Search . . .

06
Monády a functory

- Teorie kategorií
- Functory a monády
- Future/Promisy
- Řešení chyb a NPE

Next >

Objektové Modelování - podzim 2022

david.kadlecek@cvut.cz, www.czm.fel.cvut.cz

06 Teorie kategorií

Teorie kategorií zkoumá, jaké vlastnosti matematických struktur lze vyslovit či dokázat, pokud nebudeme mluvit o jejich prvcích, ale jen o zobrazeních (morfismech) mezi nimi a jejich skládání. Zobrazení, které prvku x přiřadí $g(f(x))$, značíme $g \circ f$ (oproti méně časté konvenci, která je značí $f \circ g$).

Teorie kategorií poskytuje sjednocující rámec, ve kterém lze studovat různé matematické struktury a jejich vztahy. Mnoho matematických disciplín lze interpretovat v rámci kategorií, což umožňuje:

- Identifikovat analogie mezi různými oblastmi matematiky.
- Zjednodušit důkazy pomocí abstrakce.
- Vyjádřit obecné vlastnosti struktur (např. funktory, přírodní transformace).

Kategorie množin

Objekty: Množiny.

Morfismy: Funkce mezi množinami.

Kategorie vektorových prostorů

Objekty: Vektorové prostory nad pevným tělesem (např. R nebo C).

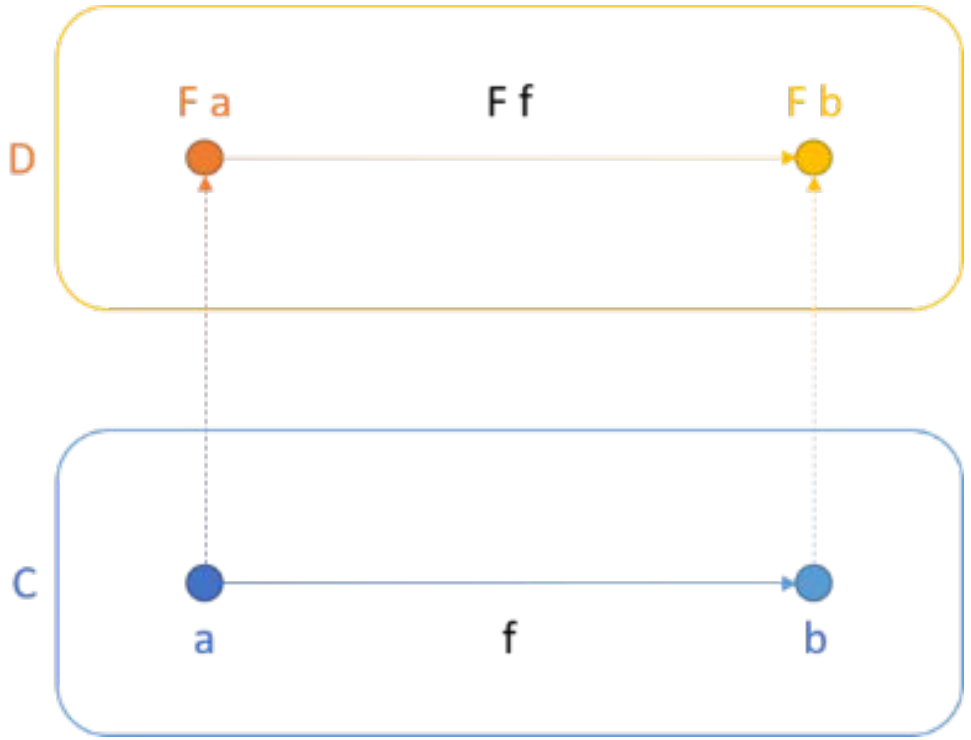
Morfismy: Lineární zobrazení.

06 Functor

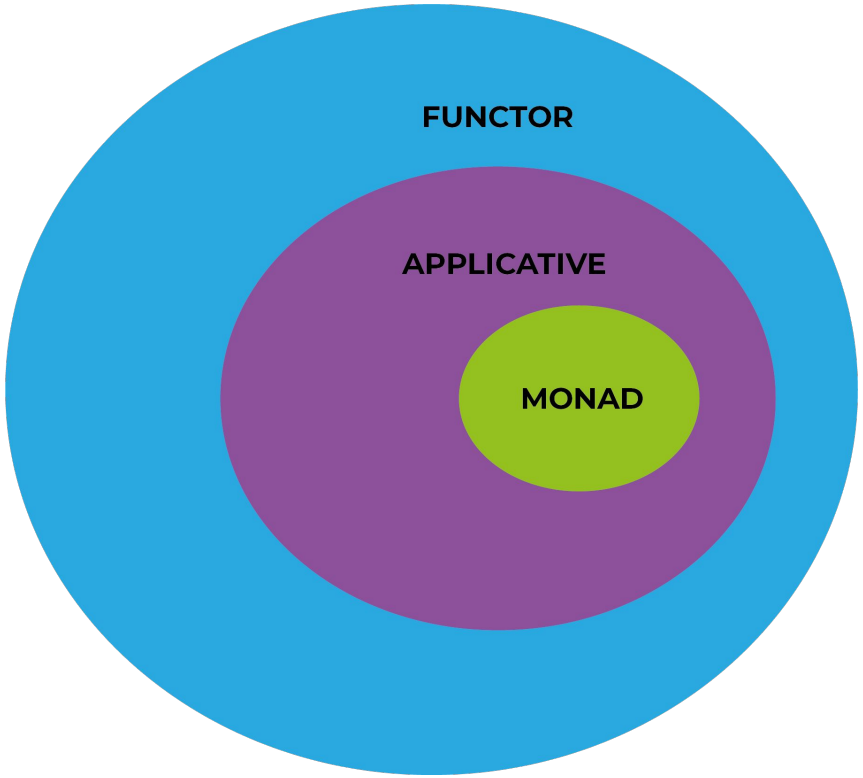
Functor je zobrazení mezi dvěma kategoriemi. Definuje nejen zobrazení mezi objekty, ale i mezi funkcemi.

Functor je zobrazení, které zachovává strukturu. Jedním příkladem mohou být seznamy. Můžete převést *List<Integer>* na *List<String>*, ale tento převod zachovává strukturu. To znamená, že výsledný objekt je také seznam a pořadí hodnot v seznamu se nemění.

Functor *F* definuje zobrazení mezi kategoriemi C a D:

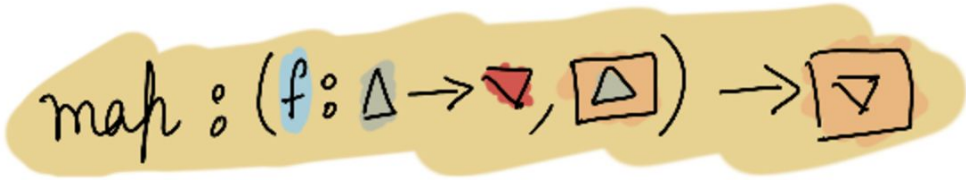


06 Functor, Applicative, Monad



06 Functor

```
interface Functor<out A> {
 fun <B> map(fn: (A) -> B) : Functor<B>
}
```



Identity Law: Pokud do map operace vložím identity funkci, tak musí zůstat původní functor (ten do kterého přes map injektuji identity funkci) nezměněn

```
F.map(x -> x).equals(F)
```

```
Optional<String> original = Optional.of("Hello");
Optional<String> result = original.map(x -> x);
assert result.equals(original);
```

Composition Law: složení dvou funkcí a následné mapování výsledku přes functor musí být stejné jako mapování každé funkce jednotlivě a následné složení výsledků

$$f.map(g \circ f) = f.map(f).map(g)$$

```
f.map(f.andThen(g)) == f.map(f).map(g)
```

```
Optional<String> original = Optional.of("Hello");

Function<String, Integer> g = String::length;
Function<Integer, Double> h = x -> x * 1.5;

Optional<Double> result1 = original.map(g.andThen(h));
Optional<Double> result2 = original.map(g).map(h);

assert result1.equals(result2);
```

06 Maybe functor

Maybe (nebo v některých jazycích nazývaný Optional, Option, či Nullable) je typ funktoru, který pomáhá bezpečně zpracovávat situace, kdy hodnota může být null, nebo chybí, bez nutnosti přímého ošetřování výjimek nebo použití podmínkových konstrukcí.

```java
public final class Maybe<T> {
    private final boolean hasItem;
    private final T item;

    // Empty constructor (no item)
    public Maybe() {
        this.hasItem = false;
        this.item = null;
    }

    // Constructor with an item
    public Maybe(T item) {
        if (item == null) {
            throw new NullPointerException("Item cannot be null");
        }
        this.hasItem = true;
        this.item = item;
    }

    // Getter for hasItem
    public boolean hasItem() {
        return hasItem;
    }

    // Getter for item
    public T getItem() {
        return item;
    }
```

06 Maybe functor

```java
// Map/Select operation
public <R> Maybe<R> map(Function<T, R> mapper) {
    if (mapper == null) {
        throw new NullPointerException("Selector cannot be null");
    }
    if (this.hasItem) {
        return new Maybe<>(mapper.apply(this.item));
    } else {
        return new Maybe<>();
    }
}

// Get the value or fallback
public T getValueOrFallback(T fallbackValue) {
    if (fallbackValue == null) {
        throw new NullPointerException("Fallback value cannot be null");
    }
    return this.hasItem ? this.item : fallbackValue;
}

// Override equals method
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof Maybe)) {
        return false;
    }
    Maybe<?> other = (Maybe<?>) obj;
    return Objects.equals(this.item, other.item);
}

// Override hashCode method
@Override
public int hashCode() {
    return hasItem ? Objects.hash(item) : 0;
}
```

06 Maybe functor

```
public static void main(String[] args) {
        Maybe<String> maybeWithValue = new Maybe<>("Hello");
        Maybe<String> emptyMaybe = new Maybe<>();

        // Map the value
        Maybe<Integer> lengthMaybe = maybeWithValue.map(String::length);

        System.out.println(lengthMaybe.hasItem()); // true
        System.out.println(lengthMaybe.getItem()); // 5

        // Fallback
        String fallback = emptyMaybe.getValueOrFallback("Fallback");
        System.out.println(fallback); // Fallback
    }
```

06 Lazy functor

Lazy funktor je koncept používaný pro odložení výpočtu do chvíle, kdy je výsledek skutečně potřeba. To znamená, že hodnota nebo transformace nejsou ihned vyhodnoceny, ale jsou zabaleny do konstrukce, která čeká, dokud nejsou explicitně požadovány.

Lazy funktory jsou užitečné v situacích, kdy:

1. Chcete optimalizovat výkon tím, že se vypočítá pouze to, co je skutečně potřeba.
2. Pracujete s nekonečnými datovými strukturami, které nelze plně vyhodnotit.
3. Řešíte řetězení složitých výpočtů, které nemusí být všechny nutné.

06 Lazy functor

```java
public class Lazy<T> {
    private final Supplier<T> supplier;
    private T value;
    private boolean isEvaluated = false;

    public Lazy(Supplier<T> supplier) {
        this.supplier = supplier;
    }

    public T getValue() {
        if (!isEvaluated) {
            value = supplier.get();
            isEvaluated = true;
        }
        return value;
    }

    // Instance method for mapping/transformation
    public <R> Lazy<R> map(Function<T, R> mapper) {
        return new Lazy<>(() -> mapper.apply(this.getValue()));
    }
```

06 Lazy functor

```
public static void main(String[] args) {
        // Create a Lazy instance with an expensive computation
        Lazy<Integer> lazyValue = new Lazy<>(() -> {
            System.out.println("Computing value...");
            return 42;
        });

        // Use the map method to transform the lazy value
        Lazy<String> lazyString = lazyValue.map(value -> "Value is: " + value);

        // Value hasn't been computed yet
        System.out.println("Lazy created, no computation yet.");

        // Access the value (triggers computation)
        System.out.println(lazyString.getValue());

        // Access the value again (uses cached result)
        System.out.println(lazyString.getValue());
    }
```

06 Stateful transformation functor

Stateful Transformation Functor je koncept, který umožňuje provádět transformace na hodnotách, přičemž zároveň uchovává a mění stav v průběhu transformace. Je užitečný, když transformace potřebují mít přístup ke sdílenému stavu, jako je počítadlo, akumulátor nebo jiné pomocné informace, které se mění během operací.

1. **Zachování a aktualizace stavu během transformací**
   Používá se k tomu, aby se vedle hlavní hodnoty uchovával i vedlejší stav, který se mění během zpracování dat. Například při transformaci seznamu je potřeba uchovávat součet zpracovaných hodnot.
2. **Řešení složitých stavových výpočtů funkcionálním způsobem**
   V klasických procedurálních jazycích bychom použili proměnnou pro sledování stavu. Stateful funktor umožňuje elegantnější a funkcionální způsob práce s těmito výpočty.
3. **Práce s iterativními nebo sekvenčními procesy**
   Například při zpracování seznamu nebo toku dat, kde je nutné uchovávat průběžný stav (např. index, součet nebo jiný akumulátor).
4. **Modelování stavových systémů**
   Používá se v případech, kdy se stav musí měnit postupně v reakci na akce nebo transformace. Příkladem je stavový automat nebo práce s akumulátory.

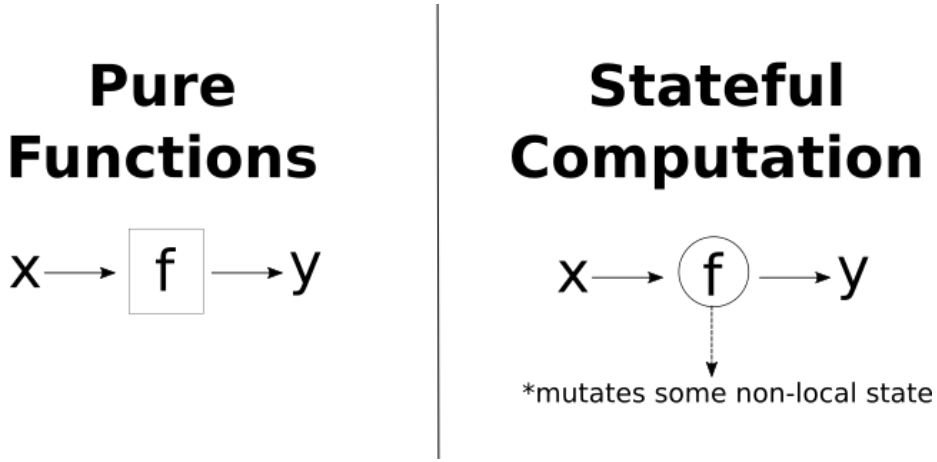
06 Stateful transformation functor

Stateful Transformation Functor kombinuje dvě části:

- Hodnotu (**T**): Hlavní datová struktura nebo hodnota, která se transformuje.
- Stav (**S**): Vedlejší stav, který se uchovává a aktualizuje během transformací.

Typicky je reprezentován jako dvojice (value, state). Operace map nebo flatMap pak umožňují transformaci hodnoty a zároveň modifikaci stavu.

Jeho výhoda je, že výsledek transformace a stav nemodifikují sdílené proměnné



06 Stateful transformation functor

```java
interface IState<S, T> {
    Pair<T, S> run(S state);

    // Functor map to transform the result
    default <U> IState<S, U> map(Function<T, U> mapper) {
        return inputState -> {
            Pair<T, S> result = this.run(inputState);
            return Pair.of(mapper.apply(result.getValue()), result.getState());
        };
    }
}
```

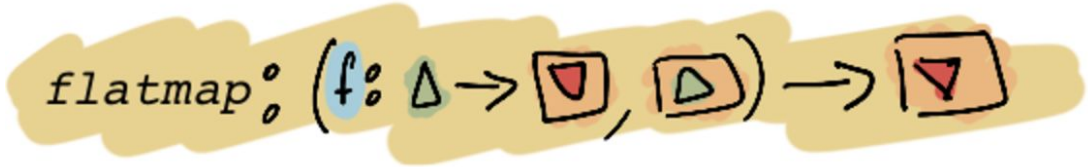
06 Stateful transformation functor

```java
public class Pair<L, R> {
    private final L value;
    private final R state;
    private Pair(L value, R state) {
        this.value = value;
        this.state = state;
    }

    public static <L, R> Pair<L, R> of(L value, R state) {
        return new Pair<>(value, state);
    }

    public L getValue() {
        return value;
    }
    public R getState() {
        return state;
    }
}
```

06 Stateful transformation functor

```java
public class StatefulParser {
    // Parse the next character and update the state
    public static IState<String, Character> parseNextChar() {
        return input -> {
            if (input.isEmpty()) {
                throw new IllegalStateException("End of input");
            }
            return Pair.of(input.charAt(0), input.substring(1));
        };
    }

    public static void main(String[] args) {
        // Step 1: Define a parser to parse the next character
        IState<String, Character> charParser = parseNextChar();

        // Step 2: Transform the parsed character to its ASCII code using map
        IState<String, Integer> asciiParser = charParser.map(ch -> (int) ch);

        // Step 3: Run the parser
        Pair<Integer, String> result = asciiParser.run("Hello");

        // Output the results
        System.out.println("Parsed ASCII code: " + result.getLeft()); // ASCII code of 'H' -> 72
        System.out.println("Remaining input: " + result.getRight()); // Remaining input -> "ello"

        // Example with another input
        Pair<Integer, String> result2 = asciiParser.run("World");
        System.out.println("Parsed ASCII code: " + result2.getLeft()); // ASCII code of 'W' -> 87
        System.out.println("Remaining input: " + result2.getRight()); // Remaining input -> "orld"
    }
}
```

06 MONÁDA - DEFINICE



Monáda je struktura, které obsahuje:

- **of** (unit): Tato operace převezme hodnotu a přenese ji do monadického kontextu - do monády zabalí obyčejnou hodnotu.
- **flatMap** (bind): Tato operace umožňuje seřadit výpočty, které produkují monadické hodnoty (monády). Přebírá monadickou hodnotu a funkci, která vytváří jinou monadickou hodnotu. Aplikuje funkci na rozbalenou hodnotu uvnitř monády a vrátí novou monádu.

Proti Functoru splňuje následující pravidla:

Left Identity Law : jestliže použijeme operaci of (zawrapujeme hodnotu do monády) a pak na ní aplikujeme ***flatMap***, tak dostaneme stejný výsledek jako aplikací funkce přímo na tuto hodnotu.

```
Optional.of(x).flatMap(f) == f.apply(x)
```

Right Identity Law: Jestliže aplikujeme flatMap na operaci ***of*** tak dostaneme původní monádu

```
m.flatMap(Optional::of) == m
```

```
var f = (Integer x) -> Optional.of(x + 1);

// Left Identity
Optional<Integer> left = Optional.of(3).flatMap(f);
Optional<Integer> right = f.apply(3);

System.out.println(left.equals(right)); // true
```

```
Optional<Integer> m = Optional.of(4);

// Right Identity
Optional<Integer> left = m.flatMap(Optional::of);
Optional<Integer> right = m;

System.out.println(left.equals(right)); // true
```

06 MONÁDA - DEFINICE

Associativity Law: Jestliže zřetězíme více *flatMap* operaci, tak nezáleží na způsobu jak je zgrupuji, když dodržím pořadí v jakém je řetězím

$$(2 + 3) + 4 = 2 + (3 + 4) = 9$$
$$2 \times (3 \times 4) = (2 \times 3) \times 4 = 24$$

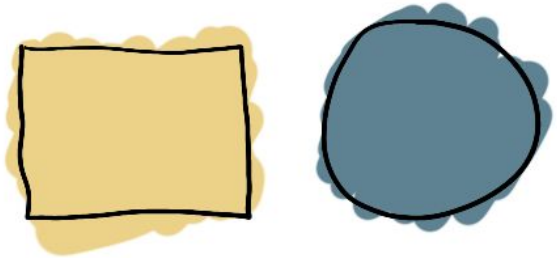
```
m.flatMap(f).flatMap(g) == m.flatMap(x -> f.apply(x).flatMap(g))
```

```java
// Monadic value
Optional<Integer> m = Optional.of(3);

// Functions f and g
var f = (Integer x) -> Optional.of(x + 1); // Adds 1
var g = (Integer y) -> Optional.of(y * 2); // Multiplies by 2

// Left-hand side: (m >>= f) >>= g
Optional<Integer> lhs = m.flatMap(f).flatMap(g);

// Right-hand side: m >>= (x -> f(x) >>= g)
Optional<Integer> rhs = m.flatMap(x -> f.apply(x).flatMap(g));

// Verify the law
System.out.println(lhs.equals(rhs)); // true
```

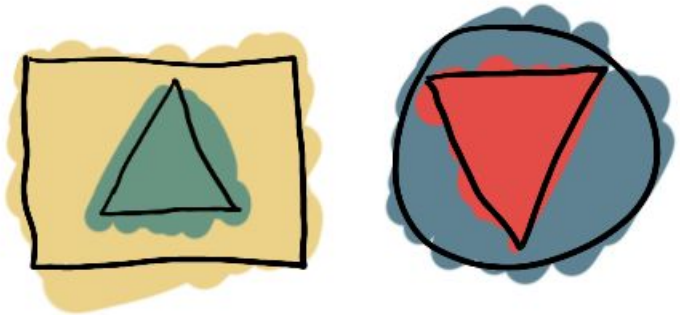
06 Cvičení na map a fmap

Základní typy:

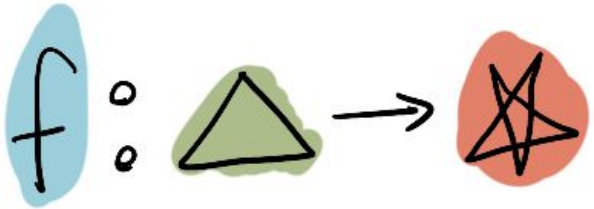
Kontainery:



Kontainery pro typy:

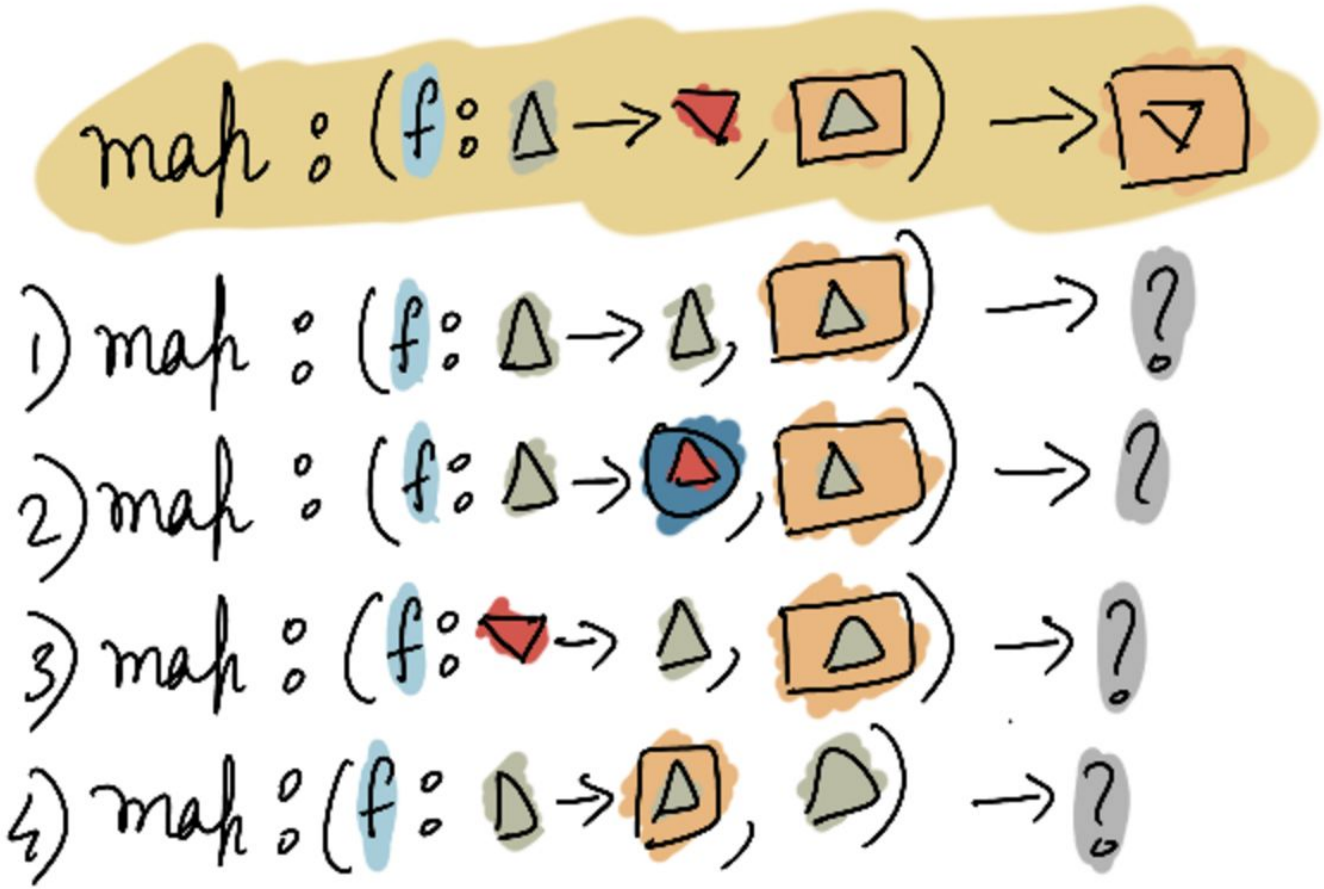


Funkce:



reference: https://last-ent.com/posts/intuitive-map-flatmap

06 Cvičení na map

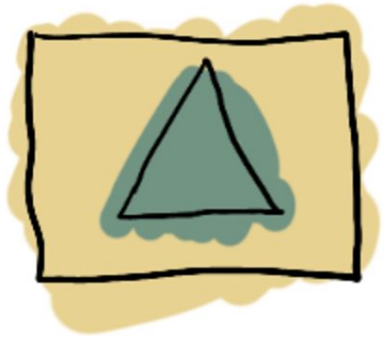


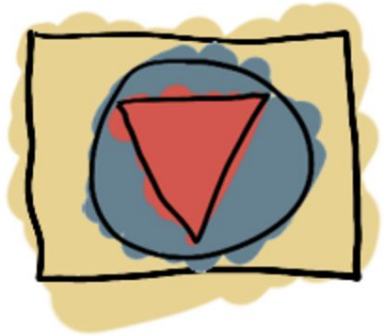
Rules for `map`

- Type contained by Container (C1) has to match Function's (F) input type.

reference: https://last-ent.com/posts/intuitive-map-flatmap

06 Cvičení na map

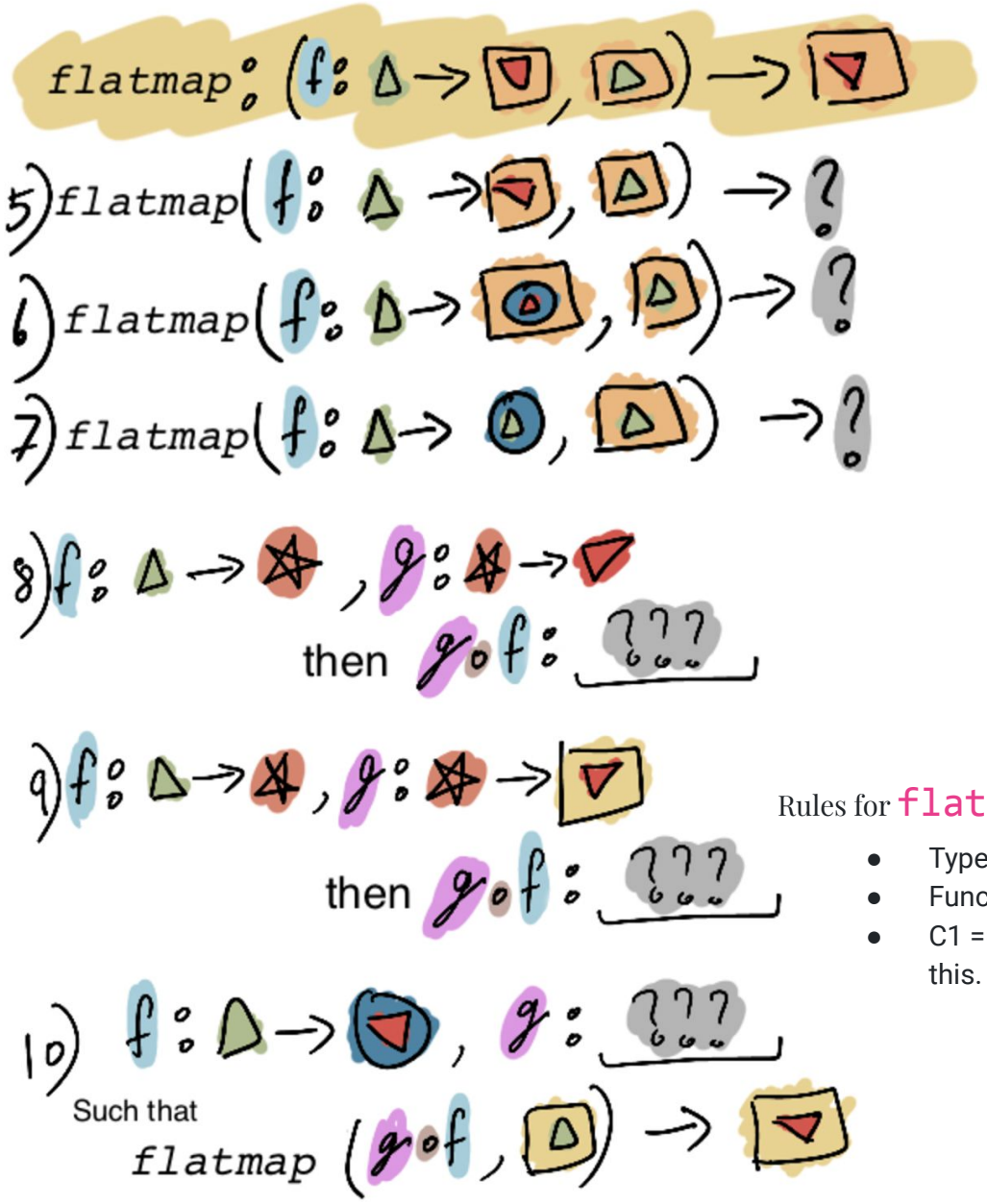
1. 

2. 

3. **Error!** *Breaks the rule for* ***map***
4. **Error!** ***map*** *works on a Container, not a Basic Type*

reference: https://last-ent.com/posts/intuitive-map-flatmap

06 Cvičení na flatMap



Rules for `flatmap`

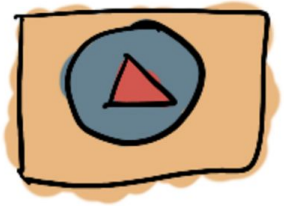
- Type contained by Container (C1) has to match Function's (F) input type.
- Function (F) has to return a Container (C2)
- C1 == C2. This is very important to remember as many people trip up on this.

reference: https://last-ent.com/posts/intuitive-map-flatmap

06 Cvičení na flatMap

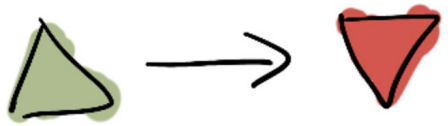
5.

6.

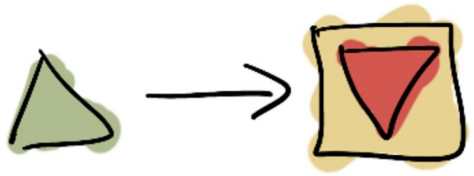


7. **Error!** *Breaks the 2nd Rule for* ***flatmap***

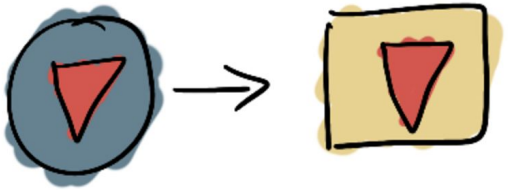
8.



9.



10.



reference: https://last-ent.com/posts/intuitive-map-flatmap

06 List monad

```java
import java.util.*;
import java.util.function.Function;
import java.util.function.Predicate;

public class ListMonad<T> {
    private List<T> value;

    // Constructor
    public ListMonad(List<T> value) {
        this.value = value;
    }

    // Utility function to retrieve the value of the ListMonad
    public List<T> getValue() {
        return value;
    }
    // For easy printing
    @Override
    public String toString() {
        return value.toString();
    }

    // The of function: Wraps a value into a ListMonad (equivalent to unit)
    public static <T> ListMonad<T> of(List<T> value) {
        return new ListMonad<>(value);
    }
```

06 List monad

```
// The map function: Applies a function to each element of the ListMonad
    public <R> ListMonad<R> map(Function<T, R> function) {
        List<R> mapped = new ArrayList<>();
        for (T item : value) {
            mapped.add(function.apply(item));
        }
        return new ListMonad<>(mapped);
    }

    // The flatMap function (bind): Applies a function that returns a monad
    public <R> ListMonad<R> flatMap(Function<T, ListMonad<R>> function) {
        List<R> result = new ArrayList<>();
        for (T item : value) {
            ListMonad<R> monad = function.apply(item);
            result.addAll(monad.value);  // Flattening the result
        }
        return new ListMonad<>(result);
    }

    public ListMonad<T> filter(Predicate<T> predicate) {
        List<T> filtered = new ArrayList<>();
        for (T item : value) {
            if (predicate.test(item)) {
                filtered.add(item);
            }
        }
        return new ListMonad<>(filtered);
    }
```

06 List monad

```
of Result: [1]
Mapped Result: [[2, 3, 4]]
Chained Result (flatMap): [2]
Complex flatMap Result: [1, 2, 3, 2, 4, 6, 3, 6, 9]
Combined Result: [20, 20, 40, 60, 60]
```

```java
public static void main(String[] args) {
        // 1. Wrapping values inside ListMonad using of
        ListMonad<Integer> monad = ListMonad.of(List.of(1));
        System.out.println("of Result: " + monad);

        // 2. Using map to transform the value inside the monad
        ListMonad<List> mapped = monad.map(x -> List.of(x + 1, x + 2, x + 3));
        System.out.println("Mapped Result: " + mapped);

        // 3. Using flatMap to chain operations that return monads
        ListMonad<Integer> chained = monad.flatMap(x -> ListMonad.of(List.of(x * 2)));
        System.out.println("Chained Result (flatMap): " + chained);

        ListMonad<Integer> complex = monad.flatMap(x -> ListMonad.of(List.of(x * 1, x * 2, x* 3 )))
                .flatMap(x -> ListMonad.of(List.of(x * 1, x * 2, x * 3 )));
        System.out.println("Complex flatMap Result: " + complex);

        // 4. Combinations
        ListMonad<Integer> combined = complex.filter(x -> x % 2 == 0)
                                             .map(x -> x * 10);

         System.out.println("Combined Result: " + combined);
    }
```

06 Monad versus Functor

```java
public static void functorExample() {
    // Starting with a Just(5)
    Maybe<Integer> computation = MaybeUtils.just(5)
        .map(value -> value + 10) // Add 10 (result: Just(15))
        .map(value -> value * 2); // Multiply by 2 (result: Just(30))

    System.out.println("Result with Functor: " + computation);
}
```

VS

```java
public static void monadExample() {
    // Starting with a Just(5)
    Maybe<Integer> computation = MaybeUtils.just(5)
        .flatMap(value -> MaybeUtils.just(value + 10)) // Add 10 (result: Just(15)
        .flatMap(value -> MaybeUtils.just(value * 2))  // Multiply by 2 (result: Ju
        .flatMap(value -> value > 30                   // Fail if result > 30
            ? MaybeUtils.nothing()
            : MaybeUtils.just(value));

    System.out.println("Result with Monad: " + computation);
}
```

06 Monad versus Functor

Functor je jednodušší, ale hodí se spíše jako utility funkce. Skládání více operací do sebe je nutné dělat staticky, nelze rozumným způsobem skládat pipeliny dynamicky, chovat se podmíněně výsledku

Functor je komplexnější a narozdíl od functoru umožňuje dynamické skládání výpočetních pipelines

06 JAK NA NPE - HRUBÁ SÍLA

```java
import lombok.AllArgsConstructor;
import lombok.Getter;

@Getter @AllArgsConstructor
public class Certification {
    private String type;
    private double score;
}

@Getter @AllArgsConstructor
public class Framework {
    private String name;
    private Certification certification;
}

@Getter @AllArgsConstructor
public class Job {
    private int yearsOfExpr;
    private Framework framework;
}

@Getter @AllArgsConstructor
class Candidate{
    private String name;
    private Job performedJob;
}
```

Snažíme se explicitně ošetřit všechna místa, kde mohl nastat null -kontrolou na !=null

```java
public class Client {
    static String assessUIProficiency(Candidate candidate) {
        String proficiency = null;
        Job job = candidate.getPerformedJob();
        if (job != null) {
            if (job.getFramework() != null) {
                Framework framework = job.getFramework();
                if (framework != null) {
                    Certification cert = framework.getCertification();
                    if (cert != null && cert.getScore() >= 60) {
                        proficiency = "Expert";
                    } else
                        proficiency = "Noob";
                }
            }
        }
        return proficiency;
    }

    public static void main(String[] args) {
        Candidate candidate = new Candidate("Karel"
                , new Job(11
                , new Framework("Spring"
                , new Certification("Gold", 99))));

        System.out.println(Client.assessUIProficiency(candidate));
    }
}
```

@Getter a @AllArgsConstructor – see Lombok buddy

06 3 JAK NA NPE - OPTIONAL - STREAMY

Následující kód nastavuje „Unable to assess” v případě jakékoliv null hodnoty v objektovém grafu

```
static String assessUIProficiency(Candidate candidate) {
    String proficiency = null;

    proficiency = Optional.ofNullable(candidate)
                .map(can -> can.getPerformedJob())
                .map(job -> job.getFramework())
                .map(framework -> framework.getCertification())
                .map(certification ->  {
                    if (certification.getScore() >= 60)
                        return "expert";
                    else
                        return "noob";
                })
                 .orElse("Unable to asses");
     return proficiency;
}
```

06 3 JAK NA NPE - OPTIONAL

Případně můžeme jít ještě dále a v případě null hodnoty na konkrétní úrovni objektového grafu zareagovat specificky – např. nastavit defaultním objektem, který umožní další zpracování

```
static String assessUIProficiencySmart(Candidate candidate) {
    String proficiency = null;

    proficiency = Optional.ofNullable(candidate)
            .map(can -> can.getPerformedJob())
            .map(job -> job.getFramework())
            .map(framework -> Optional.ofNullable(framework.getCertification())
                    .orElse(new Certification("Dummy Cert", 20.0)))
            .map(certification ->  {
                if (certification.getScore() >= 60)
                    return "expert";
                else if (certification.getScore() >= 10)
                    return "average expertize";
                else
                    return "noob";
            })
            .orElse("Unable to asses");
    return proficiency;
}

public static void main(String[] args) {
    Candidate candidate2 = new Candidate("Karel"
            , new Job(11
            , new Framework("Spring"
            , null)));

    System.out.println(Client.assessUIProficiencySmart(candidate2));
}
```

06 MONÁDY - Futures/Promise pattern v Java

Runnable a Supplier rozhraní umožňují se dále zbavit kódu pro multi-threading pomocí metod runAsync() nebo supplyAsync() s pomocí lambda expressions následovně:

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello");
assertEquals("Hello", future.get());
```

Pokud chceme výsledek výpočtu dále zprocesovat funkcí, tak použijeme metodu *thenApply*, která vezme výsledek výpočtu, ten zprocesuje funkcí o vrátí CompletableFuture, který drží nový výsledek.

```
CompletableFuture<String> compFuture = CompletableFuture.supplyAsync(() -> "Hello");
CompletableFuture<String> future = compFuture.thenApply(s -> s + " World");
assertEquals("Hello World", future.get());
```

Jestliže nepotřebujeme vracet future bez návratové hodnoty - CompletableFuture<Void> future, tak se použije thenAccept()

```
CompletableFuture<Float> compFuture = CompletableFuture.supplyAsync(() -> 120.0);
CompletableFuture<Void> future = compFuture.thenAccept(s -> ShoppingList.setPrice(s));
future.get();
```

06 MONÁDY - Futures/Promise pattern v Java

Zpracováváme více futures najednou, skládáme/vyhodnocujeme je sekvenčně, paralelně nebo dokonce hybridně.

Sekvenční zřetězení více futures

1) *thenCompose()* - výsledek zpracování funkce dáváme jako vstup do následující:

```
CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> "Hello")
    .thenCompose(s -> CompletableFuture.supplyAsync(() -> s + " World"));
 assertEquals("Hello World", completableFuture.get());
```

Pozn. Java Stream map() je realizována kombinací thenCompose a thenApply
Rozdíl mezi thenCompose() a thenApply() je jako mezi Java Stream map() a flatMap(). Prosím prostudujte.

2) *thenCombine()* - výsledek dvou funkcí zkombinujeme do následující:

```
CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> "Hello")
      .thenCombine(CompletableFuture.supplyAsync(() -> " World"), (s1, s2) -> s1 + s2));
assertEquals("Hello World", completableFuture.get());
```

06 MONÁDY - Futures/Promise pattern v Java

Paralelní zřetězení více futures

1) ***allOf()*** - blokuje se dokud nejsou vyhodnoceny všechny výsledky:

```
CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> "Hello");
CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> "Beautiful");
CompletableFuture<String> future3 = CompletableFuture.supplyAsync(() -> "World");
CompletableFuture<Void> combinedFuture = CompletableFuture.allOf(future1, future2, future3);
combinedFuture.get();
assertTrue(future1.isDone());
assertTrue(future2.isDone());
assertTrue(future3.isDone());
```

2) anyOf() - blokuje se dokud není vyhodnocen jeden z výsledků:

```
CompletableFuture<String> future0 = createCFLongTime(0);
CompletableFuture<String> future1 = createCF(1);
CompletableFuture<String> future2 = createCFLongTime(2);
CompletableFuture<Object> future = CompletableFuture.anyOf(future0, future1, future2);
System.out.println("Future result>> " + future.get());
System.out.println("All combined>> " + future0.get() + "|" + future1.get() + "|" + future2.get());}
```

06 MONÁDY - Futures/Promise pattern v Java

V porovnání s try/catch blokem (syntaktický blok), v *CompletableFuture* se použije speciální metoda *handle()*. Jejími parametry jsou výsledek zpracování jestliže vše doběhlo OK a výjimka pokud nastal problém.

```
String name = null;
...
CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() ->
{       if (name == null) {
                throw new RuntimeException("Computation error!");
        }
return "Hello, " + name;
    })}).handle((s, t) -> s != null ? s : "Hello, Stranger!");

assertEquals("Hello, Stranger!", completableFuture.get());
```

06 MONÁDY - Futures/Promise pattern v Java

Další příklad na **allOf()**

```
StringBuilder result = new StringBuilder();
List<String> messages = Arrays.asList("a", "b", "c");

List<CompletableFuture<String>> futures = messages.stream()
        .map(msg -> CompletableFuture.completedFuture(msg).thenApply(s -> delayedUpperCase(s)))
        .collect(Collectors.toList());
CompletableFuture.allOf(futures.toArray(new CompletableFuture[futures.size()])).whenComplete((v, th) -> {
        futures.forEach(cf -> assertTrue(isUpperCase(cf.getNow(null))));
        result.append("done");
});
```