

# OMO

## 1 - Klíčové koncepty modelování systémů I

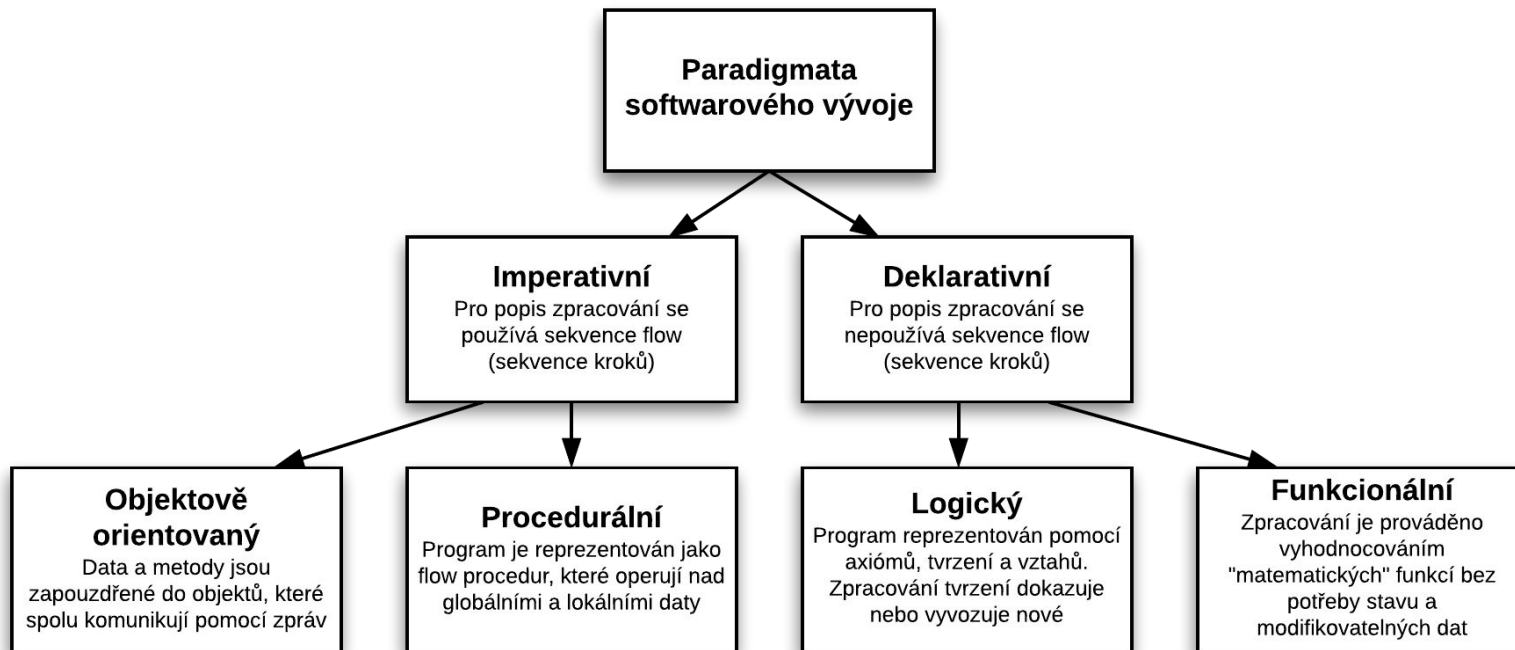
- Programovací paradigmatá
- Deklarativní versus imperativní reprezentace
- Práce s komplexitou
- Dekompozice
- Hierarchie
- Základní abstrakce (jmenné, datové, funkcionální)

---

Ing. David Kadleček, PhD.

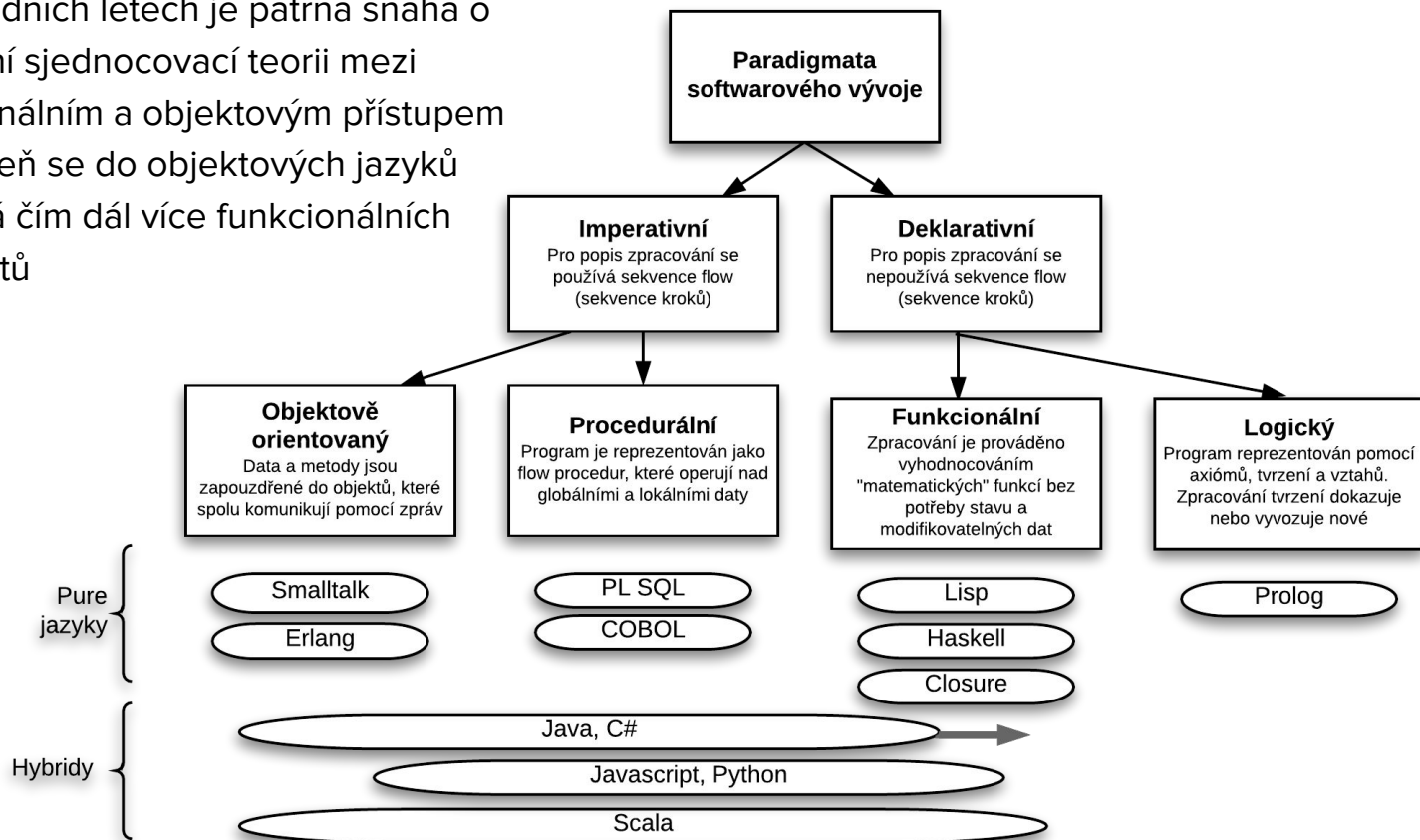
[kadlecd@fel.cvut.cz](mailto:kadlecd@fel.cvut.cz), [david.kadlecek@cz.ibm.com](mailto:david.kadlecek@cz.ibm.com)

# Paradigmata softwarového vývoje



# Paradigmata softwarového vývoje

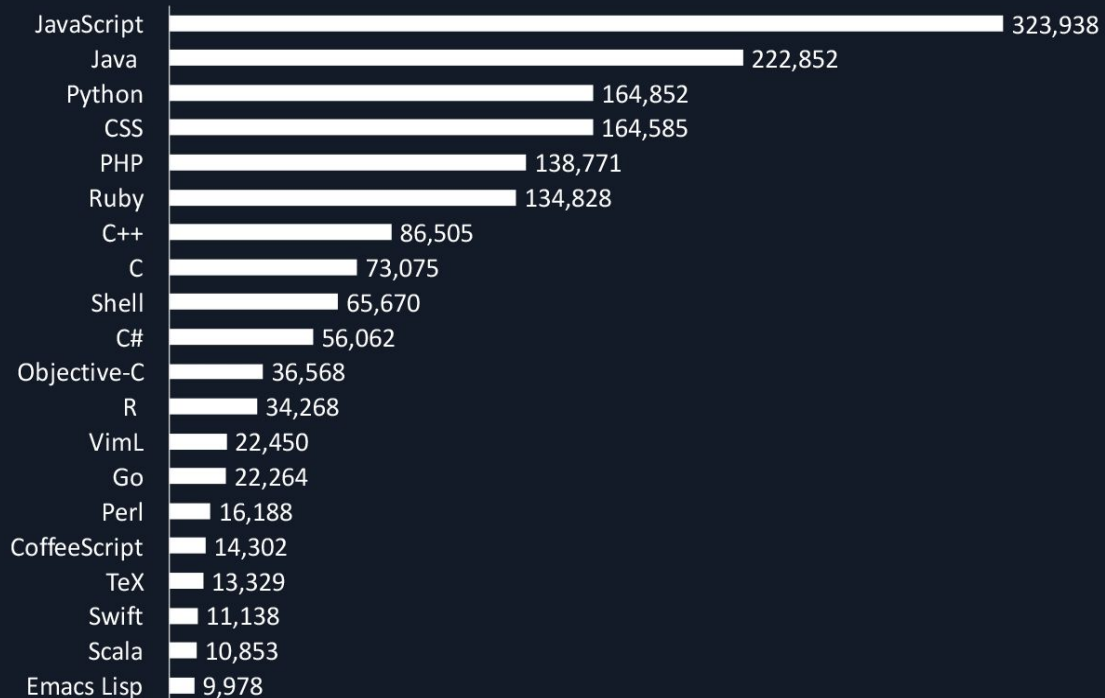
V posledních letech je patrná snaha o nalezení sjednocovací teorii mezi funkcionálním a objektovým přístupem a zároveň se do objektových jazyků dostává čím dál více funkcionálních konceptů



# Trendy - nejvíce aktivity na GitHub

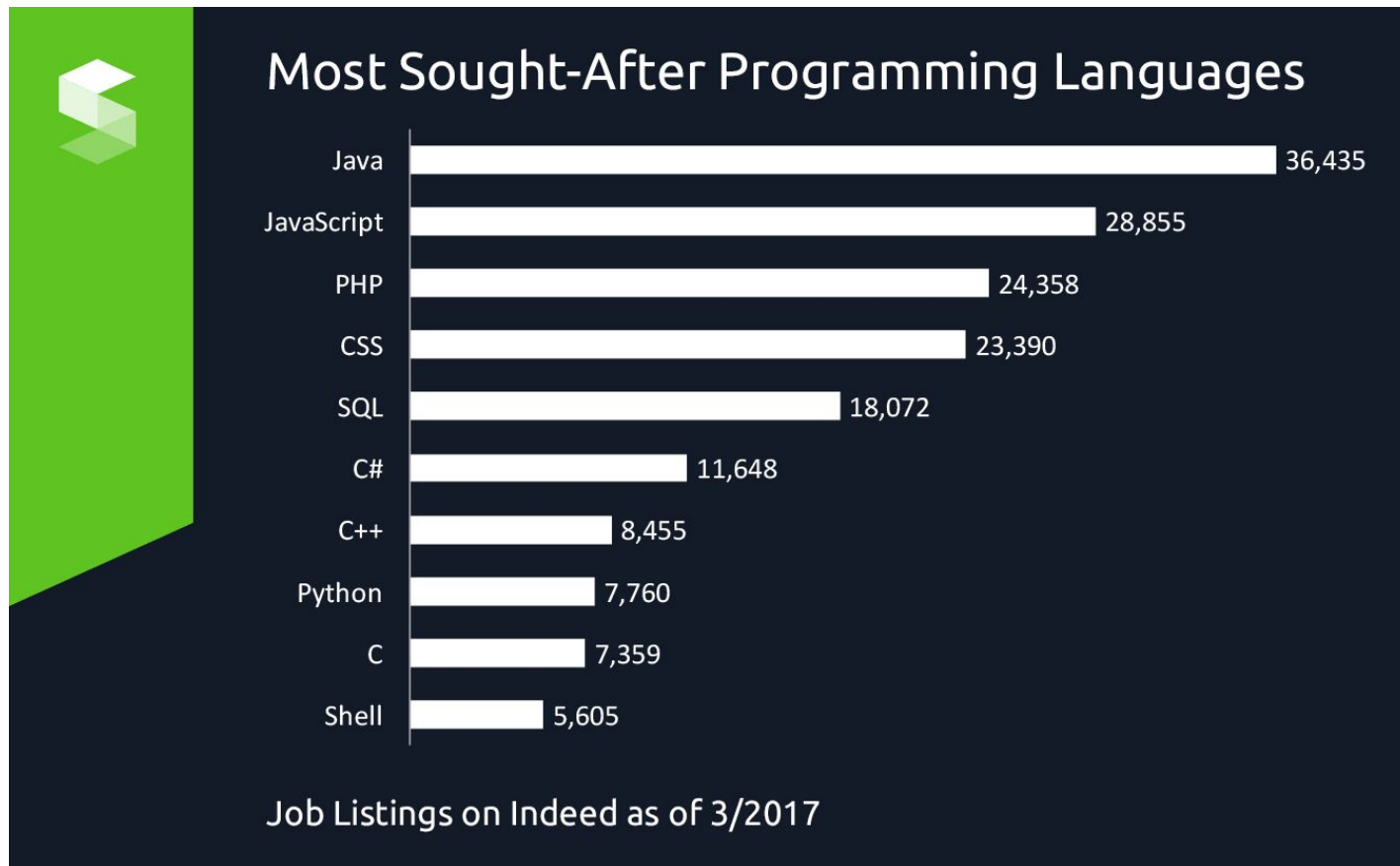


## Languages With the Most Active Repositories in GitHub



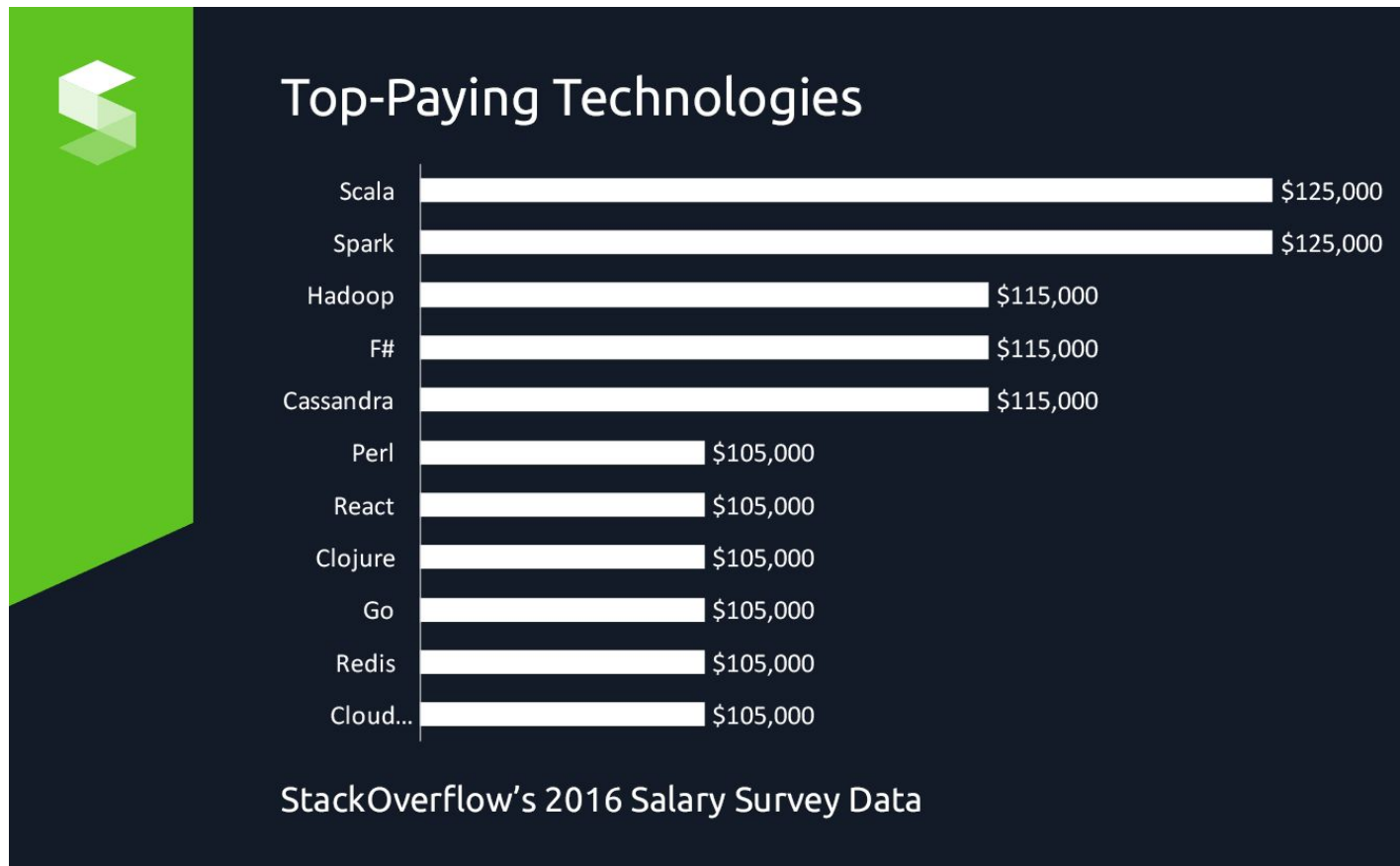
Source: [www.indeed.com](http://www.indeed.com) 2017

# Trendy - nejvíce nabídek práce



Source: [www.indeed.com](http://www.indeed.com) 2017

# Trendy - nejvyšší platy



Source: [www.indeed.com](http://www.indeed.com) 2017

# Deklarativní versus Imperativní reprezentace

- Výkres v geometrii - popisuje útvary a vztahy mezi nimi = “**What Is**” znalost, **Deklarativní reprezentace**
- Výměna oleje v autě - popisuje postup jako sekvenci činností = “**How To**” znalost, **Imperativní reprezentace**

$$\sqrt{x} \text{ je } y \text{ takové, že } y^2 = x \text{ a } y \geq 0$$

Odmocnina ze dvou má deklarativní vyjádření:  $\sqrt{x}$ . To samé lze imperativně vyjádřit ve formě postupu, kterým odmocninu ze dvou získám. Jedním z postupů jak získat aproximaci odmocniny ze dvou je např:

- *Odhadnout výsledek - G*
- *Zlepšit odhad zprůměrováním G a x/G*
- *Zlepšovat odhad dokud není dostatečně dobrý*

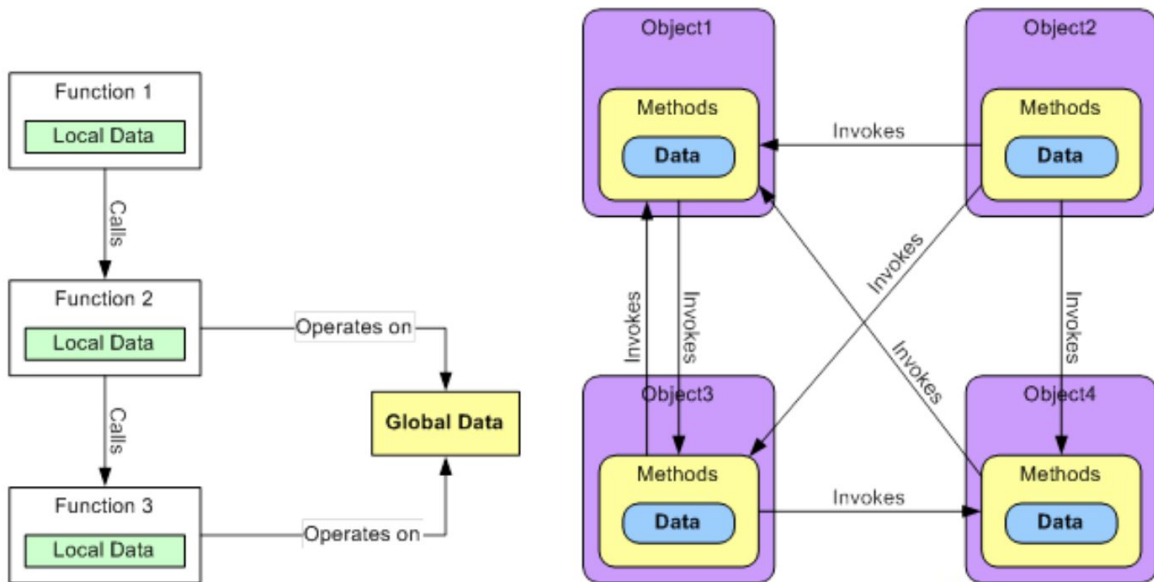
$\sqrt{x}$  pro  $x=2$

$x = 2$	$G = 1$
$x/G = 2$	$G = \frac{1}{2}(1 + 2) = 3/2$
$x/G = 4/3$	$G = \frac{1}{2}(3/2 + 4/3) = 17/12$
$x/G = 24/17$	$G = \frac{1}{2}(17/12 + 24/17) = 577/408 = 1.4142$

# Procedurální versus Objektový přístup

V procedurálním přístupu jsou základním stavebním kamenem programu procedury, které pracují nad lokálními a globálními daty, data mohou být organizována do záznamů (Record).

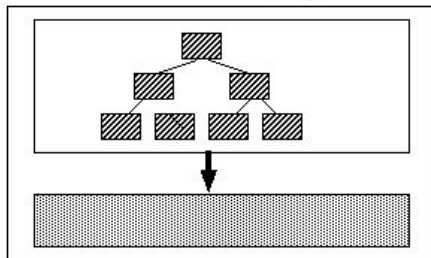
V objektovém přístupu jsou základním stavebním kamenem objekty, které zapouzdřují (a také kontrolují) volání metod a práci s daty, komunikují spolu pomocí zasílání zpráv.





# Rozdílná koncepce

## Procedurální přístup

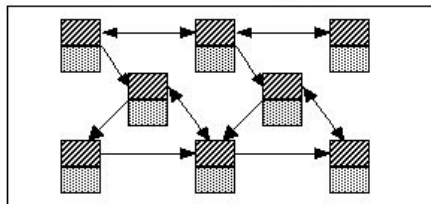


Exekuce zahrnuje provádění kódu, který operuje nad daty

 Code

 Data

## Objektový přístup

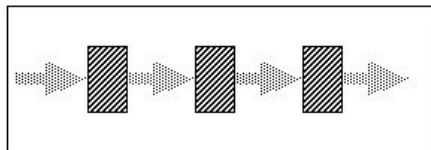


Objekt zapouzdřuje kód i data


 Code  
 Data

Výpočet zahrnuje interakci mezi objekty

## Funkcionální přístup



Data neexistují nezávisle

 Code (Functions)

Exekuce zahrnuje zřetěžené volání funkcí

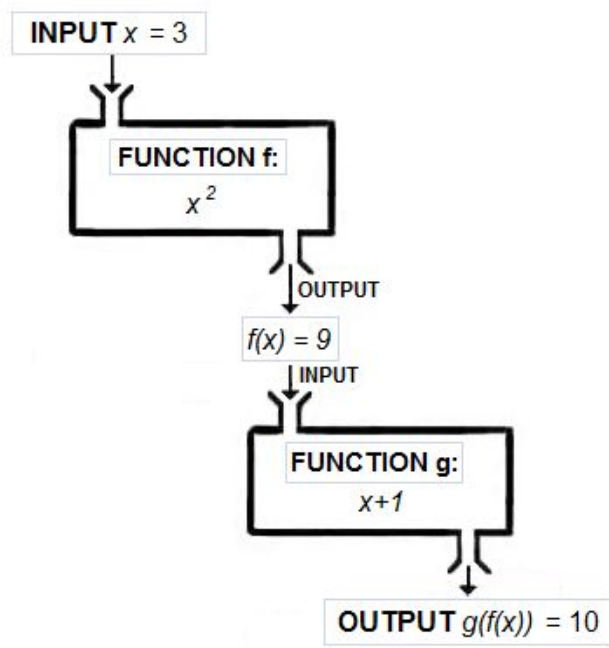
# Procedurální (a obecně imperativní) versus Funkcionální přístup

## Procedurální přístup

1. Udělej něco
2. Udělej něco jiného
3. Zkontroluj to
4. Vypiš to do souboru
5. Stalo se něco?
6. Jestliže ano, udělej toto
7. ...

## Funkcionální přístup

$f(g(h(j(k(l(x))))))$



# Procedurální (a obecně imperativní) versus Funkcionální přístup

Ve funkcionálním přístupu jsou základním stavebním kamenem programu funkce, se kterými se pracuje jako s hodnotami.

## **Procedurální (imperativní) přístup**

Mutable data, manipuluje se se stavem a objekty, iterace

- + Jednoduché porozumět kódu
- + Jednoduchý debugging
- Delší kód
- Side efekty při volání procedury
- Horší škálování a multithreading

## **Funkční přístup**

Immutable data, funkce vyššího řádu, manipulace s funkcemi a datovými množinami, rekurze

- + Kratší kód
- + Lepší škálování
- + Žádné side efekty při volání funkce
- Horší porozumění kódu
- Pomalejší pro jednoduché volání
- Pomalejší učící křivka

# Principy dekompozice

Pokud dekomponuji systém do modulů, tak aplikuji následující principy:

- **Cohesion (princip soudržnosti)** - spojuji funkcionalitu podle podobnosti. Atributy podobnosti volím intuitivně podle toho jakým způsobem a za jakým účelem budu modul používat. Atributem podobnosti může být doména, entita, uživatel atd. Intuitivní pozitivní příklad je, že mám skupinu lidí s různými profesemi, v různých počtech a ty rozdělím na skupiny, kde každá realizuje skupinu činností. Intuitivní negativní příklad je jediná společná operace provádějící vše - *operaceBůh*.
- **Coupling (provázanost)** - snažím se dosáhnout velmi kohezních modulů s co nejvolnějšími vazbami. Vazby uvnitř modulu jsou mnohem silnější než vazby mezi moduly.
- **Reusability (přepoužitelnost)** - modul je "opracován" tak, abych ho mohl přepoužívat v různých kontextech. Kontextem rozumím volat z různých modulů, v různých fázích životního cyklu aplikace a pro různé domény - např. použití toho samého modulu pro *spotřební úvěry* a *hypotéky*.
- **DRY (Don't Repeat Yourself)** - stejný kód (stejný kus funkcionality) by se neměl nacházet ve více modulech.
- **Flexibility isolation (izolace flexibility)** - jestliže budu muset implementovat změnu, tak dopad této změny bude omezen na vazby mezi moduly nebo minimum modulů.
- **Encapsulation (zapouzdření)** - data modulů jsou privátní, k modulům přistupuji ne pomocí jejich dat, ale pomocí povolených operací

# Kvalita software - multikriteriální optimalizační problém

Při vývoji software jsou **čas**, **peníze**, **scope** a **kvalita** navzájem provázány. Jelikož jsme v předmětu OMO, tak se zde budeme zabývat primárně kvalitou.

Kritéria kvality softwarového systému jsou:

- **Flexibilita** - množství a složitost změn, které musím v systému provést proto, aby fungoval i pro jiné scénáře
- **Přepoužitelnost** - použitelnost systému pro konstrukci v různých aplikacích
- **Rozšiřitelnost** - schopnost systému adaptovat se na změny ve specifikaci (rozšiřují funkcionalitu)
- **Robustnost** - schopnost systému reagovat na nepředpokládané situace
- **Kompatibilita** - jednoduchost kombinování softwarových komponent mezi sebou
- **Použitelnost** - jednoduchost použití systému uživatelem nebo jiným systémem
- **Efektivnost** - minimalizace požadavků na zdroje (hardwarové, lidské, finanční)
- **Škálování** - schopnost systému fungovat při narůstající zátěži
- **Portovatelnost** - náročnost přenesení software do jiného hardwarového a softwarového prostředí

# Kvalita software - multikriteriální optimalizační problém

Kvalitu softwarového systému (komponenty) optimalizují nejenom proti času a penězům, ale také vůči ostatním kvalitativním kritériím navzájem. Nelze najít optimální řešení, existují pouze sub optimální řešení, jelikož kritéria jsou ve vzájemné kontradikci, např:

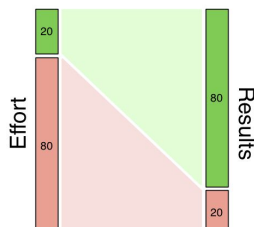
- Zvyšování flexibility zvyšuje komplexitu a tím pádem náklady a čas
- Zvyšování přepoužitelnosti snižuje použitelnost (musím zvýšit obecnost systému)
- Vyšší granularita zvyšuje použitelnost, ale snižuje přepoužitelnost

Jak se tedy rozhodovat? Paretové principy zde fungují jako všude jinde. Snažím se tedy najít místa, kde s minimálním úsilím dosahuji maximálního efektu, zastavuji ve chvíli, kdy jsou pro mě další zlepšení příliš drahá. Vylepšuji jednotlivé vlastnosti systému nezávisle dokud nedosáhnu stavu, že dalším zlepšováním jednoho kritéria degraduji ostatní.

## Optimální poměr cena výkon

### The 80-20 Rule

"For many events, roughly 80% of the effects come from 20% of the causes." - Pareto



## Paretové princip z teorie her

**Paretové zlepšení** - alokace může být paretoovsky zlepšena pokud existuje jiná alokace při které jeden hráč na tom může být lépe aniž by si žádný další hráč nepohoršil

**Paretoovsky optimální** - alokace je paretoovsky optimální jestliže není možné paretové zlepšení.

# Práce s komplexitou

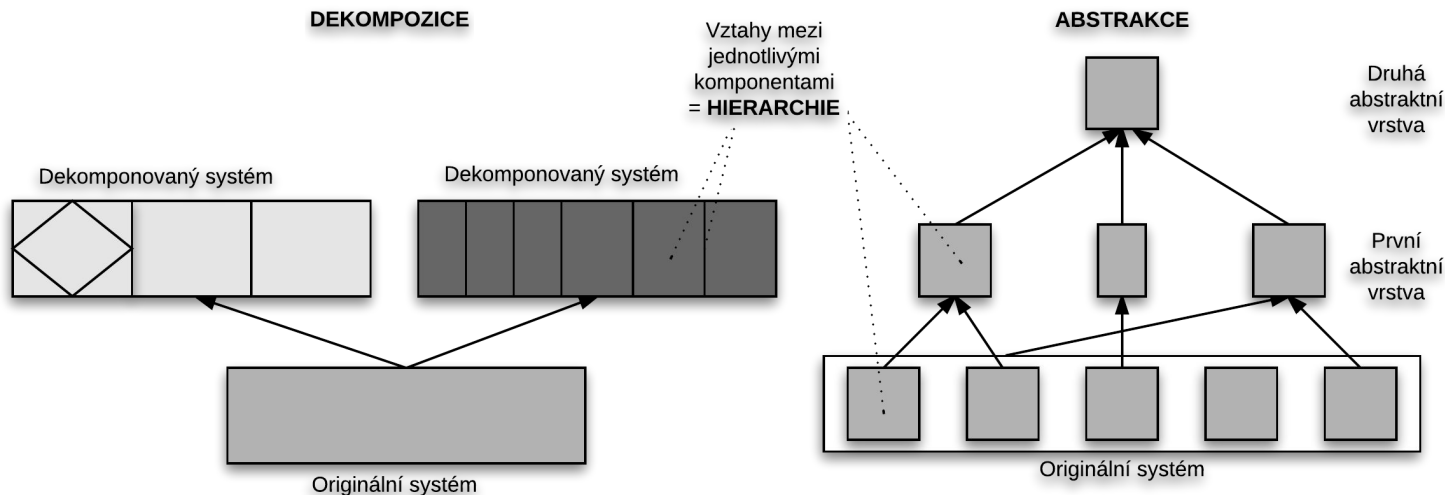
Co dělám, když mám problém (softwarový, matematický nebo i osobní) se kterým si nemohu poradit?

=> snažím se **problém strukturovat**

Jak mohu problém strukturovat?

=> pomocí **abstrakce, dekompozice a hierarchie**.

**Dekompozicí rozkládám systém na menší komponenty, abstrakcí skrývám komplexitu komponent do jednodušších, hierarchií tyto komponenty provazují mezi sebou.**



# Práce s komplexitou

Komplexní systémy mají následující vlastnosti:

## **Hierarchická struktura**

Architektura komplexního systému je funkcí jeho komponent a hierarchického vztahu mezi nimi. ***Téměř dekomponovatelný systém*** = mezi jeho komponentami jsou vazby, které jsou slabé, ale nezanedbatelné.

## **Primitiva v systému jsou relativní**

Rozklad systému na primitiva je relativní - co je primitivní pro jednoho uživatele může být příliš abstraktní pro jiného

## **Separation of concerns**

Spolu související části a vlastnosti jsou drženy spolu a odděleny od ostatních. Zjednodušeně řečeno nemícháme jablka a hrušky.

## **Common patterns**

Hierarchické systémy jsou obvykle sestavené z omezeného množství typů komponent, které se opakují v různých kombinacích a uspořádáních. Tyto systémy obsahují opakující se paternity ve kterých se přepoužívají komponenty omezeného množství typů.

## **Stabilní přechodové formy**

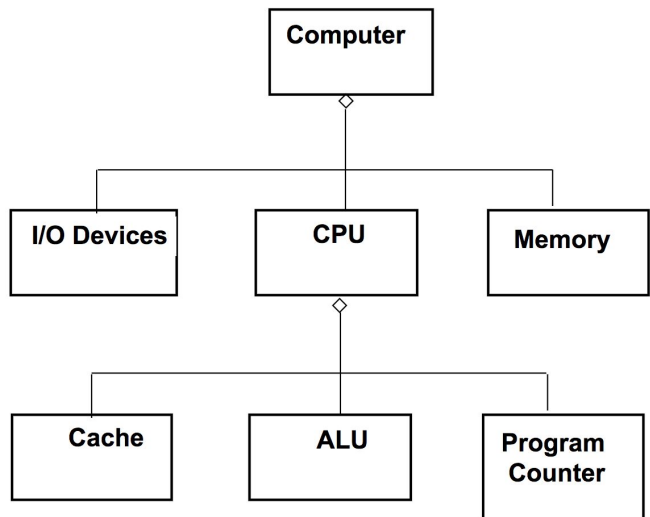
Komplexní systém, který funguje byl téměř jistě odvozen z jednoduššího systému, který funguje. Komplexní systém navržený od nuly zpravidla nefunguje a nelze ho ani rozfungovat patchováním.



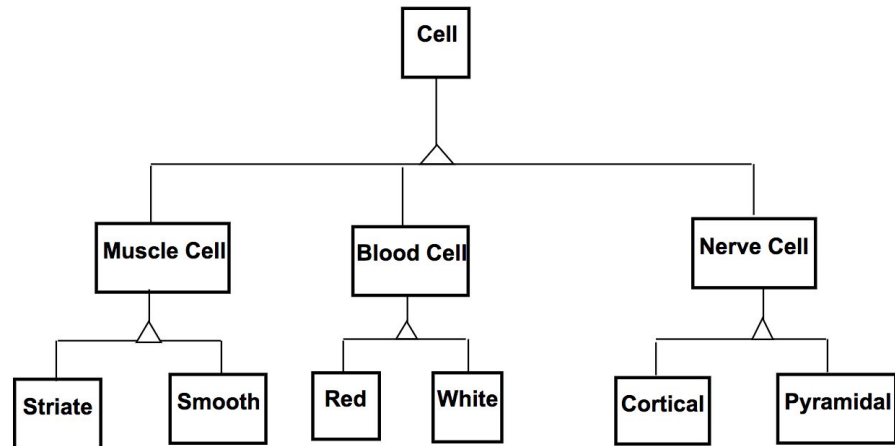
# Hierarchie

Hierarchie je určena vazbami mezi komponentami systému

## Part of hierarchie



## Is kind of hierarchie:

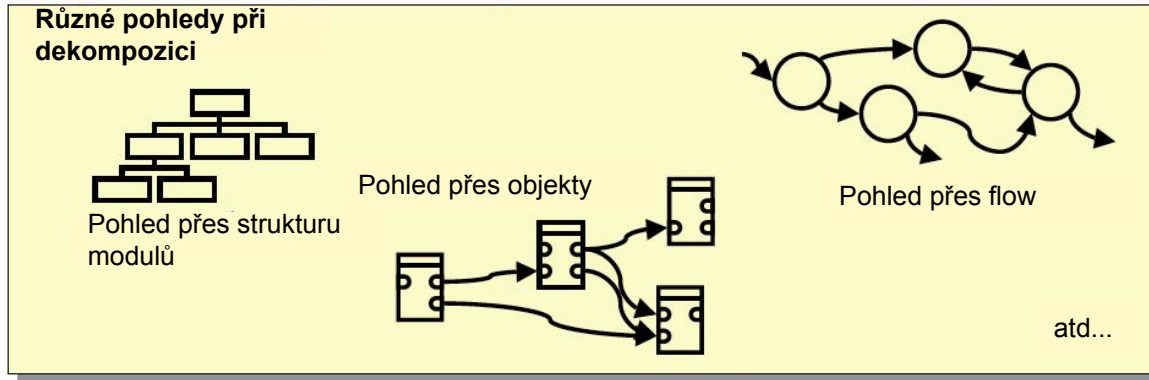


# Dekompozice

K větším problémům přistupuji způsobem "**Rozděl a panuj**" tak, aby:

- Každý podproblém měl přibližně stejnou úroveň detailu
- Každý podproblém byl řešitelný samostatně
- Řešení podproblémů lze zkombinovat tak, abych tím vyřešil celý problém

System zpravidla představuje n rozměrný problém, který nelze popsat jedním pohledem. Místo toho potřebuji **několik pohledů**. Dekompozici tak mohu provést pro tyto různé pohledy. Viz UML definuje několik druhů UML diagramů, každý navržen pro modelování jiného pohledu na systém.



# Abstrakce

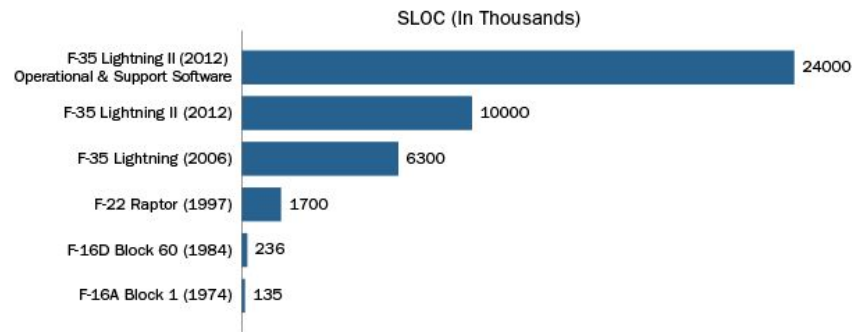
Abstrakce je jednodušší reprezentace systému pomocí které na něj nahlížím nebo k němu přistupuji aniž bych znal jeho detaily

**Zjednodušeně řečeno potřebujeme schopnost vzít větší kusy a přistupovat k nim jako k primitivum tak, abychom je mohli kombinovat do větších celků a přitom se nestarat o jejich detail.**

Druhy abstrakce v softwarovém vývoji:

- Datové abstrakce
- Procedurální abstrakce (v imperativních jazycích)
- Funkcionální abstrakce (ve funkcionálních jazycích)
- Objektové abstrakce
- A další minoritní druhy abstrakcí
  - Manifest typing (vymezení typu proměnné)
  - Streamy
  - Doménově specifické jazyky (DSL) pro nové problémy
  - Jazyky pro hardware/registry
  - Schémata a jejich vyhodnocování
  - Programové manipulace: kompilace

Evolution of the Number of Lines of Code in Avionics Software



# Abstrakce - jmenné abstrakce

**Jména** a **jmenné prostory** jsou nejzákladnějším druhem abstrakce. Umožňují odkazovat se na proměnné, konstanty, operace, typy, funkce, moduly atd.. Používají se v ostatních typech abstrakcí.

Trochu speciální koncepci využívá framework *SpringData*, kde se ze jména metody dogeneruje kód pro přístup k datům. Funguje tak, že v názvu vyhledává klíčová slova `find...By`, `read...By`, `query...By`, `count...By`, `get...By` ty propojuje pomocí `And`, `Or`, a propojuje s názvy atributů objektů.

```
public interface PersonRepository extends Repository<User, Long> {  
    List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);  
    // Enables the distinct flag for the query  
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);  
    // Enabling ignoring case for an individual property  
    List<Person> findByLastnameIgnoreCase(String lastname);  
    // Enabling static ORDER BY for a query  
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);  
    List<Person> findByLastnameOrderByFirstnameDesc(String lastname);  
}
```

# Abstrakce - datová abstrakce

Odděluje abstraktní vlastnosti datového typu od jeho implementace. Abstraktní vlastnosti jsou ty, které jsou viditelné a měly bych je brát v potaz v kódu ve kterém abstraktní datový typ používám, zatímco konkrétní implementace je schovaná a mohu ji bez dopadu na tento kód měnit.

Hlavním reprezentantem datové abstrakce je **Abstraktní Datový Typ (ADT)**. ADT je matematický model pro datový typ definovaný množinou hodnot a operací nad těmito hodnotami, které splňují definované axiomy. Ekvivalentní k algebraické struktuře v abstraktní algebře.

*Příklad:* Typ Integer je ADT definovaný hodnotami ..., -2, -1, 0, 1, 2, ... a operacemi +, -, /, <, >, = které splňují axiomy asociativity, komutativity atd. V programovacích jazycích např. typ *boolean*, *float* atd. reprezentuje tzv. ADT.

*Další příklady:*

- V objektově orientovaném programování jsou třídy reprezentantem datové abstrakce - zapouzdřují přístup k datům (encapsulation) a poskytují pouze část dat (information hiding)
- Typy z collection API jako *Collection*, *List*, *Set*, *Map* jsou příkladem datové abstrakce - předepisují práci s datovou strukturou pomocí API a definuje princip práce s daty.
- RDBMS používají abstrakci tabulky, které má záznamy (*Row*) a sloupcečky (*Column*). Uživatel je odstíněn

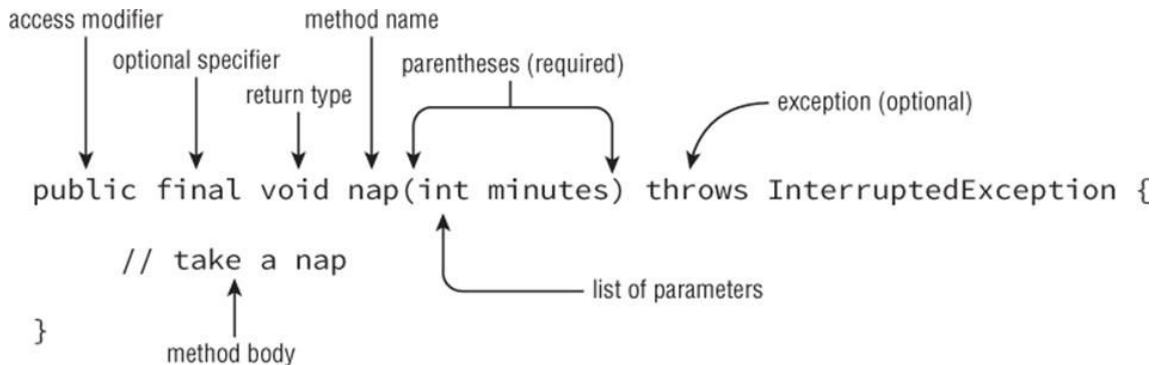
# Abstrakce - procedurální abstrakce (Control abstraction)

Cílem je, abych mohl vzít komponentu (část funkcionality svého systému) a bez zásahu do komponenty ji přepoužít na jiném místě v systému. K tomu mi stačí znát pouze rozhraní komponentu a funkcionalitu, kterou realizuje.

Tento mechanismus realizují pomocí tzv. subrutin. Tento mechanismus má následující vlastnosti:

- Izolace použití subrutiny od její implementace.
- Redukce množství duplicit v kódu a tzv. boilerplate kódu
- Možnost kombinování a zanořování

Příklad subrutiny v Java:



# Abstrakce - procedurální abstrakce (Control abstraction)

V čem se liší v různých implementacích:

- Syntax, typová kontrola
- **Mechanismus předávání parametrů do a z subrutiny**
- Statická či dynamická alokace a scope lokálních proměnných
- Overloading
- Generika
- Separátní kompilace

# Abstrakce - procedurální abstrakce (Control abstraction)

## Mechanismus předávání parametrů do a z subrutiny

### Call-by-Value

```
int a = 3;
void foo (int x) {
    //a and x have same value
    //changes to a or x don't
    //affect each other
}
//argument can be an expression
foo (a+a);
//no modifications to a
```

### Call-by-Reference

```
int a = 3;
void foo (int x) {
    //a and x reference same location
    //changes to a and x affect each other
}
//argument can be an expression
foo (a);
//a might be modified
```

### Call-by-Result

```
int a = 3;
void foo (int x) {
    //x is not initialized
    //changes to a or x don't
    //affect each other
}
//argument must be variable
foo (a);
//a will be modified by x upon
method call
```

### Call-by-Value-Result

```
int a = 3;
void foo (int x) {
    //a and x have same value
    //changes to a or x don't
    //affect each other
}
//argument must be variable
foo (a);
//a might be modified
```

### Call-by-Name

```
int a = 3;
void foo (int x) {
    //x is a function
    //to get value of argument
    //evaluate x() when value needed
}
//argument can be an expression
foo (a + a);
//no modifications to a
```



# Funkcionální abstrakce - funkce první třídy

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.*

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.

# Funkcionální abstrakce - funkce první třídy

Přepis pomocí funkce první třídy (Java 1.8+):

```
import java.util.function.Predicate;
public class FirstClassFunctionExample {

    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 boolean ageFilter(Person p){
        return p.age > 65;
    }
}
```

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

Funkce je pak volána následovně:

```
filterPerson(personList, FirstClassFunctionExample::ageFilter);
```

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

# Funkcionální abstrakce - funkce vyššího řádu

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

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

```
/* Scala: function compute má dva parametry - funkci f a hodnotu v. V těle metody je aplikace  
funkce f na hodnotu v*/  
def compute(f: Int => String, v: Int) = f(v)
```

# Funkcionální abstrakce - funkce vyššího řádu

```
/* Java 1.8+ : function compute vezme funkci f a hodnotu v a aplikuje funkci f na hodnotu v*/  
public static String compute(Function<Integer, Integer> f, Integer v) {  
    return f.apply(v);  
}
```

Funkci pak použij takto

```
/* Java 1.8+: function apply vezme funkci f a hodnotu v a aplikuje funkci v na hodnotu v*/  
  
public class AwesomeClass {  
    private static Integer invert(Integer value) {  
        return -value;  
    }  
    public static Integer invertTheNumber(){  
        Integer toInvert = 5;  
        Function<Integer, Integer> invertFunction = AwesomeClass::invert;  
        return compute(invertFunction, toInvert);  
    }  
}
```

# Funkcionální abstrakce - 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.
- Lambda expression v programovacích jazycích je funkce, kterou je možné definovat a volat bez bindingu s identifikátorem
- Lambda expression v Lambda calculu je formální systém matematické logiky a informatiky pro vyjádření výpočtu pomocí bindingu proměných a jejich substituce

# Funkcionální abstrakce - closures

**Closures** jsou lambda expressions jejichž volné proměnné (open bindings) jsou svázané s lexikálním prostředím ve kterém jsou použity což vede k vázanému výrazu (*closed expression*)

Jinak řečeno, jestliže ve funkci používám volnou proměnou, tak se podívám, jestli je definovaná ve stejném scope jako moje funkce a pokud ano, tak ji s touto definicí svážu.

*Pozor: Ve funkcionálních jazycích by výsledek funkce měl záviset pouze na jejích parametrech. Closures toto pravidlo porušují.*

*Pozn.: Java prozatím closures pořádně nepodporuje*

```
x = 0
def f(y):
    return x + y

def g(z):
    x = 1 # local x shadows global x
    return f(z)

g(1) # evaluates to 1, not 2
```

# Funkcionální abstrakce - currying

Currying spočívá ve vyhodnocování argumentů funkce per partes, kdy po každém kroku získám 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  $f$

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

currying podruhé pro 4 nám dá:

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

# Procedurální abstrakce - currying

Příklad vytvoření složené funkce při deklaraci:

```
/*Java 1.8+*/  
public class Currying {  
    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>> carrier = a -> b -> adder.apply( a, b ) ;  
        // Call apply 4 to carrier (to get a function back)  
        Function<Integer,Integer> curried = carrier.apply( 4 ) ;  
        // Results  
        System.out.printf( "Curry : %d\n", curried.apply( 3 ) ) ; // ( 4 + 3 )  
    }  
}
```



# Procedurální abstrakce - 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() ;  
}
```