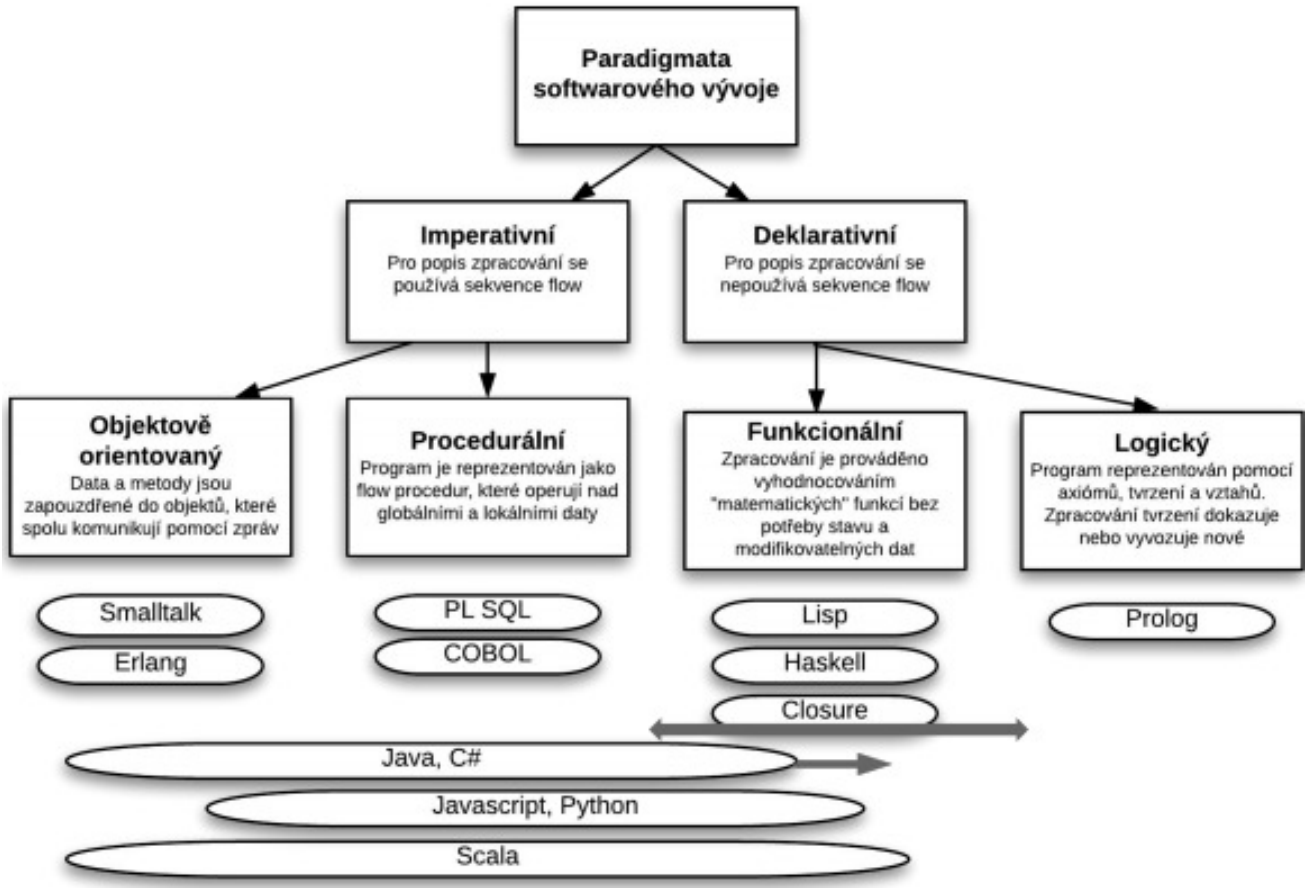
Klíčové koncepty modelování systémů I

Programovací paradigmata

- v posledních letech snaha o sjednocování funkcionálního, objektového i logického přístupu
- do OO jazyků se dostávají funkcionální koncepty



Deklarativní versus imperativní reprezentace

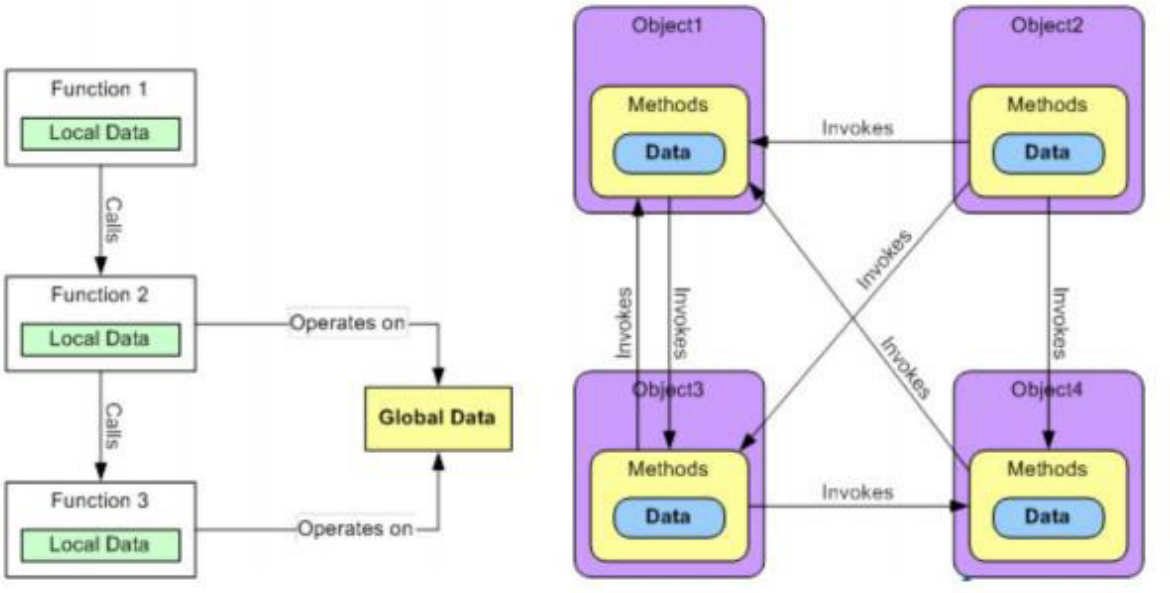
- **deklarativní = „what is“**
  - popisuje útvary a vztahy mezi nimi (např. výkřes v geometrii)
  - vyjádření odmocniny: $\sqrt{a} = y: y^2 = a; y >= 0$

- **imperativní = „how to“**
  - popisuje postup jako sekvenci činností (např. výměna oleje v autě)
  - vyjádření odmociny: 1) odhadnout výsledek G, 2) zlepšit odhad…

Procedurální vs. Objektové programování

- **procedurální**: základní stavební kámen = procedury
  - pracují nad lokálními/globálními daty; data organizována do záznamů (records)

- **objektové**: základní stavební kámen = objekty
    - zapouzdřují (a kontrolují) volání metod a práci s daty
    - komunikace mezi sebou pomocí zpráv



Procedurální (obecně imperativní) vs. funkcionální přístup

- ve funkcionálním jsou ZSK funkce, pracuje se s nimi 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 procedur
- Horší škálování a multithreading

Funkcionální 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

7

Logické programování

- vytvoření logického popisu problému → logicky odvoditelné řešení
- logický program = deklarativní zápis posloupnosti příkazů (logických vět), vyjadřujících:
    - pravidla (podmíněná)
    - fakta (nepodmíněná)
    - dotazy (cílové klauzule)
- programátor popíše problém, zadá otázky a stroj (problém solver) nalezne odpovědi

Úloha:

Všichni studenti jsou mladší než Petrova matka. Karel a Mirka jsou studenti.

Kdo je mladší než Petrova matka?

Zápis v PL1:

$\forall x\ [St(x) \supset Ml(x, f(a))]$

$St(b)$

$St(c)$

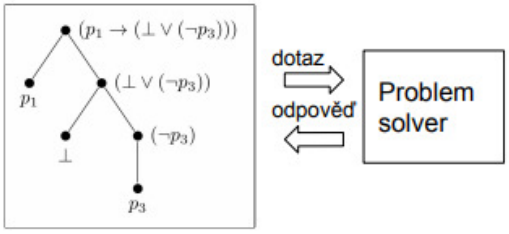
$\Rightarrow \exists y\ Ml(y, f(a))$???

Zápis v Prologu:

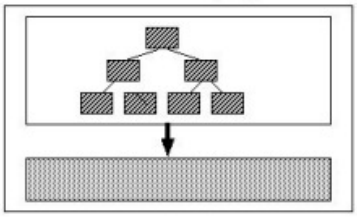
```
mladsi(X, matka(petr)):- student(X).   pravidlo
student(karel).                        fakt
student(mirka).                        fakt
?- mladsi(Y, matka(petr)).             dotaz
```

Shrnutí

Přístup logického programování



Procedurální přístup

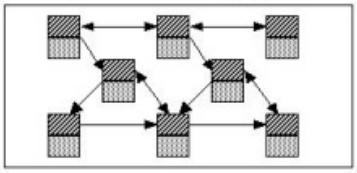


Exekuce zahrnuje provádění kódu, který operuje nad daty

Code

Data

Objektový přístup



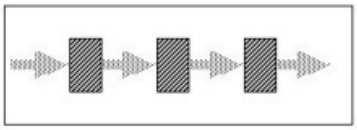
Objekt zapouzdřuje kód i data

Code

Data

Výpočet zahrnuje interakci mezi objekty

Funkcionální přístup



Data neexistují nezávisle

Code (Functions)

Exekuce zahrnuje zřetězené volání funkcí

Práce s komplexitou

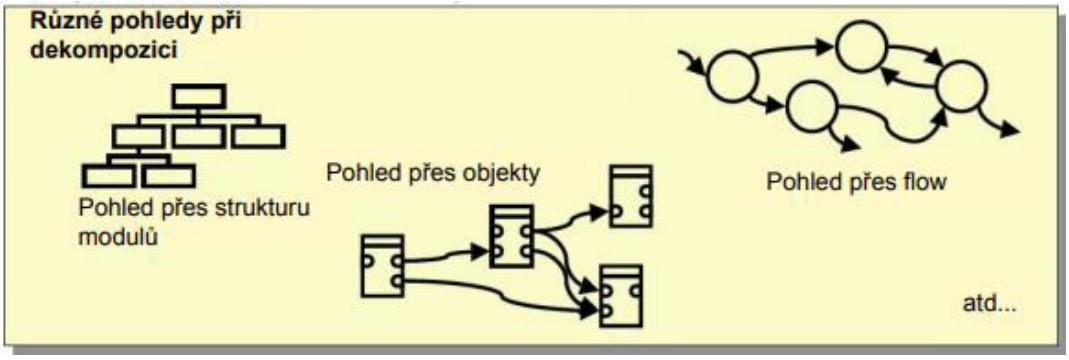
- Co dělat s problémem, se kterým si nemohu poradit? → snažit se jej strukturovat
- Jak mohu problém strukturovat? → abstrakce, dekompozice, hierarchie
- pro řešení problémů je vhodné použít **design patterny**
    - = obecné řešení problému, implementace má v různých jazycích/doménách
    - nejsou knihovnou, vložitelnou částí kódu…
    - jsou popisem či šablonou

- **dekompozice** = rozložení systému na menší komponenty
- **abstrakce** = skrytí komplexity komponent do jednodušších
- **hierarchie** = provázání komponent mezi sebou

Dekompozice

- „**Rozděl a panuj**“ - přístup k větším problémům , aby:
  - Každý podproblém měl přibližně stejnou úroveň detailu.
  - Každý podproblém byl řesitelný samostatně.
  - Řešení podproblémů lze zkombinovat tak, abych tím vyřešil celý problém.

- systém zpravidla představuje n-rozměrný problém
  - nelze jej popsat jedním pohledem → několik pohledů
  - dekompozici mohu provést pro tyto různé pohledy
  - např. UML definuje několik diagramů pro různé pohledy na systém



Jak dekomponovat?

- aplikací následujících principů…

- **Cohesion (princip soudržnosti)**
  - spojení funkcionality dle podobnosti
  - atributy podobnosti volím intuitivně (dle způsobu/účelu systému)

- **Coupling (provázanost)**
  - snaha o velmi kohezní (soudržné) moduly
  - pevné vazby uvnitř, co nejvolnější vazby mezi sebou

- **Reusability (přepoužitelnost)**
  - možnost použít v různých kontextech
  - volání z různých modulů, v různých fázích lifecyklu aplikace,…

- **DRY (Don’t Repeat Yourself)**
  - stejný kód (kus funkcionality) se nenachází na vícero místech

- **Flexibility isolation**
  - musím-li implementovat změnu, její dopad bude omezen na vazby mezi moduly či minimum modulů

- **Encapsulation (zapouzdření)**
  - data modulů jsou privátní
  - k modulům přistupuji pomocí povolených operací

Kritéria kvality softwarového vývoje

- nejdůležitějším je schopnost reagovat na změny
  - nejvíce bugů způsobují změny v kódu
  - chápání aplikace se mění
  - zasahování vícera vývojářů – neznají všechny části kódu
  - zákazník/analytik nikdy nedá kompetní, finální či 100% konzistentní požadavky
  - systém se bude rozšiřovat, včetně okolí, SW a HW komponent

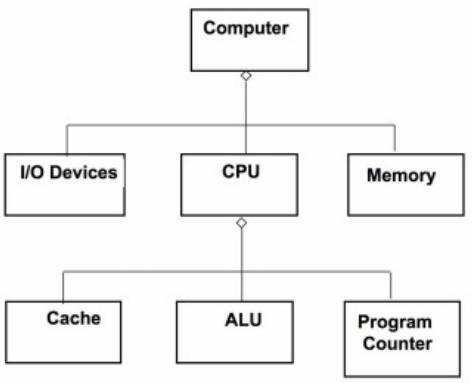
- **flexibilita** = množství/složitost nutných změn, aby systém fungoval i pro jiné scénáře
- **přepoužitelnost** = použitelnost systému v různých aplikacích
- **rozšiřitelnost** = schopnost adaptovat se na změny ve specifikaci (rozšiřování funkcionality)
- **robustnost** = schopnost reagovat na nepředpokládané situace
- **kompatibilita** = jednoduchost kombinování SW komponent mezi sebou
- **použitelnost** = jednoduchost použití systému uživatelem či jiným systémem
- **efektivnost** = minimalizace požadavků na zdroje (HW, lidské, finanční,…)
- **škálování** = schopnost fungování při narůstající zátěži
- **portovatelnost** = náročnost přenesení SW do jiného SW/HW prostředí

- **multikriteriální optimalizační problém**
  - nelze najít optimální řešení, pouze suboptimální
  - kritéria jsou ve vzájemné kontradikci
  - zvyšování flexibility → zvyšování komplexity → větší náklady a čas
  - zvyšování přepoužitelnosti → snižování použitelnosti (kvůli vyšší obecnosti)
  - vyšši granularita → vyšší použitelnost, ale nižší přepoužitelnost
  - nutno upravovat oblasti, kde minimální usilí = maximální efekt (The 80-20 rule)
  - zastavit ve chvíli, kde je další zlepšení příliš drahé
  - **paretovské zlepšení** – alokace může být PZ, existuje-li jiná alokace, kde na tom může být hráč lépe, aniž by si ostatní pohoršili
  - **paretovsky optimální** – nemožnost paretovského zlepšení

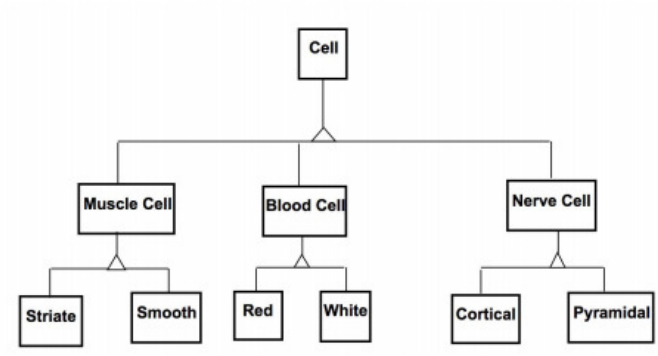
Hierarchie

- je určena vazbami mezi komponentami systému

***Part of* hierachie**



***Is kind of* hierachie:**



Abstrakce

- jednodušší reprezentace systému
- nahlížím na něj, aniž bych znal jeho detaily
- potřeba vzít větší kusy a přistupovat k nim jako k primitivům
    - ty pak zkombinujeme do celků, aniž bychom se starali o detail
- různé druhy abstrakce:
    - jmenné
    - datové
    - procedurální (v imperativních jazycích)
    - funkcionální (ve funkcionálních jazycích)
    - objektové
    - …

Jmenné abstrakce

- jména, jmenné prostory = nejzákladnější druh abstrakce
- možnost odkazování na proměnné, konstanty, operace, typy, funkce,…
- použití v ostatních typech abstrakcí
- příklad – generace Repository interface u Spring Data frameworku
    - klíčová slova find…By, read…By, count…By, apod. → propojení pomocí And/Or
    - propojení s názvy atributů a objektů (findByEmailAddressAndName(email,name))

Datová abstrakce

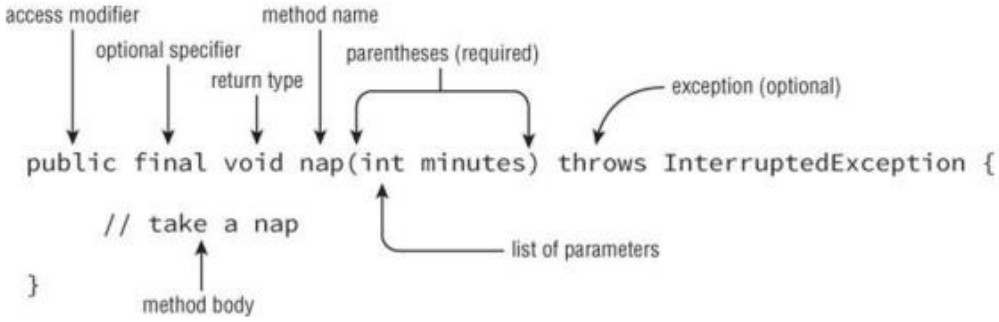
- oddělení abstraktní vlasntosti datového typu od jeho implementace
- abstraktní vlasnosti – viditelné; měl bych je brát v potaz v kódu, ve kterém ADT používám
- konkrétní implementace je schovaná, mohu ji měnit bez dopadu na tento kód

- **Abstraktní datový typ (ADT)**
    - = matematický model pro datový typ
    - definovaný množinou hodnot a operací nad hodnotami
    - operace splňují určité axiomy
    - ekvivalentní k algebraické struktuře v abstaktní algebře
    - příklady:
        - Integer je ADT definovaný hodnotami (…,-2,-1,0,1,2,…) a operacemi +,-,/,<,>,=, které splňují axiomy asociativity, komutativity apod.
        - Collection, List, Set, Map… z collection API
        - u RDBMS tabulky se záznamy (row) a sloupečky (column)

Procedurální abstrakce (Control abstraction)

- cílem je možnost vzít komponentu (část funkcionality systému) a bez zásahu do komponenty jí přepoužít na jiném místě v systému
- stačí k tomu znát rozhraní komponenty a funkcionalitu, kterou realizuje
- realizace pomocí **subrutin**
    - izolace použití subrutiny od její implementace
    - redukci množství duplicit v kódu a boilerplate kódu
    - možnost kombinování a zanořování

Příklad subrutiny v Java:



- v různých implementacích se liší:
    - syntaxí, typovou kontrolou
    - mechanismem předávání parametrů z/do rutiny
    - statická/dynamická alokace a scope lokálních proměnných
    - overloading
    - generika

Mechanismus předávání parametrů do a z subrutiny

Call-by-Value (*a* se kopíruje do *x*)

```
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 (*x* se nastaví na stejné místo v paměti jako *a*)

```
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 (hodnota *x* se inicializuje uvnitř a na konci x => a)

```
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-Result (a => x a na konci x => a)

```
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 (x se nastaví na funkci)

```
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

- objekt první třídy (first-class citizen) v PJ je entita, podporující následující operace:
  - být předána jako parametr
  - být přiřazena proměnné
  - být vrácena z funkce
- funkce první třídy splňuje výše uvedené vlastnosti
- metody a třídy nejsou hodnoty → 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.

Přepis pomocí funkce první třídy:

```
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;
   }
```

- nový parametr typu Predicate
  - obsahuje podmínku, kterou testujeme
  - dále metoda ageFilter, kterou vkládáme jako parametr p
- chceme-li filtrovat dle jiného atributu, uděláme jen drobnou změnu do implmentace filtru
- funkce volána následovně:

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

Funkcionální abstrakce – funkce vyššího řádu

- funkce splňující alespoň jednu z vlastností:
  - jedním či více parametry je funkce
  - vrací funkci jako parametr

```
/* 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 použiji 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

- = forma ve tvaru: (seznam argumentů funkce) -> tělo funkce
- používají se především k definování implementace funkčního rozhraní s jedinou metodou (inline formou) → redukce kódu + některé výhody funkc. pr. do Javy
- je to funkce, kterou je možné definovat a volat bez bindingu s identifikátorem

```
/* 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()));
```

Funkcionální abstrakce – currying

- vyhodnocování argumentů funkce per partes
- po každém kroku získám funkci, která má o jeden argument méně
- příklad:
  - pro funkci $f(x, y, z) = x * y + z$ aplikujeme argumenty 3,4,5: $f(3, 4, 5) = 3 * 4 + 5 = 17$
  - ale můžeme aplikovat jen „3“, získáme novou funkci f
    - $(3, y, z) = g(y, z) = 3 * y + z$, podruhé pro „4“: $g(4, z) = h(z) = 3 * 4 + z$

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>> 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 )
   }
```

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() ;
       }
}
```