocześnie:

- operacje równoległe są szybsze
- wystarczy użyć *parallelStream()* zamiast *stream()*

```
List<String> names = Arrays.asList("Jan", "", "Wioletta", null, "Maria");  
List<String> filtered =  
    names.parallelStream()  
        .filter(s -> s != null && !s.isEmpty())  
        .sorted()  
        .collect(Collectors.toList());  
System.out.println(filtered); // [Jan, Maria, Wioletta]
```

Strumienie – ćwiczenie (7)

Niech metoda "getIntegersForDemo" w interfejsie "MathResults" zwraca listę dowolnych 10_000_000 liczb.

W metodzie "getIntegersForDemo" klasy "Java8Demo" odwołaj się do metody nadpisywanej (słowo kluczowe `super`) i pobierz powyższe dane testowe z interfejsu.

Następnie użyj strumieni w metodzie nadpisującej tak, aby każdy element z danych testowych przemnożyć przez -1 oraz usunąć z nich wszystkie liczby parzyste.

Podsumowanie



Podsumowanie

1. Wyrażenia lambda



Podsumowanie

1. Wyrażenia lambda
2. Interfejsy funkcyjne



Podsumowanie

1. Wyrażenia lambda
2. Interfejsy funkcyjne
3. Referencje do metod



Podsumowanie

1. Wyrażenia lambda
2. Interfejsy funkcyjne
3. Referencje do metod
4. Metoda default i static



Podsumowanie

1. Wyrażenia lambda
2. Interfejsy funkcyjne
3. Referencje do metod
4. Metoda default i static
5. Klasa Optional<T>



Podsumowanie

1. Wyrażenia lambda
2. Interfejsy funkcyjne
3. Referencje do metod
4. Metoda default i static
5. Klasa Optional<T>
6. API Date/Time



Podsumowanie

1. Wyrażenia lambda
2. Interfejsy funkcyjne
3. Referencje do metod
4. Metoda default i static
5. Klasa Optional<T>
6. API Date/Time
7. Strumienie



Podsumowanie

1. Wyrażenia lambda
2. Interfejsy funkcyjne
3. Referencje do metod
4. Metoda default i static
5. Klasa Optional<T>
6. API Date/Time
7. Strumienie

<http://docs.oracle.com/javase/tutorial/>





Dziękuję :)

Rafał Kurt

Senior Software Developer

rafalkurt@gmail.com

<https://github.com/rkurt/solwit-java8.git>