czm

Search . . .

09

Funkcionální přístup 2

- Supplier a Consumer
- Akumulátor
- Monáda a functor
- Future/Promisy
- Řešení chyb a NPE

Next >

Objektové Modelování - podzim 2022

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

Funkce první třídy a funkce vyššího řádu

Java neumožňuje pracovat s funkcí jako s typem. Místo toho použilo tzv. *Funkcionální rozhraní*:

- Function

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

OPAKOVÁNÍ

- Predicate

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

- Function T->R
- UnaryOperator T->T
- BiFunction T->T
- BinaryOperator T,T->T
- Supplier ()->T
- Consumer T->()

Funkcionální rozhraní je ***SAM*** = *Single Abstract Method Interface* – rozhraní s jedinou abstraktní metodou

Můžete si definovat vlastní rozhraní, pokud budou SAM

09 SUPPLIER

Supplier se používá pro reprezentaci komponent pro poskytování objektů nějakého datového typu.
Příklad pěstitel obilí, výdejna obědů….

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

supplier

```
public class UnboundedStreamingApp {
    private final Logger logger = Logger.getLogger(UnboundedStreamingApp.class.getName());

    public void generateStreamingData(){
        Stream.generate(()->UUID.randomUUID().toString())
        .limit(10)
        .forEach(logger::info);
    }
}
```

09 SUPPLIER – Jako Factory Pattern

```java
interface AbstractProduct {}
class ConcreteProductA implements AbstractProduct {}
class ConcreteProductB implements AbstractProduct {}

class Factory {
    private Supplier<AbstractProduct> supplier;
    public Factory(Supplier<AbstractProduct> supplier) {
        super();
        this.supplier = supplier;
    }
    public void setSupplier(Supplier<AbstractProduct> supplier) {
        this.supplier = supplier;
    }
    public AbstractProduct makeProduct() {
        return supplier.get();
    }
}

public class FactoryDemo {
    public static void main(String[] args) {
        Factory factory = new Factory(()-> new ConcreteProductA());
        AbstractProduct product = factory.makeProduct();
        System.out.println(product.toString()); // prints a productA
        factory.setSupplier(()->new ConcreteProductB());
        product = factory.makeProduct();
        System.out.println(product.toString()); // now prints a productB
    }
}
```

09 Skládání operací do sebe - *andThen()*

Metoda ***andThen*** bere jako argument jinou funkci (after) a vrací novou funkci, která představuje složení aktuální funkce následované funkcí ***after***.

T typ vstupu do složené funkce, ***R*** je typ výsledku aktuální funkce a ***V*** je typ výsledku složené funkce.

```java
default <V> Function<T, V> andThen(Function<? super R, ? extends V> after)
```

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

public class FunctionAndThenExample {
    public static void main(String[] args) {
        // Create two functions
        Function<Integer, Integer> add = x -> x + 2;
        Function<Integer, Integer> multiply = x -> x * 3;

        // Using andThen to compose the functions
        Function<Integer, Integer> addAndMultiply = add.andThen(multiply);

        // Applying the composed function
        int result = addAndMultiply.apply(5);

        System.out.println(result); // Output: (5 + 2) * 3 = 21
    }
}
```

09 CONSUMER

Consumer pattern se používá pro reprezentaci komponenty, která konzumuje (a zpracovává vstupy) objekty nějakého typu

Příklad zpracovatel obilí na mouku, studenti konzumující obědy

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> {
          accept(t); after.accept(t);
        };
    }
}
```

09 CONSUMER

Vytvořili jsme implementaci, která konzumuje řetězec a poté jej vytiskne. Metoda forEach přijímá implementaci Consumer interface.

```
public void whenNamesPresentConsumeAll(){
    Consumer<String> printConsumer = t -> System.out.println(t);
    Stream<String> cities = Stream.of("Sydney", "Dhaka", "New York", "London");
    cities.forEach(printConsumer);
}
```

V tomto případě skládáme více *Consumer* implementací za sebe. V tomto konkrétním příkladu jsme vytvořili dva; jeden převede seznam položek na stringy s velkými písmeny a druhý vytiskne string s velkými písmeny.

```
public void whenNamesPresentUseBothConsumer(){
    List<String> cities = Arrays.asList("Sydney", "Dhaka", "New York", "London");
    Consumer<List<String>> upperCaseConsumer = list -> {
        for(int i=0; i< list.size(); i++){
            list.set(i, list.get(i).toUpperCase());
        }
    };
    Consumer<List<String>> printConsumer = list -> list.stream().forEach(System.out::println);
    upperCaseConsumer.andThen(printConsumer).accept(cities);
}
```

09 AKUMULÁTOR

Kotlin

Mám while cyklus, kterým postupně čtu data a z nich něco sestavuji

```
fun readStream(stream: FileReader): Int {
    var totByteRead = 0
    while(true){
        val bytesRead = stream.read()
        totByteRead = totByteRead + bytesRead
        if ( bytesRead == -1)
            break
    }
    return totByteRead
}
```

Funkcionální programování se snaží vyhnout tomuto způsobu, protože:

- pracuje s mnoho lokálními proměnnými
- do kódu se špatně zasahuje zvenku a když, tak s nekontrolovaným dopadem
- kód se špatně dekomponuje do menších částí
- nedá se dobře přepoužívat nebo komponovat do složitějších pipelines

09 AKUMULÁTOR

Kotlin

*Funkcionální programování podobný kód transformuje pomocí rekurze a akumulátoru. Máme funkci, která akceptuje **accumulator** (počet přečtených bytů) a operaci (volání čtení ze streamu) a vrací nový akumulátor. Akumulátor tedy akumuluje částečné výsledky jak postupně provolává operace.*

```
tailrec fun reduceOperations(
        accumulator: Int,
        operation: (Int) -> Int
): Int {
    val value = operation(accumulator)
    return if (value < accumulator) {
        accumulator
    } else {
        reduceOperations(value, operation)
    }
}

fun main() {
    val path = System.getProperty("user.dir")
    println("Working Directory = $path")

    val stream = FileInputStream("input.txt")
    val bytesRead2 = reduceOperations(0) { it + stream.read() }
}
```

Pozn: **it** je v Kotlinu zkratka za implicitní parametr … (it)->it + stream.read()

09 COMPOSABILITY

Obvyklá praxe ve funkcionálním přístupu, kdy libovolně zkomponujeme operace do výpočetní pipeliny (nebo zkomponujeme více pipelines) a odložíme evaluaci až do bodu, kdy budeme potřebovat výsledek

MONÁDY

Monáda je funkcionální design pattern řešící situace jako:

- Nullability: Maybe/Option monáda
- Error Handling: Either monáda
- DI (Dependency Injection): Reader monáda
- Logování: Writer monáda
- Side Effects : IO monáda
- State handling: State monáda
- Collections: List monáda
- Zpožděné zpracování: Future (promise) monáda
- a další…

Základní idea je, že funce a jejich kompozice převedeme z:

Do:

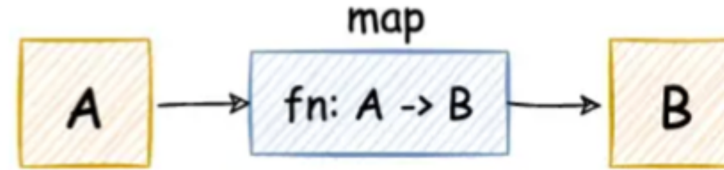
Tato zdánlivě triviální změna umožní spoustu zajímavých věcí

09 FUNCTOR - DEFINICE

Kotlin

Functor je kontainer, který nese hodnotu a umožňuje nám na tuto hodnotu aplikovat mapovací funkci *map*.

```
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 nezměněn

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

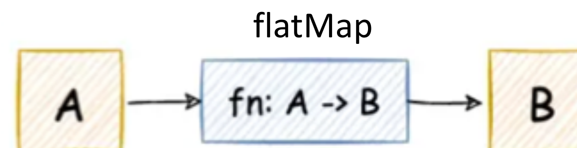
Composition Law: složení dvou funkcí a následné mapování výsledku přes functor musí být stejné jako mapování každé funkce jednotlivě a následné složení výsledků

```
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);
```

09 MONÁDA - DEFINICE

Monáda je struktura, které obsahuje:

- **unit** nebo **return** : Tato operace převezme hodnotu a přenese ji do monadického kontextu - do monády zabalí obyčejnou hodnotu.
- **bind** nebo **flatMap**: 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.



Navíc proti Functoru splňuje následující pravidla:

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

```
Optional<Integer> result1 = Optional.of(5).flatMap(x -
> Optional.of(x * 2));
Optional<Integer> result2 = Optional.of(5 * 2);

assert result1.equals(result2);
```

Right Identity Law: Jestliže aplikujeme flatMap na unit operaci tak musíme dostat původní monádu

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

assert result.equals(original);
```

09 MONÁDA - DEFINICE

Kotlin

Associativity Law: Jestliže zřetězíme více *flatMap* operaci, tak nezáleží na pořadí v jakém je řetězím

```
int absIncr(int x) {
  return abs(incr(x));
}
```

Mějme dvě funkce – *abs* a *incr*. Můžeme je zřetězit a nebo definovat novou funkci, která je kompozicí obou dvou. V obou případech bychom měli dostat ten stejný výsledek.

Pokud bychom v abs a incr chtěli sbírat i logovací hlášky a zároveň to udělat bez side efektů (print přímo při volání funkce), tak ji zabalíme do monády, která nese mezivýsledek operace, ale i logovací string.

```
public static <T> Loggable<T> of(T value) {
  return new Loggable<>(value, "");
}
...

Loggable<Integer> incrWithLog(int x) {
  return new Loggable<>(incr(x), "incr " + x + "; ");
}

Loggable.of(4)
      .flatMap(x -> incrWithLog(x))
      .flatMap(x -> absWithLog(x))
```

< = >

```
Loggable<Integer> absIncrWithLog(int x) {
  return incrWithLog(x).flatMap(y -> absWithLog(y));
}

Loggable.of(4)
      .flatMap(x -> absIncrWithLog(x))
```

obě operace na posledním řádku mají stejný výsledek

09 MONÁDA - DEFINICE

Kotlin

```
public class MonadAssociativityExample {
    public static void main(String[] args) {
        // Assume m is a Monad, and f, g, h are functions
        Optional<Integer> result1 = Optional.of(5)
            .flatMap(x -> multiplyByTwo(x))
            .flatMap(y -> addThree(y));

        Optional<Integer> result2 = Optional.of(5)
            .flatMap(x -> multiplyByTwo(x).flatMap(y -> addThree(y)));

        // Asserting that the results are equal
        assert result1.equals(result2);

        // Print the results
        System.out.println("Result 1: " + result1.orElse(-1)); // Output: 13
        System.out.println("Result 2: " + result2.orElse(-1)); // Output: 13
    }

    // Function to multiply a number by two and wrap in Optional
    static Optional<Integer> multiplyByTwo(int x) {
        return Optional.of(x * 2);
    }

    // Function to add three to a number and wrap in Optional
    static Optional<Integer> addThree(int x) {
        return Optional.of(x + 3);
    }
}
```

monad.flatMap(x -> f(x)).flatMap(x -> g(x)) dá stejný výsledek jako *monad.flatMap(x -> f(x).flatMap(y -> g(y)))*

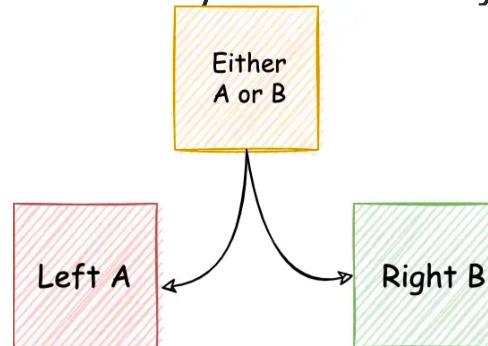
MONÁDY - Either

Zpracování chyb pomocí výjimek má spoustu benefitů, ale také negativ jako např.:

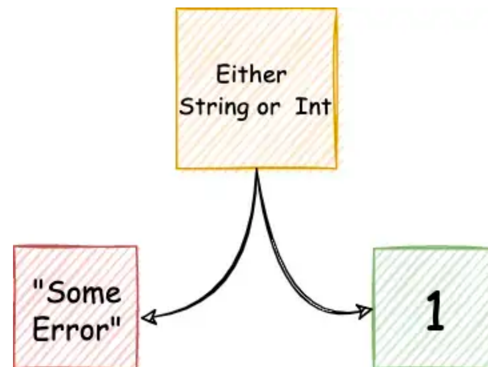
- *Nechceme jednoduše vybublat výjimku až nahoru a tam ji pouze vypsat, ale chceme se s ní vypořádat v aplikaci*
- *Z chyby je možné se zotavit*

Monády nabízejí lepší řešení:

- *Either* může nabývat dvou možných hodnot: *Left* nebo *Right*



- *Left* budeme používat pro *Failure* a *Right* pro *Success*



09 MONÁDY - Either

Kotlin

```
sealed class Either<out A, out B> {
    data class Left<A>(val value: A) : Either<A, Nothing>()
    data class Right<B>(val value: B) : Either<Nothing, B>()
}
```

Pozn: Data class je jako Record v Java

```
data class Account private constructor(val balance: BigDecimal) {

    companion object {
        fun create(initialBalance: BigDecimal): Either<NegativeAmount, Account> =
            if (initialBalance < 0) Either.Left(NegativeAmount)
            else Either.Right(Account(initialBalance))
    }
}
```

Pozn: Companion object – obdoba static v Java

Vytvoří účet s iniciálním zůstatkem a může buď failovat, protože zůstatek je negativní nebo dopadnout korektně

09 MONÁDY - Either

Kotlin

Teď budeme chtít na účet začít ukládat peníze

```
data class Account private constructor(val balance: BigDecimal) {

    companion object {
        fun create(initialBalance: BigDecimal): Either<NegativeAmount, Account> =
            if (initialBalance < 0) Either.Left(NegativeAmount)
            else Either.Right(Account(initialBalance))
    }

    fun deposit(amount: BigDecimal): Account = this.copy(balance = this.balance + amount)
}
```

Immutable kopie

a použijeme např. takto:

```
val account = Account.create(100.toBigDecimal())
when (account) {
    is Either.Right -> account.value.deposit(100.toBigDecimal())
    is Either.Left -> TODO() // now what?
}
```

09 MONÁDY - Either

Kotlin

a nebo obohatíme naši monádu o map funkci

```
sealed class Either<out A, out B> {
    class Left<A>(val value: A) : Either<A, Nothing>()
    class Right<B>(val value: B) : Either<Nothing, B>()

    fun <C> map(fn: (B) -> C): Either<A, C> = when (this) {
        is Right -> Right(fn(this.value))
        is Left -> this
    }
}
```

a kód přepsat do tvaru:

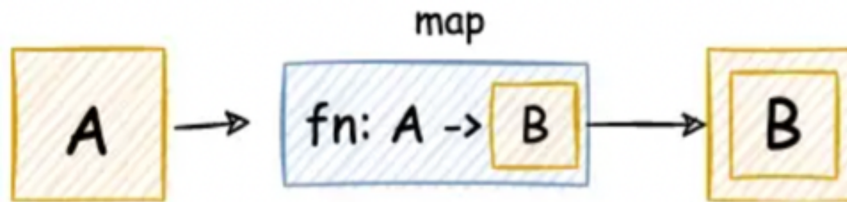
```
val account = Account.create(100.toBigDecimal())
    .map { a -> a.deposit(100.toBigDecimal()) }
```

map je sémanticky připojena k typu monády, v našem případě k monádě Either, použije fn pouze v případě, že máme *Right*, jinak bude funkce prostě ignorována

MONÁDY - flatMap

Kotlin

To nám bohužel ale zvětšilo úroveň zanoření




```
val account: Either<NegativeAmount, Either<NegativeAmount, Account>> = Account.create(100.
    .map { a -> a.deposit(100.toBigDecimal()) }
```

Co s tím??? Vzpomeňte na *flatMap* z java streams přednášky

```
sealed class Either<out A, out B> {
    class Left<A>(val value: A) : Either<A, Nothing>()
    class Right<B>(val value: B) : Either<Nothing, B>()

    fun <C> map(fn: (B) -> C): Either<A, C> = when (this) {
        is Right -> Right(fn(this.value))
        is Left -> this
    }
    fun <A, C> flatMap(fn: (B) -> Either<A, C>): Either<A, C> = when (this) {
        is Right -> fn(this.value)
        is Left -> this as Either<A, C>
    }
}
```

MONÁDY

Kotlin

Co kdybychom chtěli průběžně kontrolovat jestli se nám zůstatek účtu při withdrawal a deposit operacích nedostal do záporného zůstatku?

```
data class Account private constructor(val balance: BigDecimal) {
    companion object {
        fun create(initialBalance: BigDecimal): Either<NegativeAmount, Account> =
            applyAmount(initialBalance) { Account(it) }

        private fun applyAmount(amount: BigDecimal, fn: (BigDecimal) -> Account) =
            if (amount < ZERO) Either.Left(NegativeAmount)
            else Either.Right(fn(amount))
    }

    fun deposit(amount: BigDecimal): Either<NegativeAmount, Account> =
        applyAmount(amount) { this.copy(balance = this.balance + it) }
}
```

<== a zavoláme zde

<== přidáme apply funkci

09 MONÁDY – jako composition builder

Kotlin

Přidáme další metodu – *withdrawal* a začneme krásně komponovat operace s účtem

```
data class Account private constructor(val balance: BigDecimal) {
    // Other methods

    fun withdraw(amount: BigDecimal): Either<AccountError, Account> =
        applyAmount(amount) { this.copy(balance = this.balance - it) }
            .flatMap {
                if ((balance - amount) < ZERO) Left(NotEnoughFunds) else Right(Account(balance - amount))
            }
}
sealed class AccountError {
    object NegativeAmount : AccountError()
    object NotEnoughFunds : AccountError()
}
```

Note: object inside a class in Kotlin define a singleton object, which is a single instance lazily created when accessed.

09 MONÁDY – jako composition builder

Kotlin

Scenář 1: Jane chce otevřít účet se 100, pak vložit 100 a nakonec vybrat 250. Což má skončit chybou, protože na účtu není dostatečný zůstatek.

```
val account = Account.create(100.toBigDecimal())
    .flatMap { it.deposit(100.toBigDecimal()) }
    .flatMap { it.withdraw(250.toBigDecimal()) }
// account = Left(value=NotEnoughFunds)
```

Scenář 2: Operace na přesun peněz mezi dvěma různými účty

```
class TransferMoney {
  operator fun invoke(debtor: Account, creditor: Account, amount: BigDecimal): Either<AccountError, Pair<Account, Account>> =
    debtor
      .withdraw(amount)
      .flatMap { d -> creditor.deposit(amount).map { Pair(d, it) } }
}
```

09 MONÁDY – jako composition builder

Kotlin

Scenář 3: Kombinujeme celou řadu akcí jako vyhledat účet, vložit peníze a uložit nový stav účtu

```
interface AccountRepository {
    fun findBy(userId: UUID): Either<AccountNotFound, Account>
    fun save(account: Account)
}
sealed class AccountError {
    object NegativeAmount : AccountError()
    object NotEnoughFunds : AccountError()
    object AccountNotFound : AccountError()
}

class DepositCash(private val repository: AccountRepository) {
  operator fun invoke(userId: UUID, amount: BigDecimal): Either<AccountError, Unit> =
    repository.findBy(userId)
      .flatMap { it.deposit(amount) }
      .map(repository::save)
}
```

JAK NA NPE

Kdy vzniká NPE (Null Pointer Exception)?

- Volání metody na instanci objektu, který je NULL
- Přístup k atributu objektu, který je NULL nebo jeho úprava
- Zjištění délky pole, které je NULL
- Throw NULL, jako by to byl Throwable objekt

```
public void doSomething() {
    String result = doSomethingElse();
    if (result.equalsIgnoreCase("Success"))
        // success
    }
}

private String doSomethingElse() {
    return null;
}
```

```
public static void main(String[] args) {
    findMax(null);
}

private static void findMax(int[] arr) {
    int max = arr[0];
    //check other elements in loop
}
```

08 JAK NA NPE – HRUBÁ SÍLA

Nejčastější přístup je, že do kódu dáme na všechna místa kontroly na NULL

```
public void doSomething() {
    String result = doSomethingElse();
    if (result != null && result.equalsIgnoreCase("Success")) {
        // success
    }
    else
        // failure
}

private String doSomethingElse() {
    return null;
}
```

To ale vede na nepřehledný kód plný redundancí a stejně se velmi často zapomene na korektní ošetření všech situací

08 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

08 JAK NA NPE – KONTROLOVANÝ API KONTRAKT

API je postavené tak, že nikdy nemůže vrátit NULL a namísto toho vyhazuje Exception

```
public void print(Object param) {
    System.out.println("Printing " + param);
}

public Object process() throws Exception {
    Object result = doSomething();
    if (result == null) {
        throw new Exception("Processing fail. Got a null response");
    } else {
        return result;
    }
}
```

08 JAK NA NPE – KONTROLOVANÝ API KONTRAKT

Kontrolu do API nezavádíme ručně, ale pomocí anotací. Např. *@NonNull*

```java
import lombok.NonNull;

public class NonNullExample {
    private int yearsOfExperience;

    public NonNullExample(@NonNull Job job) {
        this.yearsOfExperience = job.getYearsOfExpr();
    }
}
```

Se během kompilace expanduje do:

```java
import lombok.NonNull;

public class NonNullExampleExpanded {
    private int yearsOfExperience;

    public NonNullExampleExpanded(@NonNull Job job) {
        if (job == null) {
            throw new NullPointerException("job is marked non-null but is null");
        }
        this.yearsOfExperience = job.getYearsOfExpr();
    }
}
```

08 JAK NA NPE – ASSERTY nebo OBJECTS API

```
public void accept(Object param){
    assert param != null;
    doSomething(param);
}
```

assert se používá velmi při unit testování, ale lze použít i pro jednoduché kontroly na stav objektu

```
public void accept(Object param) {
    Objects.requireNonNull(param);
    // doSomething()
}
```

Objects obsahuje různe statické kontroly, zde vyhazuje NPE, když je param NULL

08 JAK NA NPE – ASSERTY nebo OBJECTS API

Vždy vytvářet/vracet prázdné kolekce místo NULL

```
public class XYZ {
  List <String> names = null; //Bad
  List <String> seznam = new Arraylist<ArrayList>(); //Good


}

...

//Even better place into getters
public List<String> names() {
    if (userExists()) {
        return Stream.of(readName()).collect(Collectors.toList());
    } else {
        return Collections.emptyList();
    }
}
```

08 JAK NA NPE – OPTIONAL

Optional vytváří wrapper, který může být uvnitř prázdný nebo obsahovat hodnotu

Pokud API vrací *Optional*, tak tím explicitně říkáme, aby si programátor ošetřil kód po návratu z funkce. Tam kde API nevrací *Optional*, tak se můžeme spolehnout, že kontrolu na NULL nemusíme dělat.

```
public class HandlingNulls {
    private static String getUserById(int userId) {
        if (userId == 1) {
            return "Adam Roth";
        } else {
            return null;
        }
    }

    public static Optional<Object> process(int userId ) {
        String response = getUserById(userId);
        if (response == null) {
            return Optional.empty();
        }
        return Optional.of(response);
    }

    public static void main(String [] params){
        Optional userWrapper  = HandlingNulls.process(1);
        if (userWrapper.isPresent())
            System.out.println("User found: " + userWrapper.get());
        System.out.println("User not found");
    }
}
```

08 JAK NA NPE – OPTIONAL

```
Optional<Job> optJob = Optional.empty();
```

Vytvoří prázdnou monádu

```
Optional<Job> optJob = Optional.of(candidate.getJobProfile());
```

Vytvoří monádu, ale pokud je vkládaný objekt NULL, tak vyhodí NPE

```
Optional<Job> optJob = Optional.ofNullable(candidate.getJobProfile());
```

Vytvoří monádu, pokud je vkládaný objekt NULL, tak vrací práydnou monádu, jinak monádu s hodnotou

```
if (optJob.isPresent(){…
if (optJob.isEmpty(){…
```

Kontrolujeme obsah monády

```
x = optJob.orElse(”other“)
```

Vrátí hodnotu v monádě, jestliže isPresent() jinak vrátí “other“

JAK NA NPE – OPTIONAL

Optional<U> map(Function<? super T,? extends U> mapper)

Do mapovací operace vstupuje funkce, které provádí převod typu T na U (*T a všichni jeho předci)*
Jestliže v monádě, kterou mapujeme existuje hodnota, tak se vraci monáda s hodnotou, jinak prázdná monáda

Optional<U> flatMap(Function<? super T, Optional<U>> mapper)

Do mapovací operace vstupuje funkce, které provádí převod typu T na monádu s hodnotou typu U (*T a všichni jeho předci, U a všichni jeho potomci)*
Jestliže v monádě, kterou mapujeme existuje hodnota, tak se vraci monáda s hodnotou, jinak prázdná monáda

JAK NA NPE – OPTIONAL

(v1)

```java
public class MonadSample1 {
    //...
    private double multiplyBy2(double n) {
        return n * 2;
    }

    private double divideBy2(double n) {
        return n / 2;
    }

    private double add3(double n) {
        return n + 3;
    }

    private double subtract1(double n) {
        return n - 1;
    }

    public double apply(double n) {
        return subtract1(add3(divideBy2(multiplyBy2(multiplyBy2(n)))));
    }
    //...
}
```

(v2)

```java
public class MonadSample2 {
    //...
    public double apply(double n) {
        double n1 = multiplyBy2(n);
        double n2 = multiplyBy2(n1);
        double n3 = divideBy2(n2);
        double n4 = add3(n3);
        return subtract1(n4);
    }
    //...
```

08 JAK NA NPE – OPTIONAL

```
public class MonadSample3 {
    //...
    public double apply(double n) {
        return Optional.of(n)
          .flatMap(value -> Optional.of(multiplyBy2(value)))
          .flatMap(value -> Optional.of(multiplyBy2(value)))
          .flatMap(value -> Optional.of(divideBy2(value)))
          .flatMap(value -> Optional.of(add3(value)))
          .flatMap(value -> Optional.of(subtract1(value)))
          .get();
    }
    //..
```

08 JAK NA NPE – OPTIONAL

```java
public class MonadSample4 {
    //...
    public boolean leftIdentity() {
         Function<Integer, Optional> mapping = value -> Optional.of(value + 1);
         return Optional.of(3).flatMap(mapping).equals(mapping.apply(3));
     }

     public boolean rightIdentity() {
         return Optional.of(3).flatMap(Optional::of).equals(Optional.of(3));
     }

     public boolean associativity() {
         Function<Integer, Optional> mapping = value -> Optional.of(value + 1);
         Optional leftSide = Optional.of(3).flatMap(mapping).flatMap(Optional::of);
         Optional rightSide = Optional.of(3).flatMap(v -> mapping.apply(v).flatMap(Optional::of));
         return leftSide.equals(rightSide);
     }
     //...
```

08 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;
}
```

08 JAK NA NPE – OPTIONAL - STREAMY

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 (viz. *Dummy Certificate*)

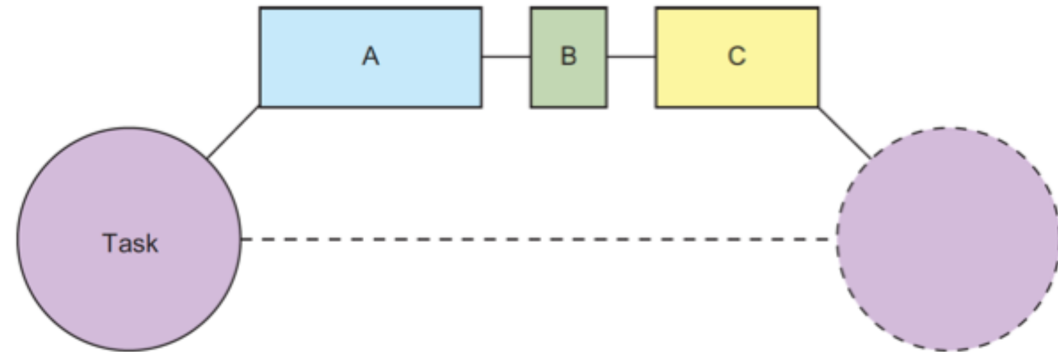
```
static String assessUIProficiencySmart(Candidate candidate) {
    String proficiency = null;

    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));
}
```

09 MONÁDY – Futures/Promise pattern

Jedním z využití Composability, které přínáší monády. je i tzv. future/promise pattern

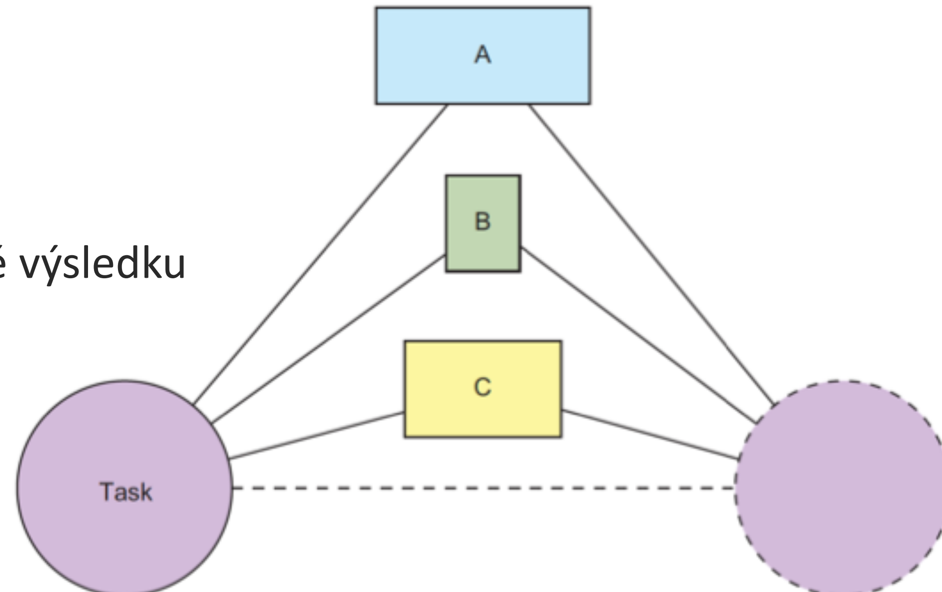
Sekvenční (eager) vyhodnocení

```
ReplyA a = computeA();
ReplyB b = computeB();
ReplyC c = computeC();
Result r = aggregate(a, b, c);
```



Takto poskládáme sekvenci, která se vyhodnotí až při potřebě výsledku

```
Future<ReplyA> a = taskA();
Future<ReplyB> b = taskB();
Future<ReplyC> c = taskC();
Result r = aggregate(a.get(), b.get(), c.get());
```



Future/Promise Pattern

Future je read-only view na proměnnou, která je nastavena někdy v budoucnu pomocí asynchronní funkce - ***Promise****.* Před vlastním nastavením i po nastavení proměnné mohou vznikat různé situace, jejichž ošetření pattern také podporuje - jako např. timeout, vrácení chyby

Použití tohoto patternu výrazně snižuje blokování/latenci v distribuované nebo mnoha threadové aplikace.

Java realizace - Java realizuje tento design pattern pomocí frameworku okolo tříd *Future* a *CompletableFuture. Future* interface byl přidán do Java 5 pro lepší podporu asynchronního zpracování. Nicméně až v Java 8 byl přidán interface CompletableFuture, který navíc umožňuje výpočetní operace řetězit, zpracovávat různé druhy chyb (včetně timeoutu)

09 Funkcionální interface/SAM

Rozhraní, které obsahuje přesně jednu abstraktní metodu, se nazývá *Functional Interface*. Může mít libovolný počet výchozích statických metod, ale může obsahovat pouze jednu abstraktní metodu. Může také deklarovat metody třídy objektů.

Funkční rozhraní je také známé jako Single Abstract Method Interfaces nebo SAM Interfaces.

Web Server jako funkce

Reference: Paper “Your Server as a Function” by Marius Eriksen that explains the functional approach they followed at Twitter to develop web servers.

- *If we look at the problem with a “functional eye,” our application works as an “engine” that transforms some inputs in outputs. We only need to consider what are the specific inputs and outputs. Reasoning in this way we can describe any web service as a single function:*

Http Handler

Request → Response

HttpHandler je funkce, která transformuje Request na Response

Pozn. Pro lepší pochopení příklad z jiné domény, není nutné na zkoušku

Design patterny Future/Promise a monáda jsou masivně využívány při psaní frontendových aplikací, kdy veškerá většina interakce mezi UI komponenty (model, view, controller), interakce se serverem je asynchronní.

Angular

```
$scope.getRequest = function () {
        console.log("I've been pressed!");
        $http.get("http://urlforapi.com/get?name=Elliot")
            .then(function successCallback(response){
                $scope.response = response;
            }, function errorCallback(response){
                console.log("Unable to perform get request");
            }
    .catch(error => this.setState({ error, isLoading: false }));
```

React

```
fetch("http://urlforapi.com/get?name=Elliot")
      .then(response => {
        if (response.ok) {
          return response.json();
        } else {
          throw new Error('Something went wrong ...');
        }
      })
      .then(data => this.setState({ hits: data.hits, isLoading: false }))
      .catch(error => this.setState({ error, isLoading: false }));
```

MONÁDY – Futures/Promise pattern v Java

BOHUŽEL ČITELNOST V JAVA JE VÝRAZNĚ HORŠÍ NEŽ V KOTLINU ČI VE SCALA

Metoda *complete()* - nastaví hodnotu proměnné, do té doby jsou blokovány všechny pokusy o získání její hodnoty

```
public Future<String> calculateAsync() throws InterruptedException {
   CompletableFuture<String> completableFuture = new CompletableFuture<>();
   Executors.newCachedThreadPool().submit(() -> {
       Thread.sleep(5000);
       completableFuture.complete("Hello");
       return null;
   });
   return completableFuture;
}
..
Future<String> completableFuture = calculateAsync();
String result = completableFuture.get(); //Program gets blocked at this line until result provided
..
```

09 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());
```

09 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();
```

09 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());
```

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());}
```

09 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");
});
```

09 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());
```