

 $\Lambda\lambda$ 

## 03

# Funkcionální přístup

- Mutabilita, immutabilita
- Pure funkce
- Funkce první třídy a funkce vyššího řádu
- Lambda expressions
- Closures
- Currying
- Representation transparency
- Lazy evaluation funkcí

## 03 MUTABILITA

Typy, vestavěné nebo uživatelem definované, lze klasifikovat jako:

- **mutable (měnitelné)**
- **immutable (neměnné).**

Mutable typy lze změnit, to znamená, že poskytují operace, které při spuštění způsobují, že výsledky dalších operací na stejném objektu dávají různé výsledky.

=> `Date` je mutable, protože můžete volat `setMonth` a sledovat změnu pomocí operace `getMonth`.

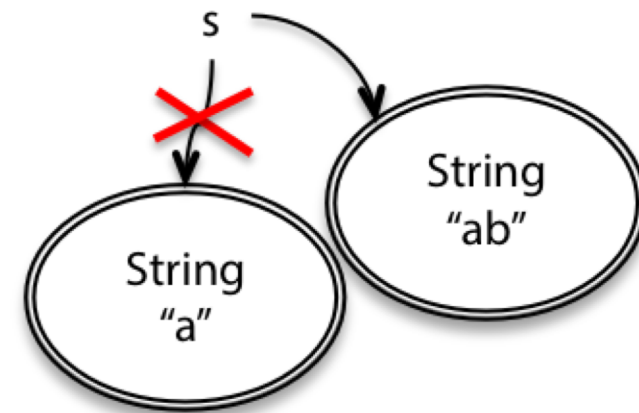
=> `String` je neměnný, protože jeho operace vytvářejí nové objekty `String` spíše než by měnily existující.

# 03 MUTABILITA

Mutable (měnitelné) objekty jsou takové, které obsahují metody, které modifikují hodnotu objektu.

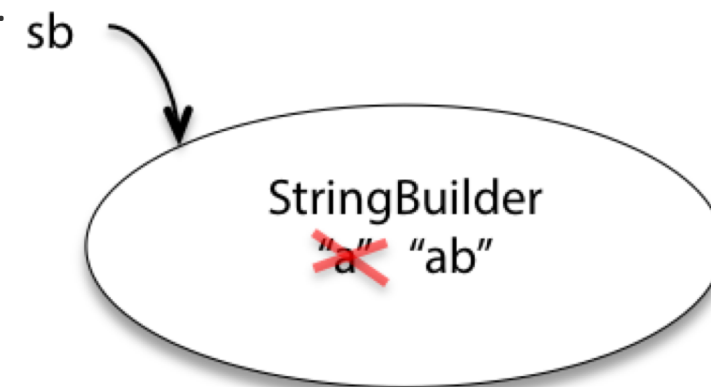
String je immutable, tedy např. při přidání znaků na konec se vždy vytváří nový String.

```
String s = "a";  
s = s.concat("b"); // s+="b" and s=s+"b" also mean the same thing
```



StringBuilder je mutable. Tedy i přidání znaků na konec modifikuje originální objekt.

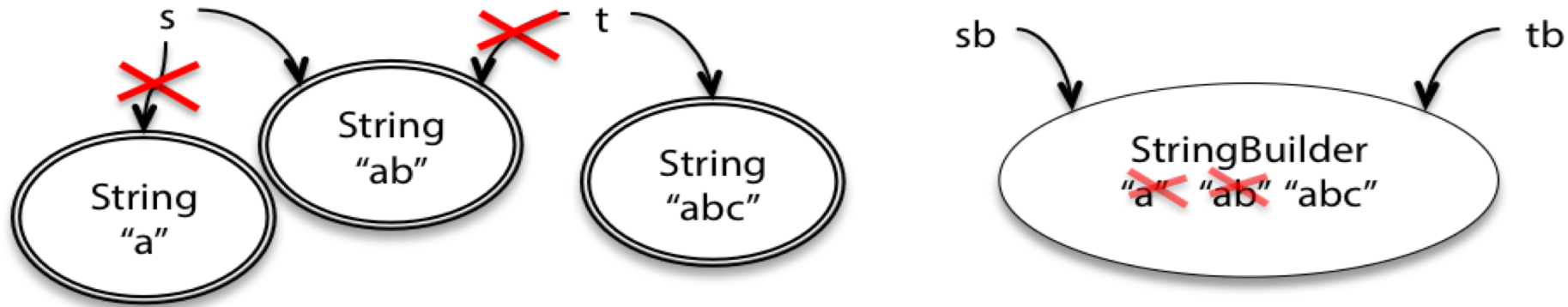
```
StringBuilder sb = new StringBuilder("a");  
sb.append("b");
```



# 03 MUTABILITA

Rozdíl mezi mutable a immutable typy začne být evidentní ve chvíli, kdy máme víc jak jednu referenci na objekt. Např. když `t` ukazuje na stejný `String` jako `s`, a `tb` ukazuje na `StringBuilder` jako `sb`, tak:

```
String t = s;  
t = t + "c";  
StringBuilder tb = sb;  
tb.append("c");
```



## 03 MUTABILITA

Mutable typy jsou “mocnější” z hlediska funkcionality a zpravidla mají i lepší performance - immutable typy při většině volání vytvářejí kopie.

Proč tedy vůbec používat immutable typy?

- **immutable typy jsou méně náchylné pro vznik bugů** - u mutable typů vzniká velké množství chyb kvůli tzv. Aliasingu - na ten samý objekt se odkazujeme z různých míst kódu.
- **kód s immutable typy je jednodušší na pochopení** - čtenář kódu nemusí studovat implementaci, aby zjistil co se vlastně děje na pozadí
- **kód s immutable typy i vlastní immutable typ je jednodušší na upravování** - jestliže máme pod kontrolou co se děje na pozadí v kódu, tak je mnohem bezpečnější zasahovat do již hotového kódu
- **Mutable objekty zesložitují kontrakt (rozhraní) a zhoršují reuse** - při přepoužití musíme zpravidla zasahovat mnohem hlouběji do kódu a řešit situace Aliasingu.

*I Když obecné doporučení je používat immutable typy, tak jsou určité situace, kdy je výhodnější použít mutable typ (např. z performance důvodů) a existují užitečné mutable typy jako např: ArrayList, Hashtable z Java Collections API.*

*Collections API poskytuje metody, které vrací immutable varianty: např. `Collections.unmodifiableList`*

## 03 Klíčová pravidla funkcionálního programování

- **Immutable:** to už známe :-)
- **No implicit state** (bez implicitního stavu): nesmí mít skrytý či implicitní stav. Stav musí být explicitní a transparentní (viditelný):
- **Pure functions:**
  - No side effects*** (bez vedlejších efektů): Funkce či operace nesmí měnit vnější stav (jiné než vlastní lokální proměnné) - funkce pouze vrací hodnotu funkci, která ji volá
  - Idempotence*** = funkce vrací hodnoty, které jsou závislé pouze na argumentech předaných při volání => nezávisí tedy na ničem jiném. Pokud funkci zavoláte vícekrát s těmi samými parametry, tak bude vždy vracet to samé

**Pure funkcionální jazyky neumožňují mutabilitu a side efekty - příklad Haskell**  
**Vyšší jazyky to povolují - příklad Java**

## 03 PURE FUNKCE

Příklad pure funkce:

```
public class ObjectWithPureFunction{  
    public int sum(int a, int b) {  
        return a + b;  
    }  
}
```

Příklad **non** pure funkce:

```
public class ObjectWithNonPureFunction{  
    private int value = 0;  
    public int add(int nextValue) {  
        this.value += nextValue;  
        return this.value;  
    }  
}
```

## 03 LAMBDA EXPRESSIONS

Lambda expression je forma ve tvaru: (seznam argumentů funkce) -> tělo funkce

```
/* Java 1.8+ Funkce, která sečte dvě čísla */
(int x, int y) -> x + y

/* Bezparametrická funkce */
() -> 42

/* Procedura */
(String s) -> { System.out.println(s); }

/* Komparátor */
List<Person> personList = Person.createShortList();
Collections.sort(personList, (Person p1, Person p2) -> p1.getSurName().compareTo(p2.getSurName()));
```

Lambda výrazy se používají především k definování implementace funkčního **rozhraní s jedinou metodou** tzv. **inline formou** což vede k výrazné redukci kódu a přináší např. do Javy některé výhody funkcionálního programování.

*Pozn. Lambda calculus je formální systém matematické logiky a informatiky pro vyjádření výpočtu pomocí bindingu proměnných a jejich substituce - nezaměňovat s lambda expressions*



## 03 Funkce první třídy a funkce vyššího řádu

Objektem **první třídy** (first-class citizen) v programovacích jazycích je entita, která podporuje následující operace: být předána jako parametr, přiřazená proměnné a být vrácená z funkce. Funkce první třídy je tedy taková funkce, která splňuje výše uvedené vlastnosti.

*Pozn. Metody a třídy, jelikož to nejsou hodnoty, tak jsou považovány za objekty druhé třídy.*

**Funkce vyššího řádu** je funkce, které splňuje přinejmenším jednu z vlastností:

- Jedním či více parametry je funkce
- Vrací funkci jako parametr

## 03 Funkce první třídy a funkce vyššího řádu

Klasický přístup:

```
public List filterPersonByAge(List<Person> list) {  
    List result = new ArrayList();  
    for (Person person : list) {  
        if(p.age > 65){  
            result.add(person);  
        }  
    }  
    return result;  
}
```

Filtruji a vracím každého, kdo je starší než 65 let. Problém je, že když chci filtrovat podle jiného atributu, tak musím celý tento kód zduplikovat, abych modifikoval pouze jednu řádku kódu.

## 03 Funkce první třídy a funkce vyššího řádu

Funkcionální přístup:

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;
public class HigherOrderFunctionExample {

    public List filterPerson(List<Person> list, Predicate<Person> p) {
        List result = new ArrayList();
        for (Person person : list) {
            if(p.test(person)) {result.add(person);}
        }
        return result;
    }
    public static boolean ageFilter(Person p){
        return p.getAge() > 65;
    }
}
```

Přidali jsme nový parametr typu *Predicate*, který obsahuje podmínku, kterou testujeme. Dále pak metoda *ageFilter*, kterou vkládáme jako parametr *p*.

## 03 Funkce první třídy a funkce vyššího řádu

```
public class Test {
    public static void main(String[] args){
        Person[] array = {new Person(13), new Person(70)};
        List<Person> personList = Arrays.asList(array);
        HigherOrderFunctionExample hof = new HigherOrderFunctionExample();

        //předání funkce definované uvnitř HigherOrderFunctionExample
        hof.filterPerson(personList, HigherOrderFunctionExample::ageFilter);

        //předání lambda expression
        hof.filterPerson(personList, p-> p.getAge() > 65);
    }
}
```

Jestliže chceme filtrovat podle jiného atributu, tak uděláme drobnou změnu do implementace filtru a vlastní kód na filtrování je přepoužít.

Nebo můžeme jednoduše posílat do filtrovací operace jiné filtry

# 03 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);  
}
```

- Predicate

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

- UnaryOperator T->T
- BinaryOperator T,S->R
- Supplier ()->T
- Consumer T->()



# 03 Funkce první třídy a funkce vyššího řádu

Funkce je regulérní typ v Kotlinu a není potřeba podobný cirkus jako v Javě

## Kotlin

```
fun <T> ArrayList<T>.filterOnCondition(condition: (T) -> Boolean): ArrayList<T>{
    val result = arrayListOf<T>()
    for (item in this){
        if (condition(item)){
            result.add(item)
        }
    }
    return result
}

fun isMultipleOf (number: Int, multipleOf : Int): Boolean{
    return number % multipleOf == 0
}

var list = arrayListOf<Int>()
for (number in 1..10){
    list.add(number)
}
var resultList = list.filterOnCondition { isMultipleOf(it, 5) }
```

## 03 CLOSURES

Closure je funkce, která si při deklaraci vytvoří lokální proměnnou, kterou si vezme z kontextu ve kterém je vytvořena

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

public class Closure {
    int global_var = 0;
    // this is a higher-order-function that returns an instance of Function interface
    Function<Integer, Integer> add(final int x) {
        // The lambda expression is returned here as closure
        // x is obtained from the outer scope of this method which is declared as final
        int local_var = 6;
        return y -> x + y;
        //return y -> x + y + global_var + local_var; STILL POSSIBLE
        //return y -> x + y + global_var + local_var; STILL POSSIBLE
        //return y -> global_var = x + y; STILL POSSIBLE to modify global variable
        //return y -> local_var = x + y; NOT POSSIBLE
    }
    public static void main(String[] args) {
        Closure sample = new Closure();
        // we are currying the add method to create more variations
        Function<Integer, Integer> add10 = sample.add(10);
        Function<Integer, Integer> add20 = sample.add(20);
        Function<Integer, Integer> add30 = sample.add(30);
        System.out.println(add10.apply(5)); //15
        System.out.println(add20.apply(5)); //25
        System.out.println(add30.apply(5)); //35
    }
}
```

## 03 CURRYING

Currying spočívá ve vyhodnocování argumentů funkce per partes, kdy po každém kroku získáme funkci, která má o jeden argument méně.

Např. pro funkci

$$f(x, y, z) = x * y + z$$

můžeme aplikovat argumenty 3, 4, 5 a dostaneme:

$$f(3, 4, 5) = 3 * 4 + 5 = 17$$

Současně ale můžeme aplikovat pouze 3 a získáme novou funkci  $f$

$$(3, y, z) = g(y, z) = 3 * y + z$$

currying podruhé pro 4 nám dá:

$$g(4, z) = h(z) = 3 * 4 + z$$



## 03 CURRYING

Příklad vytvoření složené funkce při deklaraci:

```
public class Curryng {  
    public void currying() {  
        // Create a function that adds 2 integers  
        BiFunction<Integer,Integer,Integer> adder = ( a, b ) -> a + b ;  
        // And a function that takes an integer and returns a function  
        Function<Integer,Function<Integer,Integer>> currier = a -> b -> adder.apply( a, b ) ;  
        // Call apply 4 to currier (to get a function back)  
        Function<Integer,Integer> curried = currier.apply( 4 ) ;  
        // Results  
        System.out.printf( "Curry : %d\n", curried.apply( 3 ) ) ; // ( 4 + 3 )  
    }  
}
```

## 03 CURRYING

Vytvoření složené funkce ex post po jejich deklaraci:

```
public void composition() {
    // A function that adds 3
    Function<Integer,Integer> add3 = (a) -> a + 3 ;
    // And a function that multiplies by 2
    Function<Integer,Integer> times2 = (a) -> a * 2 ;
    // Compose add with times
    Function<Integer,Integer> composedA = add3.compose( times2 ) ;
    // And compose times with add
    Function<Integer,Integer> composedB = times2.compose( add3 ) ;
    // Results
    System.out.printf( "Times then add: %d\n", composedA.apply( 6 ) ) ; // ( 6 * 2 ) + 3
    System.out.printf( "Add then times: %d\n", composedB.apply( 6 ) ) ; // ( 6 + 3 ) * 2
}
public static void main( String[] args ) {
    new Currying().currying() ;
    new Currying().composition() ;
}
}
```

## 03 REFERENTIAL TRANSPARENCY

Výraz se nazývá referenčně transparentní, pokud jej lze nahradit odpovídající hodnotou (a naopak), aniž by se změnilo chování programu.

Vychází z idempotentnosti pure funkcí. Hezkým důsledkem toho je, že můžeme volání funkce nahradit hodnotou, kterou funkce vrátila naposledy. Tzv. *memoizace* nebo *caching funkčního volání*, abychom nemuseli provádět vícekrát tu samou funkci

## 03 LAZY EVALUACE

Process kdy zpozdím vyhodnocení výrazu až do doby než potřebuju výsledek. Opak **lazy** je **eager**

Java ve většině případů funguje eager - kromě operandů **&&**, **||** and **?:**, které jsou lazy. Lambda expressions umožňují psát “lazy” kód v Java.

```
public class EagerSample {
    public static void main(String[] args) {
        System.out.println(addOrMultiply(true, add(4), multiply(4))); // 8
        System.out.println(addOrMultiply(false, add(4), multiply(4))); // 16
    }
    static int add(int x) {
        System.out.println("executing add");
        return x + x;
    }
    static int multiply(int x) {
        System.out.println("executing multiply");
        return x * x;
    }
    static int addOrMultiply(boolean add, int onAdd, int onMultiply) {
        return (add) ? onAdd : onMultiply;
    }
}
```

```
executing add
executing multiply
8
executing add
executing multiply
16
```

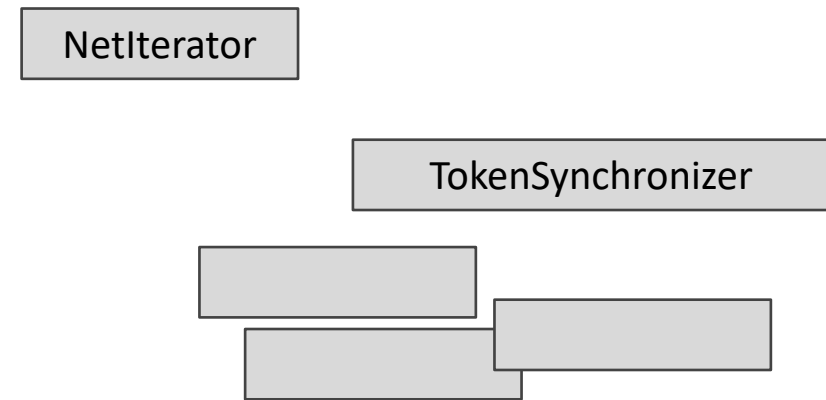
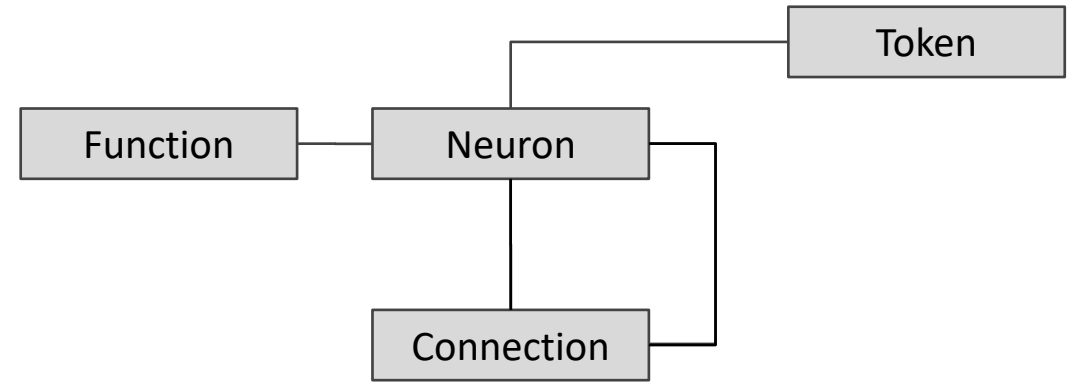
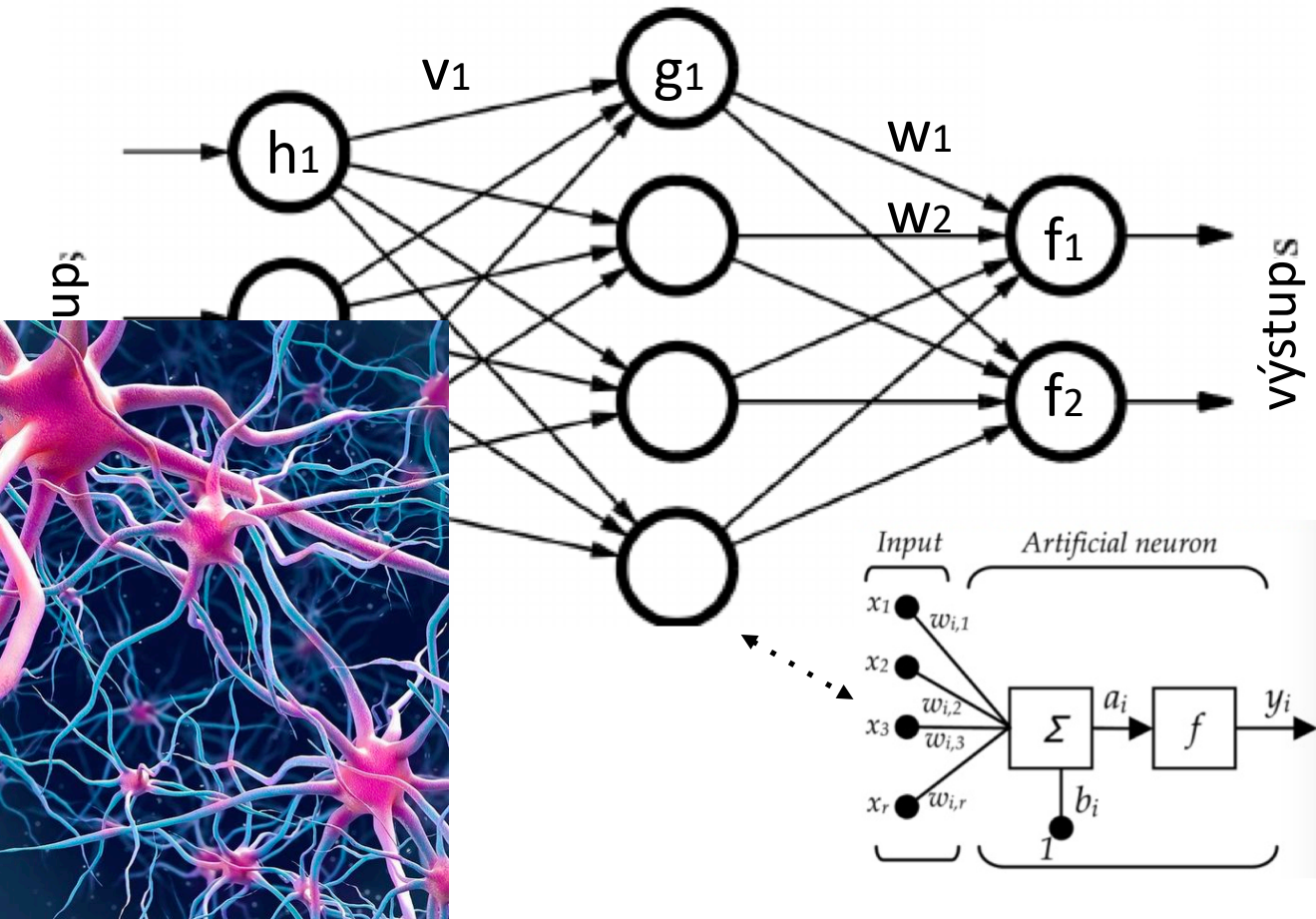
## 03 LAZY EVALUATE

```
public class LazySample {
    public static void main(String[] args) {
        // This is a lambda expression behaving as a closure
        Function<Integer, Integer> add = t -> {
            System.out.println("executing add");
            return t + t;
        };
        // This is a lambda expression behaving as a closure
        Function<Integer, Integer> multiply = t -> {
            System.out.println("executing multiply");
            return t * t;
        };
        // Lambda closures are passed instead of plain functions
        System.out.println(addOrMultiply(true, add, multiply, 4));
        System.out.println(addOrMultiply(false, add, multiply, 4));
    }
    // This is a higher-order-function
    static <T, R> R addOrMultiply( boolean add, Function<T, R> onAdd,
                                Function<T, R> onMultiply, T t) {
        return (add ? onAdd.apply(t) : onMultiply.apply(t));
    }
}
```

```
executing add
8
executing multiply
16
```

# 03 SYSTÉMY NEVHODNÉ PRO OOP

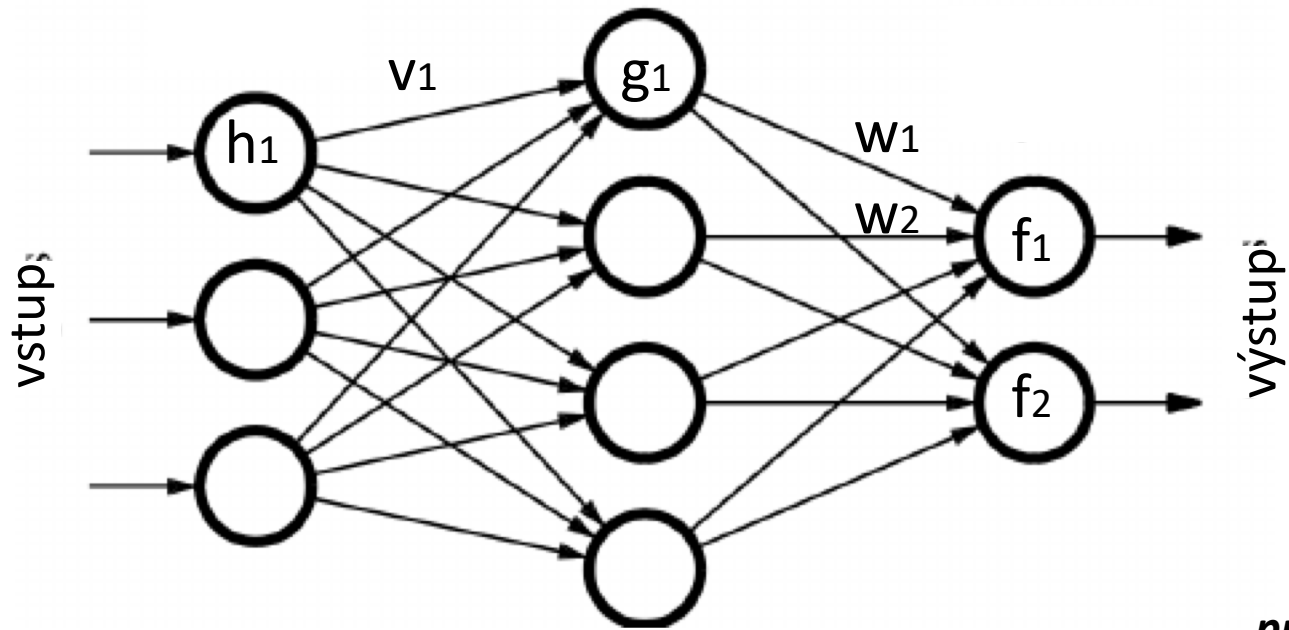
Chci realizovat aplikaci pro výpočet v neuronové síti (nebo jiném výpočetním grafu). Na vstup mi přichází signál, který se přenáší po vazbách mezi nody (neurony). V neuronu se sečtou signály ze všech příchozích vazeb (s příslušnými váhami) a na tento součet se aplikuje funkce. Pak signál pokračuje k dalším neuronům až dorazí na výstup.



... a přidávám další třídy, abych byl schopný systém realizovat pomocí OOP. A to jsem ještě nenapsal ani řádku kódu

# Jak budu modelovat neuronovou síť?

... kde výpočet hodnot na výstupu realizují pomocí pomocí pár řádků kódu a bez nutnosti složité objektové reprezentace



**=> Systém nevhodný pro realizaci pomocí OOP**

```
public double f(double in){  
    return Math.atan(in)
```

```
}
```

```
Vystup1 = f ( w1 * g1( v1 * h1 (...)) + w2 *g2(...) ...)
```

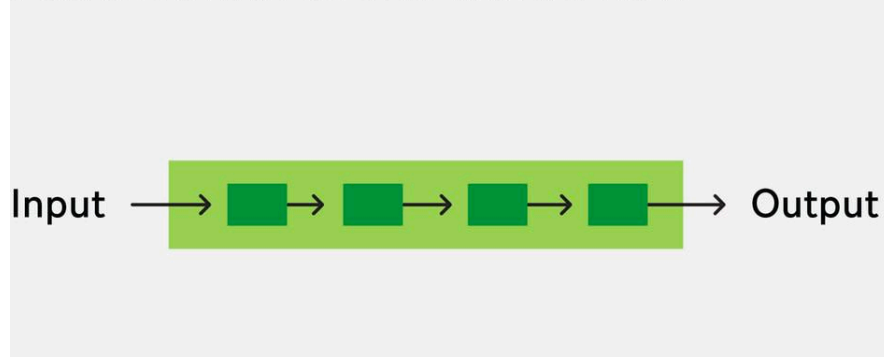
```
Vystup2 = f (...
```



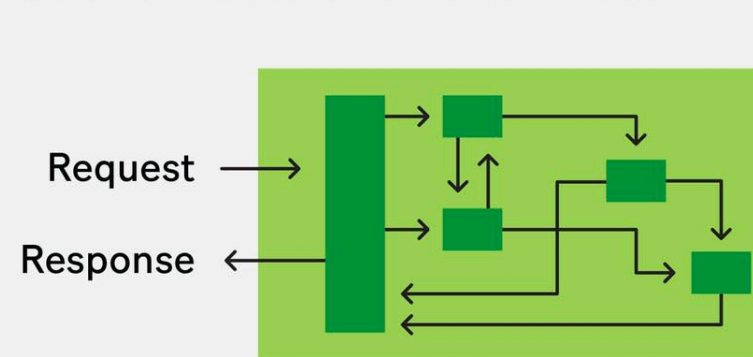
# 03 Srovnání Objektového a Funkcionálního přístupu

OBLASTI SROVNÁNÍ	Funkcionální programování	OOP
<b>Přístup</b>	Vyhodnocování funkcí	Objekty a interakce s nimi
<b>Data</b>	Immutable data	Mutable data
<b>Support</b>	Out of box podporuje parallelismus	Out of box nepodporuje parallelismus
<b>Exekuce</b>	Operace mohou být volány v jakémkoliv pořadí	Operace musí být volány v předem definovaném pořadí
<b>Iterace</b>	Rekurze	Iterování (cykly)
<b>Stavební kameny</b>	Základní stavební kameny jsou proměnné, funkce, moduly	Základní stavební kameny jsou objekty, jejich metody a packages
<b>Použití</b>	Málo věcí s mnoho operacemi	Mnoho věcí s málo operacemi

FUNCTIONAL-STYLE WORKFLOW



OBJECT-ORIENTED WORKFLOW





# 03 FUNKCIONÁLNÍ PŘÍSTUP

