UNIWERSYTET IM. ADAMA MICKIEWICZA W POZNANIU
Wydział Matematyki i Informatyki

Programowanie funkcyjne

Ostatnim razem

Za Wikipedią:

> Funkcja (łac. functio, -onis „odbywanie, wykonywanie, czynność”) - dla danych dwóch zbiorów ***X*** i ***Y*** przyporządkowanie każdemu elementowi zbioru ***X*** dokładnie jednego elementu zbioru ***Y***.

Czy można tak programować, aby program był złożeniem wielu funkcji?

Do pewnego stopnia TAK.

W świecie Javy

Pakiet `java.util.function`:

- `Function<T,R>` – funkcja, przyjmuje T, zwraca R
- Predicate<T> – predykat (funkcja, która przyjmuje T, zwraca `Boolean`)
- Consumer<T> – konsument (funkcja, która przyjmuje T, nie zwraca nic [gdy potrzeba typu: `Void`])
- `Supplier<R>` – dostawca (funkcja bezargumentowa, zwraca R)
- UnaryOperator<T> – funkcja, która przyjmuje i zwraca T

W świecie Javy

Pakiet `java.util.function`:

- `BiFunction<T,U,R>` – funkcja dwuargumentowa, przyjmuje `T` i `U`, zwraca `R`
- `BiPredicate<T,U>` – predykat dwuargumentowy, przyjmuje `T` i `U`, zwraca `Boolean`
- `BiConsumer<T,U>` – konsument dwuargumentowy, przyjmuje `T` i `U`, nie zwraca nic
- `BinaryOperator<T>` – funkcja, która przyjmuje dwa argumenty typu `T` i zwraca `T`

W świecie Javy

Pakiet [java.util.function](), dla dociekliwych czym są:

- BooleanSupplier
- ToDoubleFunction<T>
- ToDoubleBiFunction<T>
- DoubleBinaryOperator
- DoubleConsumer
- DoubleFunction<R>
- DoublePredicate
- DoubleSupplier
- DoubleToIntFunction
- DoubleToLongFunction
- DoubleUnaryOperator
- ObjDoubleConsumer

(oraz Long zamiast Int)

- ToIntFunction<T>
- ToIntBiFunction<T>
- IntBinaryOperator
- IntConsumer
- IntFunction<R>
- IntPredicate
- IntSupplier
- IntToDoubleFunction
- IntToLongFunction
- IntUnaryOperator
- ObjIntConsumer

Pojęcie funktora

WIKIPEDIA
Wolna encyklopedia

Strona główna
Losuj artykuł
Kategorie artykułów

Funktor (teoria kategorii) [edytuj]

W teorii kategorii **funktor** to odwzorowanie jednej kategorii do drugiej zachowujące złożenia i morfizmy tożsamościowe[a]. Można o nim myśleć jako o homomorfizmie wyższego rzędu. Ważne jest rozróżnienie dwóch typów funktorów: kowariantnych i kontrawariantnych.

Pojęcie funktora

(nie tylko) W Javie są metody, które przyjmują wszelkiego rodzaju funkcje (te z poprzednich slajdów).

W jakiś sposób trzeba im przekazać kod do wykonania:

- poprzez wyrażenie **`lambda`**,
- albo przez referencję do metody (ang. method reference).

Wyrażenia lambda i referencje do metod będziemy nazywać funktorem, bo pełni rolę funkcji i może zostać wywołany.

Przykład – kod imperatywny

UglyImperativePrimes.java

```java
package pl.amu.edu.demo.primes;

import lombok.NoArgsConstructor;

@NoArgsConstructor
public class UglyImperativePrimes {

    public static void main(String[] args) {
        new UglyImperativePrimes().printPrimes();
    }

    public void printPrimes() {
        for (int i = 0; i < 121; i++) {
            if (isPrime(i)) {
                System.out.printf("%d is prime\n", i);
            }
        }
    }

    private boolean isPrime(int number) {
        if (number <= 0) {
            return false;
        }
        for (int i = 2; i <= (int) Math.sqrt(number); i++) {
            if (number % i == 0) {
                return false;
            }
        }
        return true;
    }

}
```

Run: UglyImperativePrimes

```
1 is prime
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
101 is prime
103 is prime
107 is prime
109 is prime
113 is prime
```

Przykład – kod funkcyjny

UglyImperativePrimes.java × FunctionalPrimes.java ×

```java
package pl.amu.edu.demo.primes;

import lombok.NoArgsConstructor;

import java.util.function.IntPredicate;
import java.util.stream.IntStream;

@NoArgsConstructor
public class FunctionalPrimes {

    public static void main(String[] args) {
        new FunctionalPrimes().printPrimes();
    }

    public void printPrimes() {
        IntStream.range(0, 121)
                .filter(this::isPrime)
                .forEach(i -> System.out.printf("%d is prime\n", i));
    }

    private boolean isPrime(int number) {
        if (number <= 0) {
            return false;
        }
        IntPredicate isDivisible = (int divisor) -> number % divisor == 0;
        return IntStream.rangeClosed(2, (int) Math.sqrt(number))
                .noneMatch(isDivisible);
    }

}
```

Run

Run: FunctionalPrimes ×

```
1 is prime
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
101 is prime
103 is prime
107 is prime
109 is prime
113 is prime
```

Map - Reduce

```
List<Person> people; // niechaj mamy dane wielu ludzi
var maxAge = people.stream()
        .mapToInt(Person::getAge)
        .reduce((age1, age2) -> max(age1, age2));
```

Idea: przetwarzamy (względnie) dużo danych i dla każdej pary redukujemy wynik do pojedynczej wartości. Na koniec dostajemy pojedynczą, zredukowaną wartość.

Ale: Rzadko jest potrzebne bezpośrednie wywołanie metody *reduce* na klasach `Stream`.

Wbudowane metody redukcji

Pakiet `java.util.stream`, klasy typu `IntStream`, `LongStream`, `DoubleStream` zawierają metody:

- `average()` – oblicza wartość średnią
- `max()` – znajduje wartość maksymalną
- `min()` – znajduje wartość minimalną
- `sum()` – oblicza sumę (uwaga na przepełnienie dla int!)
- `summaryStatistics()` – w zasadzie wszystko powyższe naraz
- `count()` – zliczanie elementów

Dla dociekliwych: co z tego (i jak tego użyć) zawiera zwykły `java.util.stream.Stream<T>`?

Co jeżeli lista jest pusta?

```
List<Person> people; // niechaj mamy dane wielu ludzi
var maxAge = people.stream()
        .mapToInt(Person::getAge)
        .max()

var averageAge = people.stream()
        .mapToInt(Person::getAge)
        .average()
```

Pusta lista? I co?

Wzorzec projektowy Maybe

Odpowiedzią są klasy (z pakietu java.util):

- `Optional<T>`
- `OptionalDouble`
- `OptionalInt`
- `OptionalLong`

Każda z nich może zawierać element (`isPresent() == true`) lub nie (`isEmpty() == true`).

UWAGA! Nie stosujemy ich jako pól w klasie!

Przykład – wykorzystanie Optional

```
private void run() {
    var person : Person = Person.builder()
            .displayName("Gąska Balbinka").birthDate(LocalDate.ofYearDay( year: 1959,  dayOfYear: 1)).build();
    var address : Address = Address.builder().build();
    var underTheBridge : Housing = Housing.builder().area(0.2).address(address).build();
    var maybeAdult : Optional<Person> = Optional.ofNullable(person)
            .filter(Person::isAdult);
    var housing : Housing = maybeAdult
            .map(Person::getHousing)
            .orElse(underTheBridge);
    var zipCode : String = Optional.of(housing) Optional<Housing>
            .flatMap(Housing::zipCode) Optional<String>
            .orElseThrow();
    maybeAdult
            .map(Person::getBirthDate) Optional<LocalDate>
            .map(birthDate -> {
                var name : String = person.displayName;
                return String.format("%s: born %s lives at %s area.", name, birthDate, zipCode);
            }) Optional<String>
            .ifPresent(System.out::println);
}
```

Dla dociekliwych – składanie

Co robią metody:

- Predicate::and
- Predicate::or
- Predicate::not
- Predicate::negate
- Function::compose
- Function::andThen
- (bonus) Predicate::isEqual

I jak z nich skorzystać (warto poszukać przykładów!)

Function::identity - zwraca to, co dostała – często używana.

java.util.stream.Stream

Co ważne:

- strumień da się przetworzyć **tylko jeden raz**. Jeżeli trzeba więcej razy, to trzeba zapisać (wynik pośredni?) w klasie `List`
- kod jest ewaluowany leniwie, tzn. żadna metoda `map()`, `filter()`, `flatMap()`, itd. nie wykona się dopóki nie wywołamy operacji terminalnej (takiej jak agregacja, redukcja, czy metody `collect(), forEach()`)
- w wyrażeniach lambda, predykatach i funkcjach możemy korzystać tylko ze zmiennych efektywnie finalnych (tzn. takich, które nie będą przypisane więcej niż raz w ciele metody, lub po prostu mają modyfikator **`final`**)

java.util.stream.Stream

Wracając do strumieni:

- za ich pomocą możemy konwertować dane np.

```
lista.stream()
     .filter(<predykat>)
     .map(<funkcja>)
     .collect(Collectors.toList())
```

- możemy rzecz jasna także przetwarzać inne klasy:

```
Map<String, List<Person>> bySurname = lista.stream()
    .collect(Collectors.groupingBy(Person::getDisplayName)
bySurname.entrySet().stream
    .map(…) // Map.Entry<String, List<Person>> -> ?
    .collect(…)
```

Przykład parallel stream (i Try)

```java
public class StreamExample {

    public static void main(String[] args) {
        new StreamExample().run();
    }

    private void run() {
        // Huge number of threads!
        getPeople().parallel().mapToInt(Person::getAge).average().ifPresent(System.out::println);
        // Better solution, limiting the number of threads
        try (var pool = new ForkJoinPool( parallelism: 8)) {
            var maybeAverageAgeFuture : ForkJoinTask<OptionalDouble> = pool.submit(
                    () -> getPeople().parallel().mapToInt(Person::getAge).average()
            );
            // io.vavr.control.Try
            var triedAverageAge : Try<Double> = Try.of(
                    () -> maybeAverageAgeFuture.get( timeout: 30, TimeUnit.SECONDS)
            ).map(OptionalDouble::orElseThrow);
            if (triedAverageAge.isSuccess()) { // there is also isFailure()
                System.out.println(triedAverageAge.get()); // instead of if, one can use getOrElse()
            }
        }

    }
```

parallel stream

Strumień może działać wielowątkowo (`.parallel()`), ale:

- uruchamia ogromną liczbę wątków (jeżeli danych jest dużo)
- dzieli każdy strumień na dwie części, potem część na dwie części, i dalej aż dojdzie do pojedynczego zadania. Potem składa wynik w całość łącząc pojedyncze wyniki, wyniki z kolejnych części, aż osiągnie pełny wynik. To przykład algorytmu Divide & Conquer (dziel i zwyciężaj). Podział zadań na wątki przypomina diament
- należy stosować metodę pokazaną na obrazku: musimy ograniczać liczbę wątków uruchamianych równolegle, bo inaczej… Kury przestaną się cielić, a krowy nieść (zabijemy maszynę, na której to uruchamiamy).

Funkcyjne Try

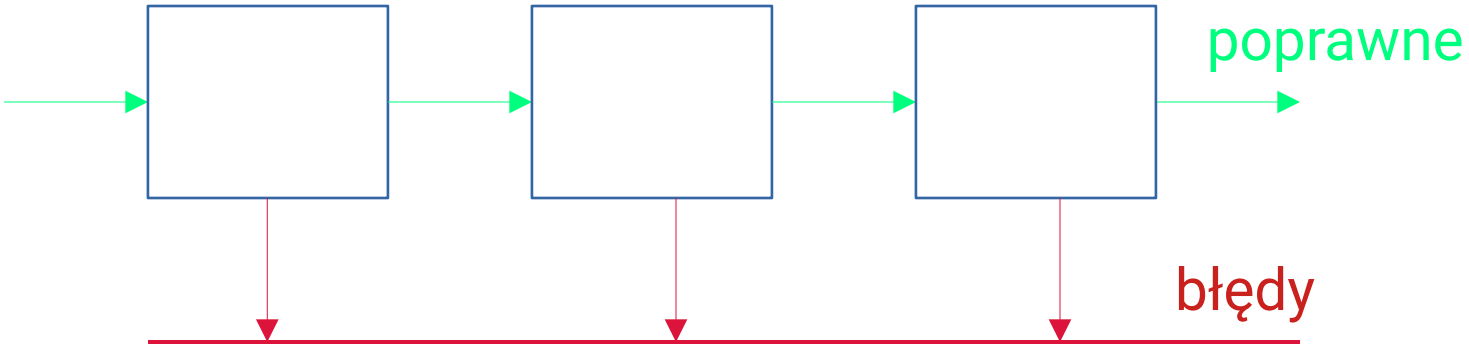
Zamiast pisać try...catch można użyć funkcyjnego Try:

- nie ma w bibliotece standardowej Javy (trochę błąd)
- występuje np. w języku Scala (i wielu innych)
- jest w bibliotece vavr.io (patrz Maven Central)
- wygląda o niebo lepiej, ale
- nie nadaje się do zarządzania zasobami (patrz: `try(var pool = ForkJoinPool(8))`)
- Java ma tzw. checked exceptions (gruby błąd), przez co Try musi łapać za dużo (`Throwable` w tym `OutOfMemoryError`) – to może spowodować bardzo poważne problemy

Railway Oriented Programming

Idea:

- Dwa “potoki” przetwarzania:
  - dla normalnego poprawnego kodu
  - dla błędów



Railway Oriented Programming

```java
public class EitherExample {

    public static void main(String[] args) {
        new EitherExample().run();
    }

    private void run() {
        var balbina : Person = Person.builder()
                .displayName("Gąska Balbinka").birthDate(LocalDate.ofYearDay( year: 1959, dayOfYear: 1)).build();
        var poziomka : Person = Person.builder()
                .displayName("magic Poziomka").build();
        var maybeBalbinasAge : Either<ProcessingError, Integer> = getAge(balbina);
        var maybePoziomkasAge : Either<ProcessingError, Integer> = getAge(poziomka);
        System.out.printf("Is Balbinas age empty? %s\n", maybeBalbinasAge.isEmpty());
        System.out.printf("Is Poziomkas age empty? %s\n", maybePoziomkasAge.isEmpty());
        System.out.printf("Balbinas age: %d\n", maybeBalbinasAge.get());
        System.out.printf("Poziomkas error: %s\n", maybePoziomkasAge.getLeft());
    }

    private Either<ProcessingError, Integer> getAge(Person person) {
        if (isNull(person.birthDate)) {
            // konwencja: błędy po lewej
            return Either.left(ProcessingError.of( errorMessage: "No birthdate!"));
        }
        return Either.right(person.getAge());
    }

}
```

Either

Either może mieć albo wartość lewą (konwencja: błędną), albo prawą (poprawne wykonanie):

- po lewej mogą być oczywiście pochodne Exception
- po prawej cokolwiek tam powinno być przy prawidłowym przebiegu
- Either także jest monoidem, ma filter(), map(), flatMap()
  - niekoniecznie powinno być tam przekazane Either…
  - ma jeszcze mapLeft(), itd. (to nie jest klasyczny monad)

https://docs.vavr.io/

Ćwiczenia

Ćwiczenia do prezentacji:

https://link.wmi.amu.edu.pl/aorCs1

Dziękuję za korporację
(thank you for your corporation)