Wyrażenia lambda
- naturalna ewolucja języka Java

Programowanie funkcyjne

Filozofia i metodyka programowania będąca odmianą programowania deklaratywnego, w której funkcje należą do wartości podstawowych, a nacisk kładzie się na wartościowanie funkcji (często rekurencyjnych), a nie wykonywanie poleceń. W czystym programowaniu funkcyjnym, raz zdefiniowana funkcja zwraca zawszę tą samą wartość dla danych wartości argumentów, tak jak w matematyce.

Źródło: http://pl.wikipedia.org/wiki/Programowanie_funkcyjne

Czy opłaca się programować funkcyjnie w Javie?

Nowe sposoby budowania abstrakcji

Czysty, czytelny i bardziej bezpieczny kod

Nowe, efektywne api istniejących bibliotek

Wyrażenia lambda

Kompaktowy sposób definiowania i przekazywania zachowania

Przykład z użyciem wyrażenia lambda

```
JButton button = new JButton();
button.addActionListener(event -> System.out.println("clicked"));
```

Ten sam przykład z klasą anonimową

```
JButton button = new JButton();
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("clicked");
    }
});
```

Wariacje zapisu wyrażeń lambda

Wyrażenie bez argumentów

```
() -> System.out.println("Hello lambda expressions");
```

Wyrażenie z jednym argumentem

```
event -> System.out.println("clicked");
```

Wyrażenie z wieloma argumentami

```
(x, y) -> x + y;
```

Wyrażenie z wieloma argumentami i wskazaniem typu

```
(Long x, Long y) -> x + y;
```

Wyrażenie bez argumentów i blokiem instrukcji

```
() -> {
    System.out.print("Hello lambda expressions");
};
```

Wykorzystanie zmiennych z poza bloku

Zmienne z poza bloku wyrażenia lambda, muszą być efektywnie finalne tzn. ich wartość może zostać określona tylko raz (analogicznie jak w przypadku zmiennych finalnych)

Użycie modyfikatora final jest opcjonalne

```
String name = "Jan";          // zmienna efektywnie finalna
JButton button = new JButton();
button.addActionListener(event -> System.out.println("Hello  " + name));
```

Interfejs funkcyjny (ang. functional interface)

Dowolny interfejs definiujący jedną metodę

Idiom umożliwiający utworzenie odwołania do wyrażenia lambda, określenie jego typu oraz nazwanie przekazywanych argumentów

Przykłady wbudowanych interfejsów funkcyjnych

Nazwa interfejsu	Argumenty	Typ zwracany
Predicate<T>	T	boolean
Consumer<T>	T	void
Function<T,R>	T	R
Supplier<T>		T
UnaryOperator<T>	T	T
BinaryOperator<T> (T, T)	(T, T)	T

Adnotacja @FunctionalInterface

Powinna być stosowana na poziomie interfejsów wykorzystywanych jako funkcyjne

Wymusza kontrolę poprawności interfejsu na poziomie kompilacji

Stanowi dodatkową informację o sposobie wykorzystania interfejsu

Referencje do metody (ang. method reference)

Pozwalają na alternatywny zapis wyrażeń lambda, w których dochodzi do wywołania metody na przekazanych argumentach

Mają postać NazwaKlasy::nazwaMetody lub NazwaKlasy:new dla konstruktora

Stanowią odwołanie do metody z wykorzystaniem jej nazwy

Przykłady

```
Order::getName  // równoważne do order -> order.getName()

Item::new  // równoważne do (name, price) ->  new Item(name, price)
```

Wnioskowanie typu (ang. type inference)

Typ wyrażenia lambda oraz przekazywanych do niego argumentów może zostać określony jawnie lub wyznaczony w czasie kompilacji (w większości przypadków)

Przykład jawnego zdefiniowania typu wyrażenia lambda

```
public interface Predicate<T> {
    boolean test(T t);
}
Predicate<Integer> atLeast100 = x -> x > 100;
```

Wnioskowanie typu, a przeciążanie metod

Kiedy metoda nie jest przeciążona, typ wyrażenia lambda wnioskowany jest na podstawie typu przyjmowanego argumentu interfejsu funkcyjnego

Kiedy metoda jest przeciążona wygrywa typ najbardziej wyspecjalizowany, a jeśli takiego nie ma musi on zostać jawnie wskazany

Wnioskowanie typu, a przeciążanie - przykład 1

```
private interface IntegerBinaryFunction extends BinaryOperator<Integer> {
}

private void someMethod(BinaryOperator<Integer> lambda) {
   // some code
}

private void someMethod(IntegerBinaryFunction lambda) {
   // some code
}

someMethod((x, y) -> x + y);
```

Wnioskowanie typu, a przeciążanie - przykład 2

```
private interface IntPredicate {
    public boolean test(int value);
}

private void someMethod(Predicate<Integer> predicate) {
    // some code
}

private void someMethod(IntPredicate predicate) {
   // some code
}

someMethod((x) -> true);  // Błąd kompilacji - wymaga jawnego rzutowania na IntPredicate lub Predicate<Integer>
```

Metody domyślne (ang. default methods)

Definiowane na poziomie interfejsów z użyciem słowa kluczowego default

Umożliwiają bezpieczne rozszerzanie istniejącego api o nowe metody (w taki sposób wprowadzono do api Javy metody wykorzystujące wyrażenia lambda)

Nadają nowe znaczenie interfejsom Javy

	Interfejsy	Klasy abstrakcyjne	Klasy
Pola statyczne	Tak	Tak	Tak
Pola instancyjne	Nie	Tak	Tak
Metody statyczne	Tak	Tak	Tak
Metody instancyjne	Nie	Tak	Tak
Wielodziedziczenie	Tak	Nie	Nie

Typ Optional

Nowy typ zaprojektowany jako elegancka alternatywa dla wartości null

Przykłady tworzenia i użycia typu Optional

```
Optional<String> text = Optional.of("Java lambdas");
if (text.isPresent()) {
    System.out.println(text.get());
}
System.out.println(text.orElse("Java 8"));
System.out.println(text.orElseGet(() -> "Java 8 is fun"));  // jeśli utworzenie   alternatywy  jest  kosztowne  można przekazać
                                                            // wyrażenie typu Supplier, wywoływane tylko w tedy jeśli wartość
                                                            // naprawdę nie istnieje
```

Iteracja i odczyt elementów listy - klasycznie

```
List<String> languages= Arrays.asList("Java", "PHP", "JavaScript", "Swift", "Python");
for(int i = 0; i < languages.size(); i++) {
    System.out.println(languages.get(i));
}
```

Iteracja i odczyt elementów listy - metoda forEach

```
List<String> languages= Arrays.asList("Java", "PHP", "JavaScript", "Swift", "Python");
languages.forEach(new Consumer<String>() {
    public void accept(String name) {
        System.out.println(name);
    }
});
```

Iteracja po elementach listy - metoda forEach lepiej

```
List<String> languages= Arrays.asList("Java", "PHP", "JavaScript", "Swift", "Python");
```

Zastosowanie wyrażenia lambda

```
languages.forEach((String name) -> System.out.println(name));
```

Zastosowanie referencji do metody

```
languages.forEach(System.out::println);
```

Iteracja i modyfikacja elementów listy - klasycznie

```
List<String> languages= Arrays.asList("Java", "PHP", "JavaScript", "Swift", "Python");
List<String> uppercaseNames = new ArrayList<>();
for(String name : languages) {
    uppercaseNames.add(name.toUpperCase());
}
```

Iteracja i modyfikacja elementów listy - funkcyjnie

```
List<String> languages= Arrayąs.asList("Java", "PHP", "JavaScript", "Swift", "Python");

List<String> uppercaseNames = new ArrayList<>();

languages.forEach(name -> uppercaseNames.add(name.toUpperCase()));
```

Iteracja, modyfikacja i wykorzystanie elementów listy

```
List<String> languages= Arrayąs.asList("Java", "PHP", "JavaScript", "Swift", "Python");
languages.stream()
    .map(name -> name.toUpperCase())
    .forEach(name -> System.out.print(name + " "));
```

Filtrowanie elementów listy

```
List<String> languages= Arrayąs.asList("Java", "PHP", "JavaScript", "Swift", "Python");
List<String> startsWithJava = languages.stream()
    .filter(name -> name.startsWithJava("Java"))
    .collect(Collectors.toList());
System.out.println(String.format("Found %d languages", startsWithJava.size()));
```

Duplikacja wyrażeń lambda

```
List<String> friends = Arrays.asList("Adam", "Olaf", "Martin");
List<String> familyMembers = Arrays.asList("Mike", "John", "Anna");
long countFriendsStartingWithA = friends.stream()
    .filter(name -> name.startsWith("A")).count();
long countFamilyMembersStartingWithA = familyMembers.stream()
    .filter(name -> name.startsWith("A")).count();
```

Reużywanie wyrażeń lambda

```
Predicate<String> startsWithA = name -> name.startsWith("A");
List<String> friends = Arrays.asList("Adam", "Olaf", "Martin");
List<String> familyMembers = Arrays.asList("Mike", "John", "Anna");
long countFriendsStartingWithA = friends.stream()
    .filter(startWithA).count();
long countFamilyMembersStartingWithA = familyMembers.stream()
    .filter(startWithA).count();
```

Wykorzystanie funkcji wyższego rzędu i domknięcia

Zapis pełny

```
Function<String, Predicate<String>> startsWithLetter = (String letter) -> {
    Predicate<String> checkStarts = (String name) -> name.startsWith(letter);
    return checkStarts;
};
```

Zapis skrócony - bez użycia typów

```
Function<String, Predicate<String>> startsWithLetter = letter -> name -> name.startsWith(letter);
```

Wykorzystanie funkcji wyższego rzędu i domknięcia c.d.

```
long countFriendsStartingWithA = friends.stream()
    .filter(startsWithLetter.apply("A")).count();
long countFriendsStartingWithB = friends.stream()
    .filter(startsWithLetter.apply("B")).count();
```

Pobieranie jednego elementu z kolekcji - klasycznie

```
public void pickName(List<String> names, String startingLetter) {
    String foundName = null;
    for(String name : names) {
        if (name.startsWith(startingLetter)) {
            foundName = name; break;
        }
    }
    System.out.print(String.format("A name starting with %s: ", startingLetter));
    if (foundName != null) { System.out.println(foundName); } else { System.out.println("No name found"); }
}
```

Pobieranie jednego elementu z kolekcji - funkcyjnie

```
public void pickName(List<String> names, String startingLetter) {
    Optional<String> foundName = names.stream()
        .filter(name ->name.startsWith(startingLetter))
        .findFirst();
    System.out.println(String.format("A name starting with %s: %s”, startingLetter, foundName.orElse("No name found")));
}
```

Wzorzec MapReduce

```
List<String> friends = Arrays.asList("Adam", "Olaf", "Martin");
System.out.println("Total number of characters in all names: " + friends.stream()
    .mapToInt(name -> name.length())
    .sum());
```

Inny przykład MapReduce

```
List<String> friends = Arrays.asList("Adam", "Olaf", "Martin");
Optional<String> result = friends.stream()
    .reduce((name1, name2) -> name1.length() >= name2.length() ? name1 : name2);
result.ifPresent(name ->System.out.println(String.format("A longest name: %s", name)));
```

Obsługa wyjątków

```
public static void main(String[] args) throws IOException {
    Stream.of("/home", "/opt”).map(path -> new File(path).getCanonicalPath())
        .forEach(System.out::println);  // Błąd kompilacji
}
```

Równoległe wykonanie kodu

```
public int sum(List<Integer> numbers) {
    return albums.parallelStream()
        .sum();
}
```

Refaktoryzacja pod kątem nowych elementów języka

Deklaratywny kod

Niezmienność i brak efektów ubocznych

Funkcje wyższego rzędu

Refaktoryzacja - przykład 1

Przed

```
Logger logger = new Logger();
if (logger.isDebugEnabled()) {
    logger.debug("Debug: " + buildMessage());
}
```

Po

```
Logger logger = new Logger();
logger.debug(() -> "Debug: " + buildMessage());
```

Refaktoryzacja - przykład 2

Przed

```
ThreadLocal<Client> thisClient = new ThreadLocal<> () {
    protected Client initialValue() {
        return database.lookupCurrentClient();
    }
};
```

Po

```
ThreadLocal<Client> thisClient = ThreadLocal.withInitial(() -> database.lookupCurrentClient());
```

Refaktoryzacja - przykład 3

Przed

```
public long countTotalTime() {
    long count = 0;
    for (Album album : albums) {
      for (Track track : album.getTrackList()) {
          count += track.getLength();
      }
    }
   return count;
}
```

Refaktoryzacja - przykład 3 c.d.

```
public long countTracks() {
    long count = 0;
    for (Album album : albums) {
        count += album.getTrackList().size();
    }
    return count;
}
```

Refaktoryzacja - przykład 3 c.d.

Po

```
    public long countFeature(ToLongFunction<Album> function) {
        return albums.stream().mapToLong(function).sum();
    }
 public long countTotalTime() {
        return countFeature(album -> album.getTracks().mapToLong(track -> track.getLength()).sum());
    }
    public long countTracks() {
        return countFeature(album -> album.getTracks().count());
    }
```

Debugowanie operacji wykonywanych na strumieniu

```
Set<String> nationalities = books.getAuthors().stream()
    .filter(author-> author.getName().startsWith("The"))
    .map(author-> author.getNationality())
    .peek(nation -> System.out.println("Found nationality: " + nation))
    .collect(Collectors.<String>toSet());
```