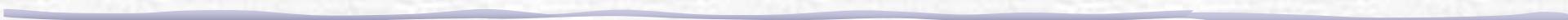




Principy objektově- orientovaného programování

Zbyněk Křivka

<http://www.fit.vut.cz/person/krivka>



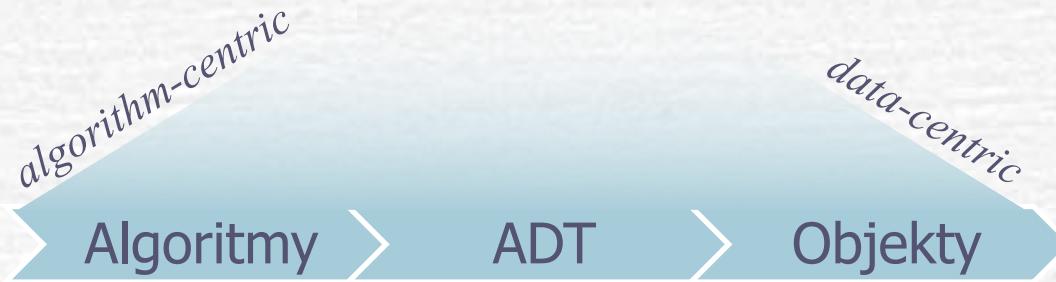
Cíle přednášky

- ☛ Prozkoumat základní vlastnosti OOJ
 - Klíčové pojmy (definice)
 - Klasifikace
 - Reprezentace, implementace
- ☛ Existence formálního modelu
- ☛ Seznámit se s různými modely OOJ
- ☛ Použití OOJ (okrajově)

Anglické termíny budou sázeny kurzívou.

Motivace

- Modelování = tvorba abstrakce reálného systému
- Proč modelovat?
- Abstrakce (*Black box*)
 - míra (volba variantní a invariantní části problému)
 - skrývání vnitřní struktury \Rightarrow pouze veřejný protokol
 - kvalita (uživatelská přívětivost, přesnost chování, implementace)

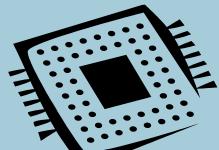


Vývoj abstrakcí v programování

START

Instrukce

- Provádění v CPU
- Předmět ISU



HALT

parametry

Rutiny

- Předávání toku řízení voláním
- Předmět IZP



výsledek

data
operace

Moduly

- Předávání toku řízení voláním
- Předmět IAL



protokol

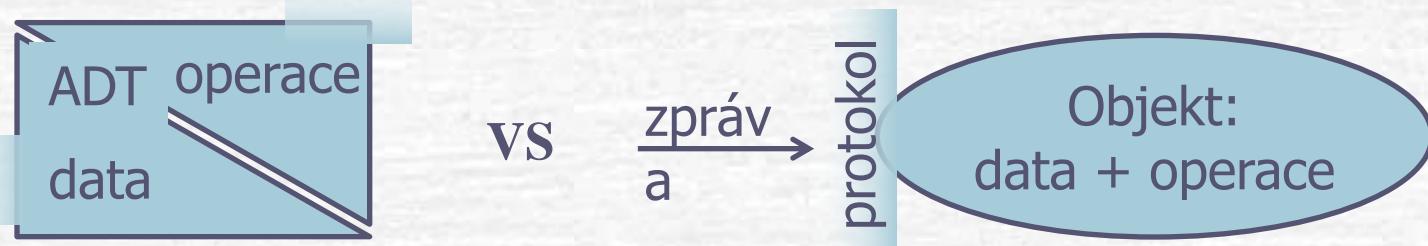
Objekty

- Zasílání zpráv
- Předměty IPP+IJA+ICP



Objektová-orientace (OO)

- Analýza, modelování, návrh, implementace



- OOP = OO programování, OO paradigm
 - OOJ = OO jazyk
 - Data = **atributy** (datové položky)
 - Operace = **metody** (členské funkce)
 - Objektový systém = kolekce komunikujících objektů
- } Položky
} (*fields, members*)

Případová studie (Ukázka třídy v PHP)

Car.php

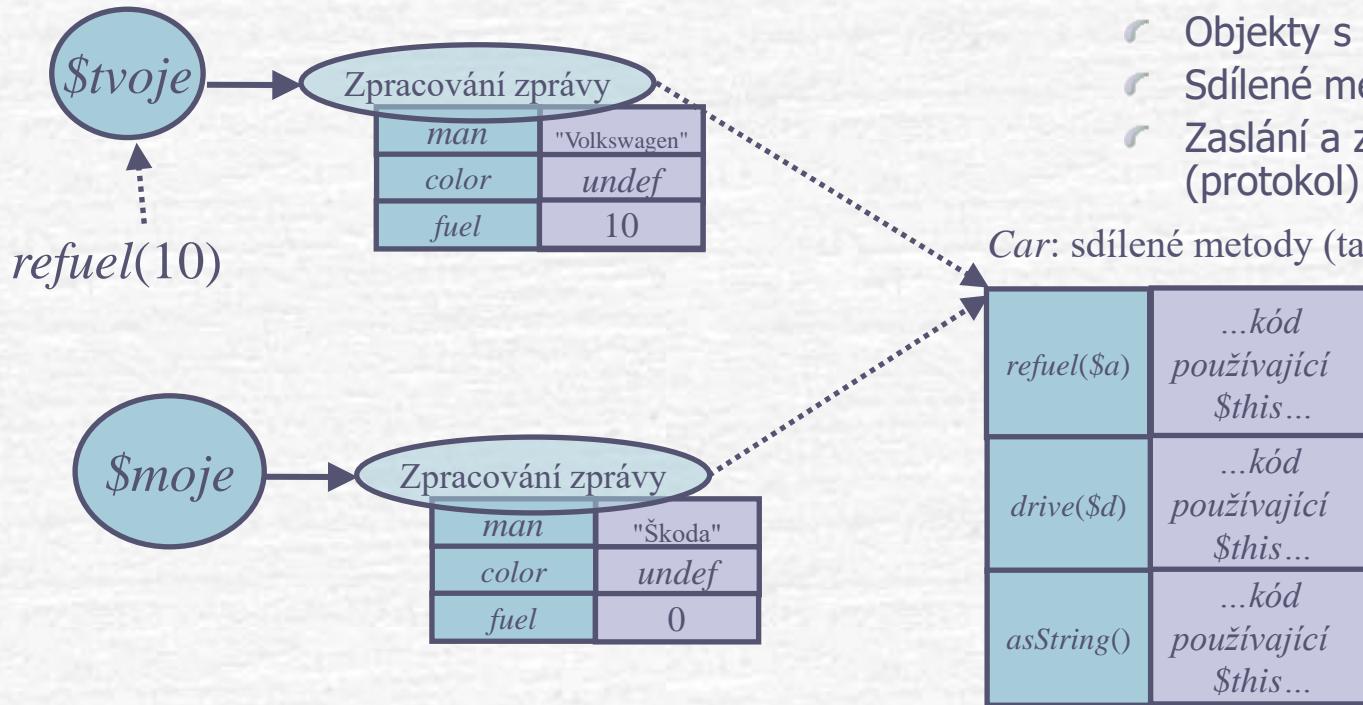
```
class Car {  
    var $man;  
    var $color;  
    var $fuel = 0;  
    function asString() {  
        return $this->man." in ". $this->color;  
    }  
    function refuel($amount) {  
        $this->fuel += $amount;  
        return $this;  
    }  
    function drive($dist) {  
        $this->fuel -= min($this->fuel,  
            $dist/100 * 5);  
        return $this;  
    }  
}
```

Car_demo.php

```
$moje = new Car();  
$moje->man = "Škoda";  
$tvoje = new Car();  
$tvoje->man = "Volkswagen";  
echo $tvoje->refuel(10 /*l*/)  
    ->drive(150 /*km*/)  
    ->fuel;
```

- Bez typové anotace
- Pojmy: třída, instanční atribut, instanční metoda, \$this
- Proměnná, instance, zpráva, invokace metody, kaskáda

Základní model výpočtu

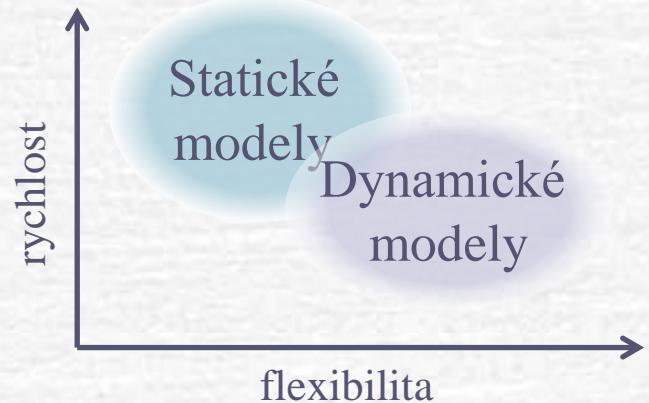


- Proměnné
- Objekty s atributy
- Sdílené metody
- Zaslání a zpracování zpráv (protokol)

`Car: sdílené metody (tabulka metod)`

Modely OOP

- ⌚ Model OOP/OOJ = pravidla pro tvorbu obj. systému
 - ⌚ konstrukce objektu, propojování objektů, model výpočtu
- ⌚ Statické modely (překládané, typované?)
 - „OO zdrojový text“, statické tabulky metod
- ⌚ Dynamické modely (VM, netypované?)
 - Expresivnější, reflektivita, ...
- ⌚ Každý OOJ má svůj model.
 - ⌚ C++, Java, C#, ...
 - ⌚ PHP 8, Python 3, Smalltalk, SELF, ...
- ⌚ Jak vypadá minimální model OOP?



Případová studie (Zjednodušení základního modelu)

Bicycle.fictional_PL

```
bike1 = object {  
    getSpeed() { return 0 };  
    speedUp(increment) {  
        update getSpeed = { return  
            getSpeed() + increment };  
        ...  
    }  
}
```

```
bike2 = object {  
    getSpeed() { return 0 };  
    speedUp(increment) {  
        update getSpeed = { return  
            getSpeed() + increment };  
        ...  
    }  
}
```

pokračování Bicycle

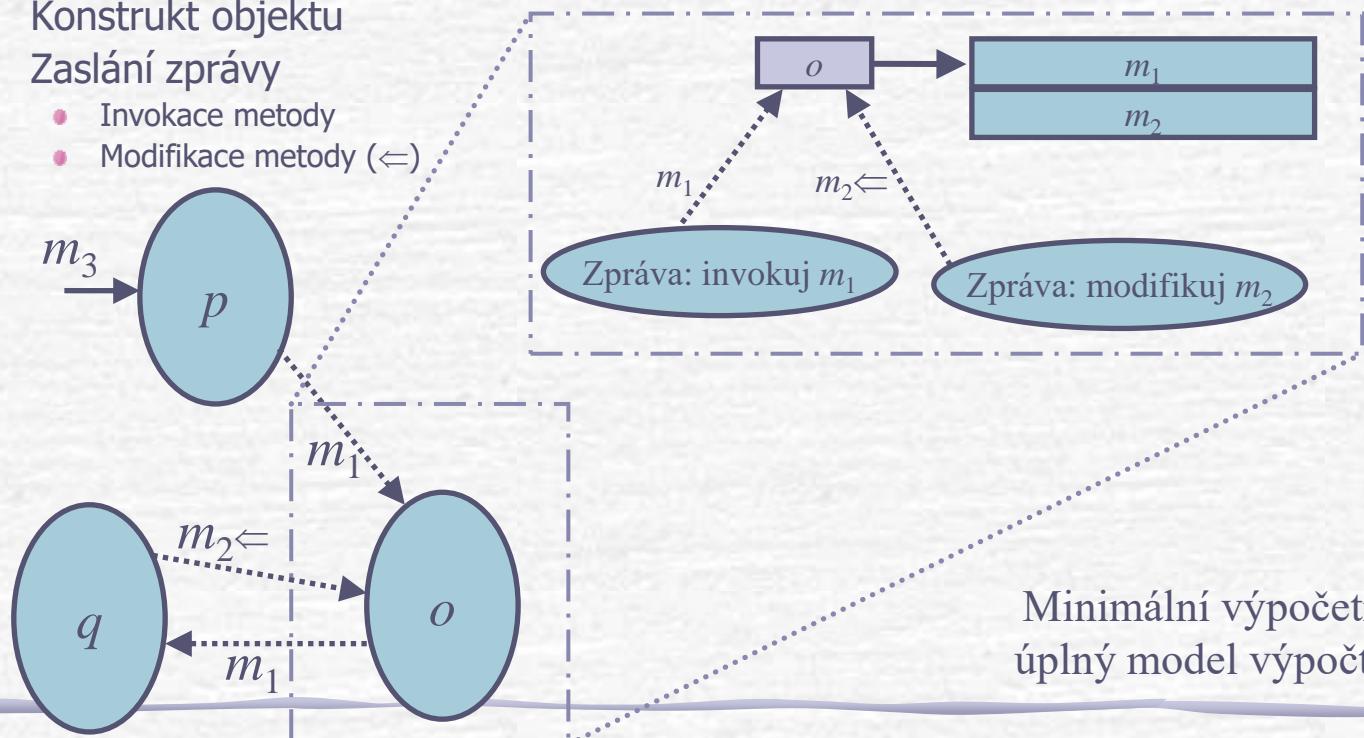
```
bike1.speedUp(10);  
bike1.printStates();  
  
bike2.speedUp(20);  
bike2.slowDown(5);  
bike2.printStates();
```

- ☞ Pro min. model NEpotřebujeme:
 - ☞ Třídy, dědičnost
 - ☞ Atributy (nahrazeny metodami a možností modifikovat kód metod)
 - ☞ Zapouzdření
 - ☞ Parametry metod (stačí „skrytý“)

Minimální model výpočtu

- Proměnná
- Konstrukt objektu
- Zaslání zprávy

- Invokace metody
- Modifikace metody (\Leftarrow)



Formální báze OOP

- 🕒 Matematický popis vlastností jazyků
- 🕒 Příklady
 - UML – OO analýza, OO návrh, OO modelování
 - ζ -kalkul – popisuje minimální OO model
 - OCaml – imperativní OOJ s formální bází

Popis syntaxe a sémantiky složitějších OOJ

✓ **Syntaxe:**

- Kombinace (E)BNF, gramatik, textu, příkladů

✓ **Sémantika:**

- Kombinace textu, příkladů, diagramů
- Zvýšení složitosti
 - pár nových klíčových slov \Rightarrow mnoho nových významových kombinací (např. function nebo static v PHP 8)
- Víceúrovňový popis (od jednoduššího ke složitějšímu)

OO Paradigma – Koncepty

- ⌚ Abstrakce ≈ Objekty
- ⌚ Zapouzdření (*Encapsulation*)
- ⌚ Mnohotvárnost (*Polymorphism*)
- ⌚ Dědičnost (*Inheritance*)

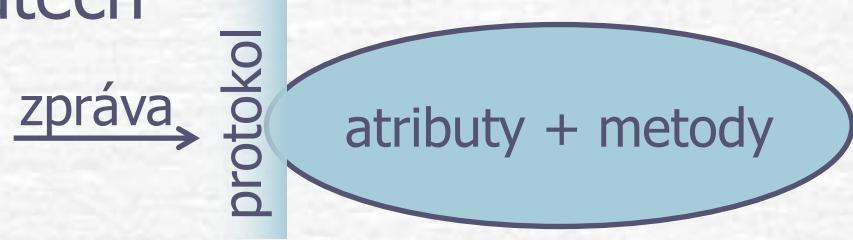


Generická řešení

Objekt

- ✓ Autonomní výpočetně úplná entita
- ✓ Identita nezávislá na atributech

- ✓ Datová povaha objektu:
 - proměnná obsahuje/odkazuje objekt
 - datový typ objektu (množina možných hodnot datových položek a množina operací nad nimi)



odesilatel =
sender

zpráva *name(argumenty)*

příjemce =
receiver

Zapouzdření

- ✓ Uzavřenost vůči okolním objektům (modifikátory viditelnosti)
- ✓ Přístup pouze přes **veřejný** protokol (*message lookup*):
 - Zaslání **zprávy** (*message*) = příjemce, selektor, argumenty
 - Protokol příjemce rozhodne o reakci na zprávu:
 - a) **Invokace** odpovídající metody (dle selektoru a argumentů) a navrácení výsledné hodnoty odesilateli
 - b) **Chyba** – nerozumí zprávě (*DoesNotUnderstand*)

```
receiver.name(arg1, arg2);  
receiver.name(arg1, arg2)  
$receiver->name($arg1, $arg2);  
receiver name: arg1 and: arg2.
```

Analogická syntaxe
s přístupem
do struktury

odesílatel = příjemce

Zapouzdření

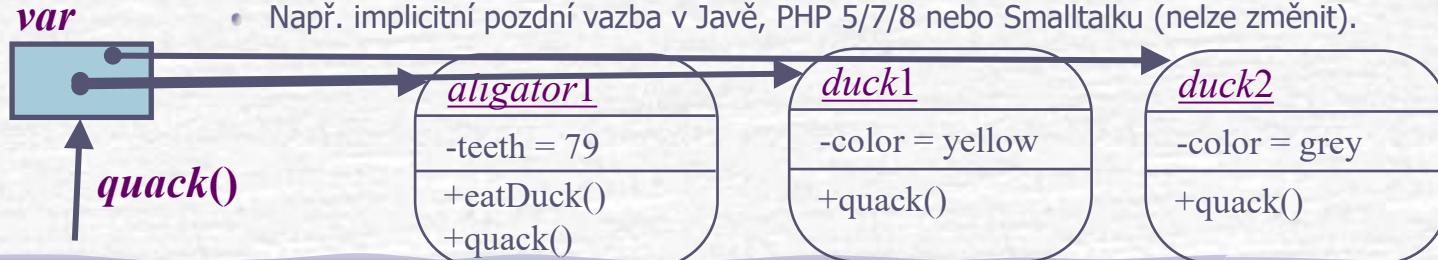
- Posílá-li objekt zprávu sám sobě, přistupuje přes **interní** protokol:
 - Zaslání zprávy
 - Protokol příjemce rozhodne o reakci na zprávu:
 - Přístup k atributu (čtení nebo modifikace)
 - Invokace odpovídající metody a návrat výsledku
 - Chyba – nerozumí zprávě (*DoesNotUnderstand*)

Identifikace sama
sebe jako příjemce
(*self*-parametr nebo
kontextový objekt)

```
→ this.attr = this.attr + 1;  
→ self.attr = self.attr + 1;  
$this->setAttr($this->getAttr()+1);  
    self attr: self attr + 1.
```

Mnohotvárnost/Polymorfismus

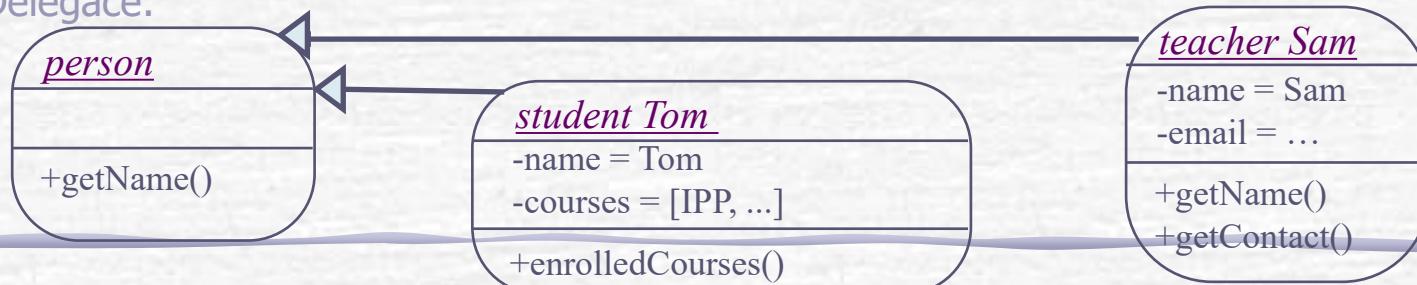
- Stejnou zprávu lze zaslat různým objektům!
 - Některé OOJ toto omezují typovým systémem.
- Protokol umožňuje **individuální reakci**, protože kvůli zapouzdření neznám implementaci invokované metody (a ta se může pro různé objekty lišit)
 - Statické (v době překladu) určení reakce = **brzká vazba**
 - Dynamické (za běhu) určení reakce = **pozdní vazba**
 - Některé OOJ umožňují druh vazby určit pro každou zprávu (jeden typ vazby bývá implicitní).
 - Např. V C++ implicitně brzká vazba; pozdní vazba jen virtuální metody.
 - Např. implicitní pozdní vazba v Javě, PHP 5/7/8 nebo Smalltalku (nelze změnit).



Dědičnost (obecně)

- Musíme v každém objektu implementovat stejné metody, aby rozuměli stejné množině zpráv? Ne, při použití dědičnosti!
- Dědičnost = sdílení společných položek (od předků) + individuální položky (v potomcích)
 - Primárně pro **sdílení/znovupoužití metod**
 - Specializace** zděděného objektu:
 - Přidání nových položek
 - Modifikace metod
 - Nezvyklé: modifikace atributů, odebrání/zakázání položek

Delegace:



Vytváření nových objektů

1. Vytvoření prázdného objektu + **úprava** položek
2. Kopie (**klonování**) + **úprava** položek
 - Prototyp = klonovaný objekt
3. Vytvořením pomocí „šablony“ + **naplnění** atributů
 - Nutnost definovat „šablonu“, tzv. **třída**
 - Obsah šablony?
 - Je šablonou objektem první kategorie?

Třída (*Class*)

1. šablona pro vytváření nových objektů tzv. **instancí**
2. **Třída** = Objekt/entita obsahující:
 - seznam atributů v instancích (vč. metadat)
 - implementace metod (sdílené instancemi)
 - PS: Je-li třída objekt první kategorie, komunikujeme s ní též zasíláním **třídních zpráv**.

Instanciacie = proces vytvoření objektu + volání konstruktoru

Instance = identita + reference na její třídu + data

Případová studie (PHP 8: instanciace/klonování)

ShopProduct.php

```
<?php // vynescháváno
class ShopProduct {
    private string $name;
    protected int $salary = 0;

    public function setTitle(string $t)
        : static {
        $this->title = $t;
        return $this;
    }

    public function getTitle() : string {
        return $this->title;
    }
}

?>
```

test_ShopProduct.php

```
<?php
require_once "ShopProduct.php";
// require versus autoload + namespaces
$p1 = new ShopProduct();
$p1->setTitle("Catch 22");
$p2 = clone $p1;
$p1->type = "book"; // deprecated

$p3 = $p1;
$p3->SetTitle("Star Wars");
print "P1: {$p1->getTitle()}\n";
if (isset($p2->type))
    print "P2: {$p2->type}\n"; // skipped
?>
```

MetaClassClass : MetaClass

třída

metatřída

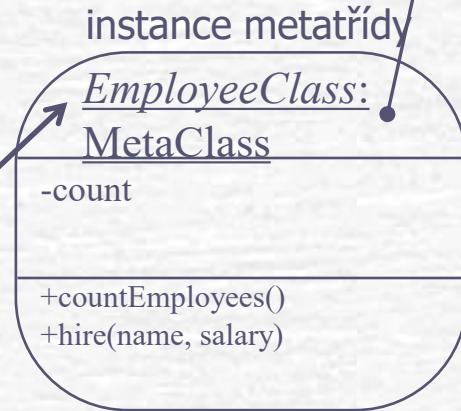
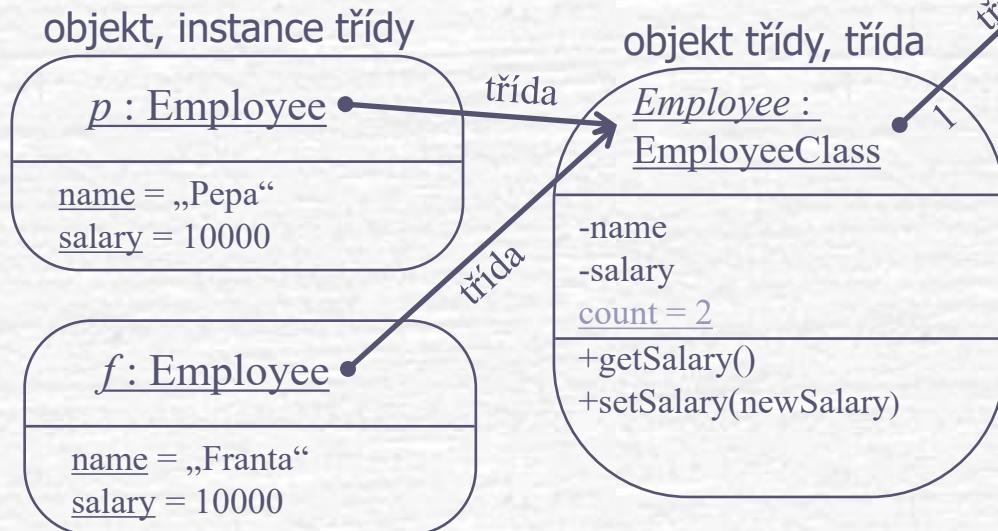
MetaClass : MetaClassClass

třída

třída

Čistý třídní model OOJ Smalltalk

- Každý objekt má svou třídu.
- Třída je také objekt.



Instanční položky
Třídní položky
Metatřída

Případová studie (PHP 8: třída a statické položky)

Employee_static.php

```
class Employee {  
    private string $name;  
    private int $salary = 0;  
    public function setSalary(int $s)  
        : static {...}  
  
    ...  
    // EmployeeClass part:  
    private static int $count = 0;  
    public static function hire(string $n, int $s) :  
    static {  
        static::$count++;  
        return new Employee()->setName($n)-  
>setSalary($s); }  
    public static function count() : int {  
        return static::$count; }  
}
```

Employee_static_demo.php

```
$p = new Employee();  
$p->setName("Pepa")->setSalary(10000);  
$f = Employee::hire("Franta", 10000);  
  
echo Employee::count(); // 1?
```

Ortogonalita kritérií klasifikace = nezávislost

Klasifikace OOJ 1/3

Dle přístupu k vytváření objektů:

- ✓ Beztřídní OOJ (*prototype-based, class-less*)
- ✓ Třídní OOJ (*class-based*)

Dle čistoty objektového modelu:

- ✓ Čisté (vše je objekt)
- ✓ Hybridní (míchání s jinými paradigmami, doplněny objekty)

Dle platformy pro běh OO programů:

- ✓ Překládané (efektivita běhu, více zdrojového textu)
- ✓ Interpretované (pomalé, velmi flexibilní)
- ✓ Částečně interpretované (mezikód, virtuální stroj)

Klasifikace OOJ 2/3

Dle údržby objektového prostředí:

Klasifikace dle údržby objektového systému		Iniciace výpočtu	
Uložení objektů	Zdrojový kód + knihovny	Výrazem	Speciální metodou/funkcí
	Zdrojový kód + knihovny	ç-kalkul, OCaml	Java, C++, PHP 5, Python 3, ECMAScript
	Obraz objektové paměti	Pharo/Squeak Smalltalk, SELF	-

Klasifikace OOJ 3/3

Dle dědičnosti (počet přímých předků):

- ◐ Jednoduchá
- ◐ Vícenásobná

Dle předmětu dědění:

- Dědičnost implementace
 - Třídní dědičnost
 - Nadřída (*superclass*), podřída (*subclass*)
 - Delegace
- Dědičnost rozhraní

Klasifikace nejen OOJ

Dle způsobu určení typů (opakování):

- Beztypové, Netypované, Typované

Dle důslednosti kontroly typů:

- Silně typované (*type-safe*)
- Slabě typované (implicitní konverze)

Dle doby kontroly typů:

- Statická kontrola typů (při překladu, před spuštěním)
- Dynamická kontrola typů (za běhu, *run-time*)

Čistý třídní dynamicky silně typovaný

Čistý beztřídní dynamicky silně typovaný

Historie OOJ

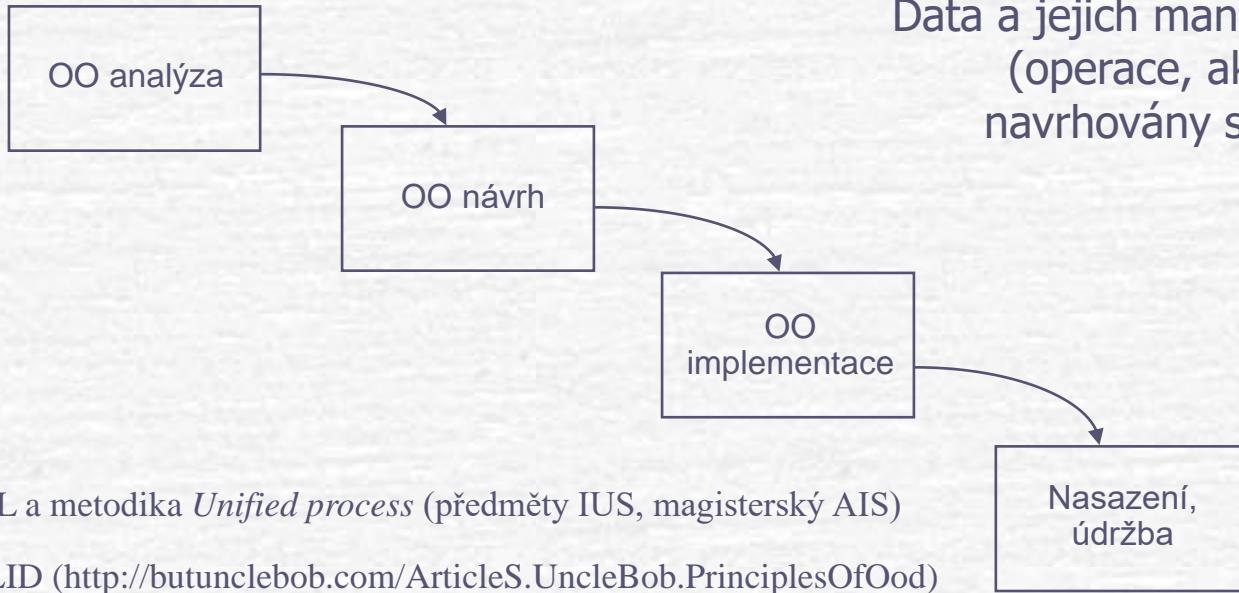
- ⌚ *Simula* (1967)
- ⌚ *Smalltalk* (1980), *CLOS* (1986-8), *Self* (1986)
- ⌚ *C++* (1983), [*Objective-C*] (1986)
- ⌚ *Object Pascal* (1986), *Delphi* (1995)
- ⌚ *FORTRAN 2003*
- ⌚ *Java* (1995), *C#* (2001)
- ⌚ *OCaml* (1996)
- ⌚ *ECMAScript (JavaScript)*, *Ruby*, *Python*, *PHP*, ...

Hybridní třídní staticky slabě typovaný

Čistý (bez)třídní dynamicky silně typovaný

Hybridní beztřídní dynamicky slabě typovaný

OO návrh (*Design*)



Principy OO návrhu

- 🕒 Identifikace objektů/tříd/instancí
- 🕒 Rozdělení zodpovědností mezi objekty
- 🕒 Různá kritéria dobrého návrhu
 - Trasovatelnost vlastností (*Traceability*)
 - Velké × Malé metody, *Ravioli code*
 - Robusnost a udržovatelnost (*Maintainability*)
 - Splnění (ne)funkčních požadavků
 - Kvalita služeb (QoS, *Quality of Service*)

UML

- ✓ Vizuální jazyk vhodný pro OO modelování (verze 2.5.1 v prosinci 2017)
- ✓ Podporuje formální verifikaci a validaci
- ✓ Strukturální × Behaviorální **pohled** (*View*)
- ✓ **Diagramy** (graf z entit a vztahů)
 - Tříd, komponent, nasazení
 - Objektů, sekvenční, aktivit, stavový, případů použití
- ✓ Většina prvků je nepovinných (prototypování)

Diagram tříd/objektů

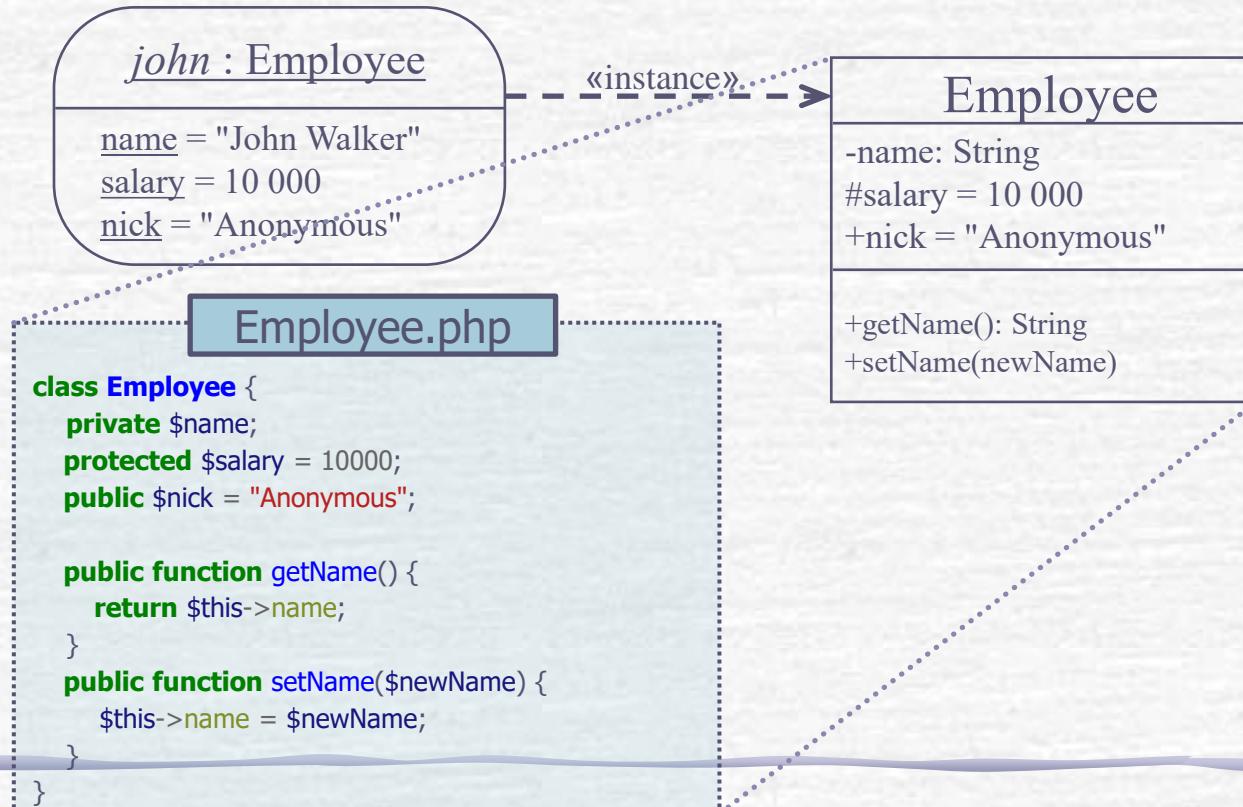
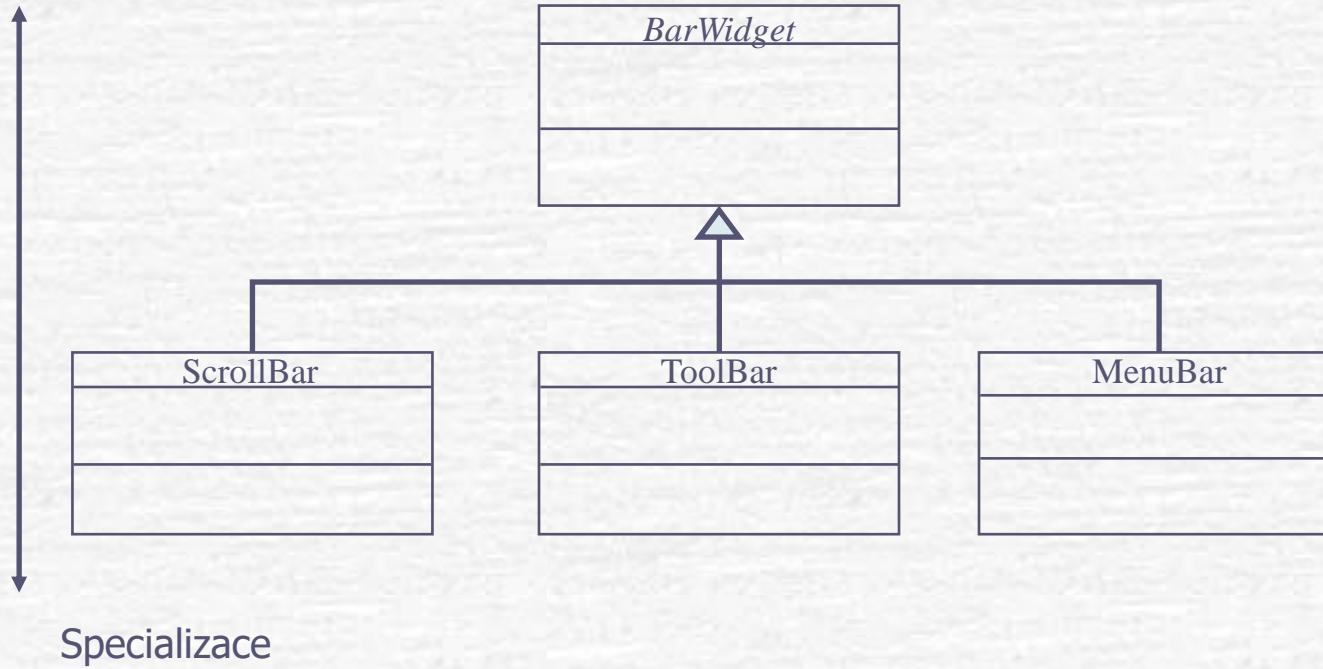


Diagram tříd: Dědičnost

Generalizace



Násobnost vztahů (Association)

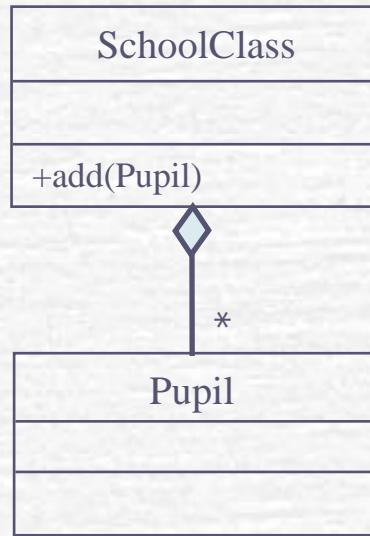
- ⌚ 1 zdrojové entity (ZE) k 1 cílové 
- ⌚ 1 ZE k několika cílovým (CE) 
- ⌚ 1 ZE volitelně k 1 CE (*optional*) 
- ⌚ 1 ZE k alespoň 1 CE (*required*) 
- ⌚ 1 ZE k n CE 

Druhy vztahů mezi entitami

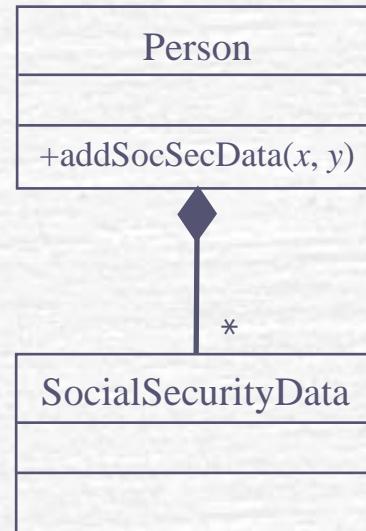
- ◉ Agregace (*Aggregation*) 
- ◉ Kompozice (*Containment*) 
- ◉ Závislost (*Dependency*; použití) 
- ◉ Realizace 

- ◉ Generalizace 
- ◉ Obecná asociace 
- ◉ Instanciace (od instance k třídě) 

Diagram tříd: Agregace/Kompozice

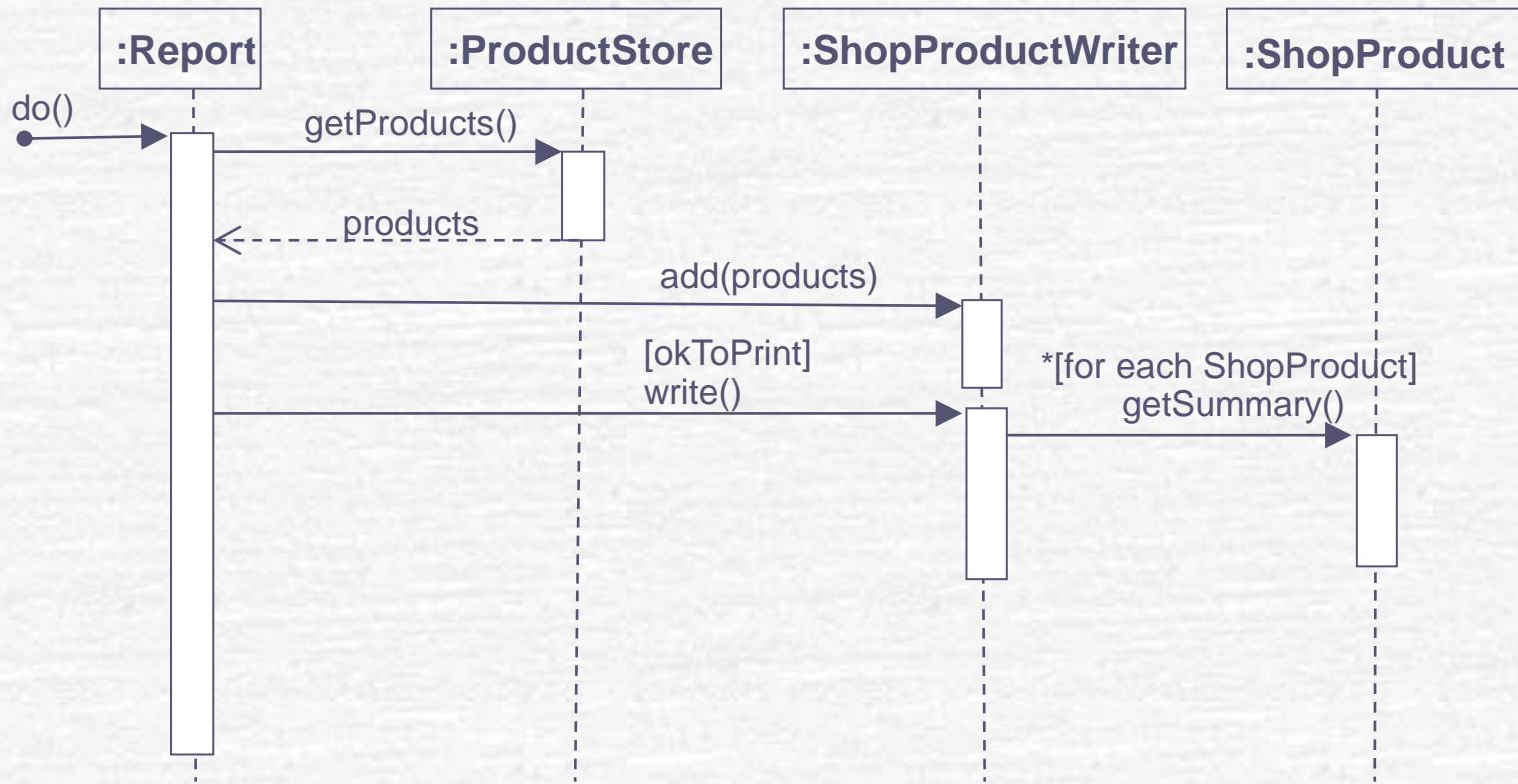


```
<?php
$p = new Pupil("Petr");
$c = new SchoolClass("1A");
$c->add($p);
```



```
<?php
$p = new Person("Jan");
$p->addSocSecData("VZP", 2021,
12345);
```

Sekvenční diagram



Formální návrh v UML

- ✓ Společný jazyk (zákazník, návrhář, programátor) pro validaci/verifikaci
- ✓ Výpočetně neúplný (hlavně pro komunikaci)
- ✓ Rozšiřitelný (Profily v UML 2.0)
 - Možný vstup pro generování kódu (MDD)
 - Profil *Executable UML*
 - Generování aplikačního kódu (např. EMF, GMF)
 - Profil s reálným časem (*real-time*), ...

Výhody/Nevýhody OOP I

“Jednoduché” na pochopení

- Skutečné objekty → Softwarové objekty (přímočaře)
- Podpora dekompozice: Rozdělení komplexity na abstrakci a zapouzdření

Praktické

- Nasazení ve skutečných aplikacích (*většinou* vhodné)

Zlepšení produktivity (hlavně velké aplikace)

- Znovupoužitelnost (*Reusability*)
- Propracované metodologie softwarového inženýrství

Stabilita

- Změny většinou lokální vůči objektu/třídě

Výhody/Nevýhody OOP II

- Delší učící křivka
- Praxe některé výhody nepotvrdila
 - Probíhá zkoumání možností komponentních technologií
- Generuje pomalejší kód
 - Režie na objekty, režie na polymorfismus

K zapamatování

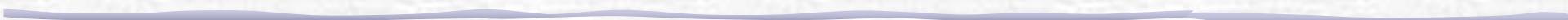
- ☛ OO paradigmá – koncepty, pojmy
- ☛ Formální modely OOP - UML
- ☛ Klasifikace OOJ

Cvičení

- 👉 Vyzkoušejte si syntaktický zápis definice třídy v jazycích Java, C++, C#, Python 3 a PHP 8.
- 👉 Zabývejte se rozdíly v OO modelech těchto jazyků.
 - Jak vytvořit, pokud to lze, statickou metodu, třídní metodu, instanční metodu?
 - Jak redefinovat metodu? Případně jak poznat polymorfní a nepolymorfní metody?



Třídní objektově-orientované jazyky a jejich implementace



Třída (*Class*)

Třída = Objekt/entita obsahující:

- seznam **instančních atributů** (vč. metadat)
- data třídních atributů
- implementace **instančních metod**
- reference na její třídu (např. Smalltalk, Python 3)
- statické (\neq třídní) položky (např. Java, C++, PHP 5/8)

Instance = identita + „reference na její třídu“ + data
instančních atributů

Extent A = kolekce všech instancí třídy A a podtříd, „třída“ jako množina

Kolekce instancí = množina libovolných instancí (i různých) tříd

Subclass, (přímý) potomek

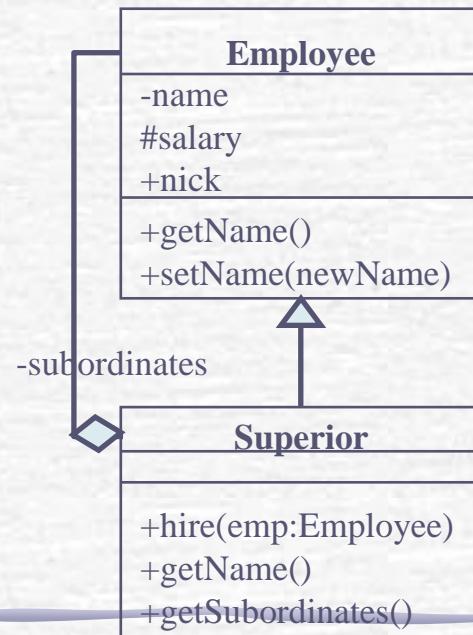
Superclass, (přímý) předek

Třídní dědičnost

- Účel: sdílení/znovupoužití položek skrz třídy
- Podtřída = odvozená třída popisující změny oproti přímé nadtřídě
 - Replikace a přidání nových položek
 - Redefinice zděděných metod (*overriding*)

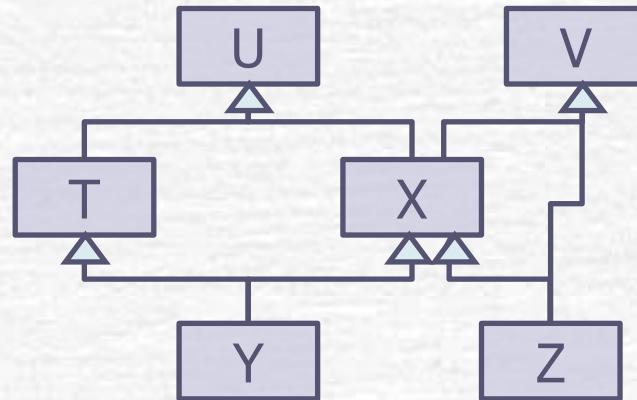
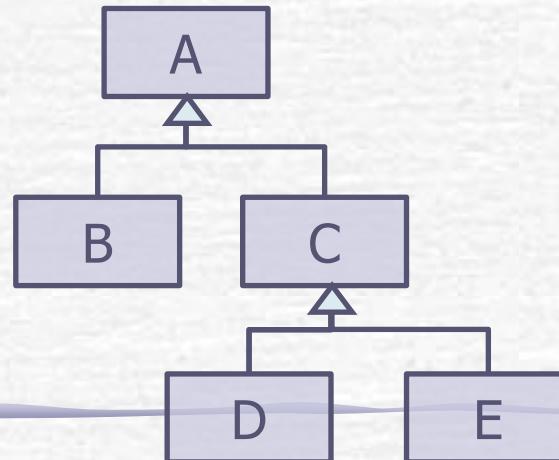
Superior.php

```
class Superior extends Employee {
    private $subordinates = array();
    public function hire(Employee $emp) {
        $this->subordinates[] = $emp;
    }
    public function getName() { // overridden
        return "manager ".parent::getName();
    }
    function getSubordinates() {
        return $this->subordinates;
    }
}
```



Hierarchie třídní dědičnosti

- Relace dědičnosti „*is-a*“ = částečné uspořádání na množině tříd
- Hierarchie třídní dědičnosti = graf relace dědičnosti



Definice/Deklarace

- ☞ Objekt nebo třída musí být popsán(a) před prvním použitím
 - Beztřídní OOJ: Objekt je naklonován před jeho změnou
 - Třída je definována před první instanciací
- ☞ Problém: Vzájemné vazby mezi objekty/třídami (skrz reference/ukazatele)
 - Možné řešení: Vazba přes jméno skrz deklaraci (tj. „zarezervování“ jména, definice bude následovat později)
- ☞ **Jmenný prostor (namespace)** = mapování jmen na objekty jazyka (instance, třídy, atributy, metody, ...)
- ☞ Rozsah platnosti jmenného prostoru (kde je dostupný),
 - Úrovně zanoření (nejvnitřnější=lokální → třídní → globální → vestavěná)
 - Např. třída versus vnitřní/zanořená třída (*Nested/nesting class*)

Příklad dopředné deklarace třídy v C++

```
class Bar; // forward declaration of a class
class Foo {
    public: Bar * pt;
            Bar & passBar(Bar & bar) {
                // cannot use bar's members
            };
};
class Bar : Foo {
    public: Foo f;
            void m(Bar b) {
                b.passBar(b);
            };
};

```

Operace nad instancemi

Objekt jako „chytrá“ datová struktura:

- ☞ Přístup k atributům (pořadí/umístění zná třída)
- ☞ Zaslání zprávy instanci (jednotný protokol)
 - Získání třídy, hledání metody ve třídě a nadtřídách
 - Potřebuji hledat v nadtřídě? ⇒ *super, base, parent::*, jméno konkrétní nadtřídy
 - Hledání metody (*dispatching*): staticky nebo dynamicky
 - Invokace metody ((skrytý) *self*-parametr)
 - Metoda jako „chytrá“ funkce (viz dále)
 - Návrat hodnoty výsledku odesilateli
- ☞ Modifikátory přístupu (konfigurace zapouzdření)
 - *private, protected, internal, public, ...*

Operace nad třídami

Třída jako objekt první kategorie:

- ☞ Zaslání třídní zprávy objektu třídy
 - Třída je objekt = analogický postup
 - *self*-parametr třídní metody je objekt třídy
 - Např. Python umožňuje instanci, pak dodatečně ještě zjistí třídu
- ☞ Je vlastní třída podtřídou dané třídy (`instanceOf`)? Různé metody prohledávání hierarchie.

Třída jako speciální entita jazyka:

- ☞ statické atributy (typicky deklarace i definice ve třídě)
- ☞ statické metody (bez *self*-parametru; např. Java, C++, PHP 5)
 - Jako funkce ve jmenném prostoru třídy, možnost redefinice
 - Např. Python 3 podporuje třídní i statické metody.

Případová studie: Python (I)

```
class Circle(object):  
  
    pi = 3.14159  
    __secret = 'password'  
  
    def __init__(self, radius = 1):  
        self.radius = radius  
  
    @staticmethod  
    def square(number):  
        return number*number  
  
    def area(self):  
        square = self.__class__.square(self.radius)  
        return square * self.__class__.pi  
  
    @classmethod  
    def getClass(cls):  
        return cls.__name__
```

""""příklad definice třídy""""

""""třídní proměnná""""

""""(privátní) třídní proměnná""""

""""inicializátor po konstrukci + param.""""

""""instanční proměnná""""

""""funkční dekorátor""""

""""statická metoda, bez self-param.""""

""""stále možnost redefinovat""""

""""instanční metoda, *self*""""

""""lokální proměnná""""

""""přístup k třídní proměnné""""

""""funkční dekorátor""""

""""třídní metoda, *cls* je třída""""

""""třída má atribut se jménem""""

Případová studie: Python (II)

```
>>> c = Circle(10)      """instanciace včetně invokace konstruktoru"""

>>> c.area()           """zaslání zprávy ⇒ invokace instanční metody"""

>>> Circle.pi          """přístup k třídní proměnné"""

>>> Circle.getClass()    """zaslání třídní zprávy ⇒ invokace třídní metody"""

>>> Circle.__secret     """nepovolený přístup k privátní třídní proměnné (mangling)"""

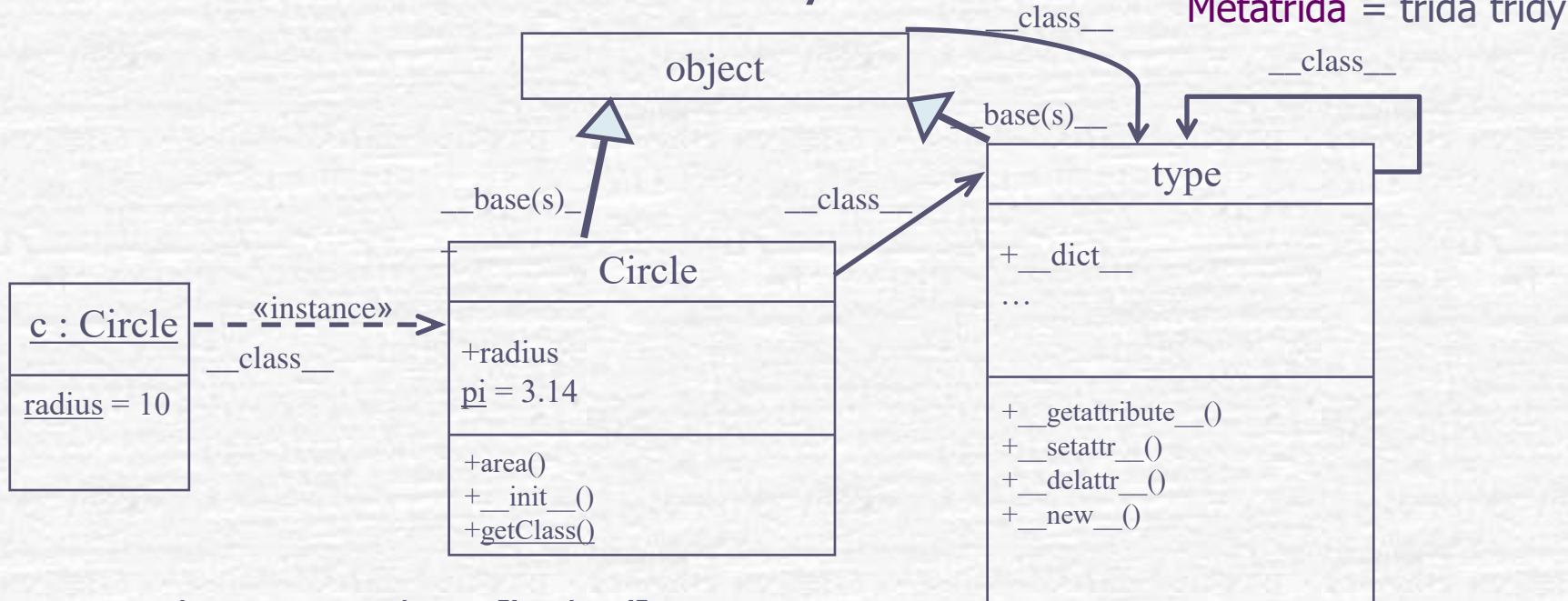
>>> c.__class__         """získání třídy instance, třída je také objekt"""

>>> Circle.__class__    """Python 2.x: Neplatné = žádné metatřídy;
                           Python 3.x: OK = existují metatřídy"""
```

Případová studie: Python (III)

Třídní model Python 3

Metatřída = třída třídy



`c.radius` ≡ `c.__dict__['radius']`

`Circle.pi` ≡ `Circle.__dict__['pi']` ≡ `c.pi` ≡ `c.__class__.__dict__['pi']`

/* Viz knihovna inspect. */

Invokace metody – kontext

sender.callerMethod() :

- receiver.method(arg1, arg2)

- Kontext invokace metody:

- self/this (příjemce zprávy)
 - argumenty/parametry metody
 - lokální proměnné

- pozice v kódu metody
 - zá sobník kontextů (kontext sender.callerMethod, ...)
 - lexikální kontext (např. *closures*)

Typicky
nepřístupné

Typy v OOJ: 2 přístupy

- ❑ **Čistý** = vše je objekt (Smalltalk, Python 3, SELF)
- ❑ **Hybridní** = odvození od strukturovaných/modulárních jazyků
 - Základní/primitivní datové typy (čísla, znaky)
 - Datové struktury
 - **Třída** je speciální případ datové struktury
 - Často jiná syntaxe a sémantika oproti základním
- ❑ Hierarchie třídní dědičnosti:
 - s kořenovou třídou (Java, Python, Delphi) × bez ní (C++, PHP 5/8)

Implementace (třídního) OOJ

Požadavky × Skutečnost:

- ❑ Objektová paměť (asociativní, heterogenní)

×

- ❑ Počítačová paměť (lineární, homogenní)

-
- ❑ Model výpočtu: zasílání zpráv/invokace metod

×

- ❑ Instrukce CPU: CALL, zásobník volání

Implementační problémy

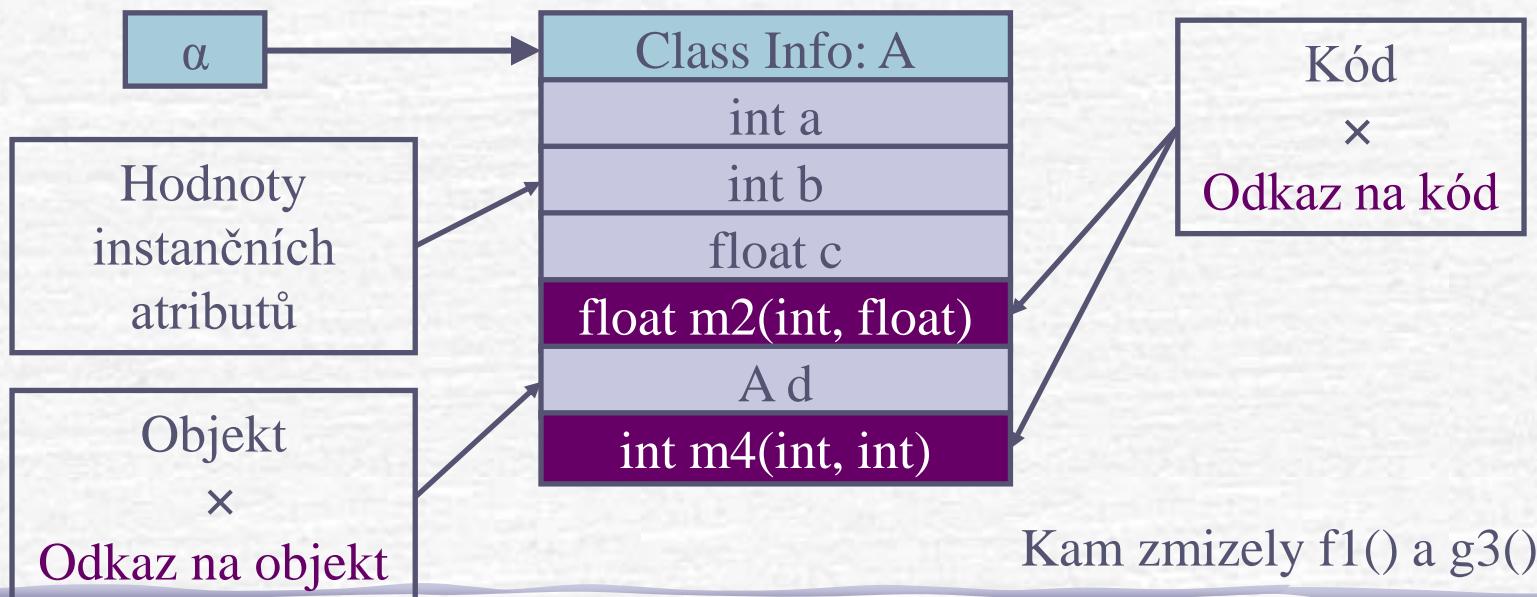
- ➊ Uložení instancí (atributů i metod)
 - Přístup ke zděděným položkám
- ➋ Přístup k atributům a metodám
 - Reflektování třídní hierarchie
 - Polymorfní invokace metody (pozdní vazba)
- ➌ Problémy vícenásobné dědičnosti
 - implementace i rozhraní
- ➍ Obtížnější řešení ve **staticky typovaných** OOJ

Příklad

```
class A is
    attr int a, b;
    static method int f1(int);
    attr float c;
    polymorphic method float m2(int, float);
    nonpolymorphic method float g3(float);
private:
    attr A d;
    polymorphic method int m4(int, int);
endOfClass
```

Uložení instance v paměti: Návrh 1

- Analogie s datovými strukturami (atributy za sebou)
 - Ale rozhodně ne variantními, tj. ne přes sebe (`union`)!
 - Statická struktura (*Class Instance Record, CIR, Layout*)



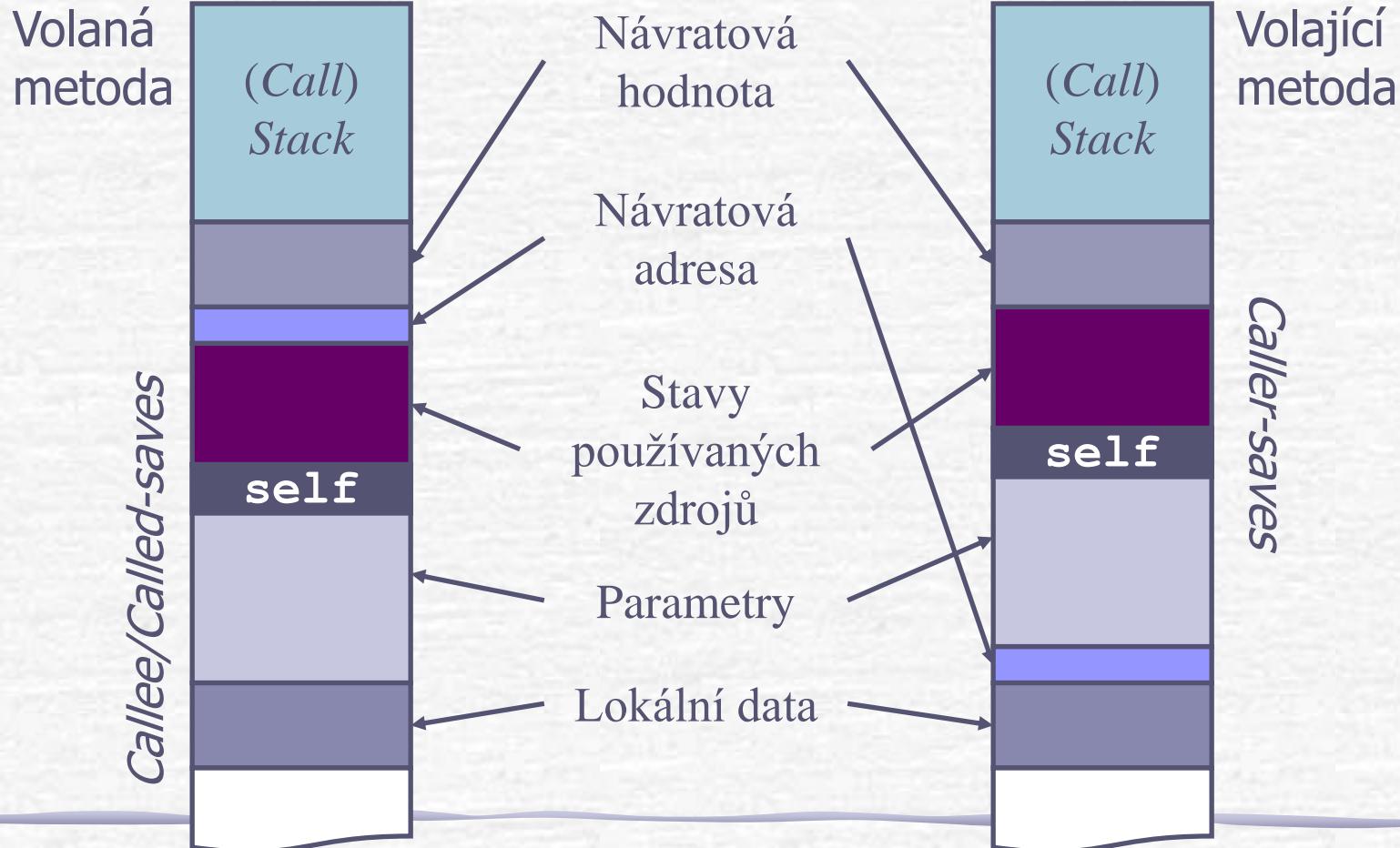
Statické třídní a instanční položky

- ☞ Statická funkce: **static method f1()**
 - Jako funkce ve jmenném prostoru třídy A
 - Chybí self-parametr ⇒ nemůže přistupovat k položkám instance
 - Může pracovat s dalšími statickými položkami
- ☞ Metoda s brzkou vazbou: **nonpolymorphic g3()**
 - Implementační kód vybrán při překladu (staticky)
 - Může přistupovat k položkám instance (má self-parametr)

Přístup k položkám instance

- ⌚ V těle metody přes *self*-parametr:
 - **self**, **this**, ..., vlastní pojmenování
 - klíčová slovo, (speciální) proměnná (jen pro čtení), ...
 - Aktuální i při volání zděděné metody
- ⌚ Implementace: „nultý“ (skrytý) parametr
 - Předáván implicitně (odkaz na instanci)
 - ABI (*Application Binary Interface*)
 - spolupráce odděleně kompliovaných modulů
- ⌚ Modifikátory viditelnosti (většinou řízeno staticky)

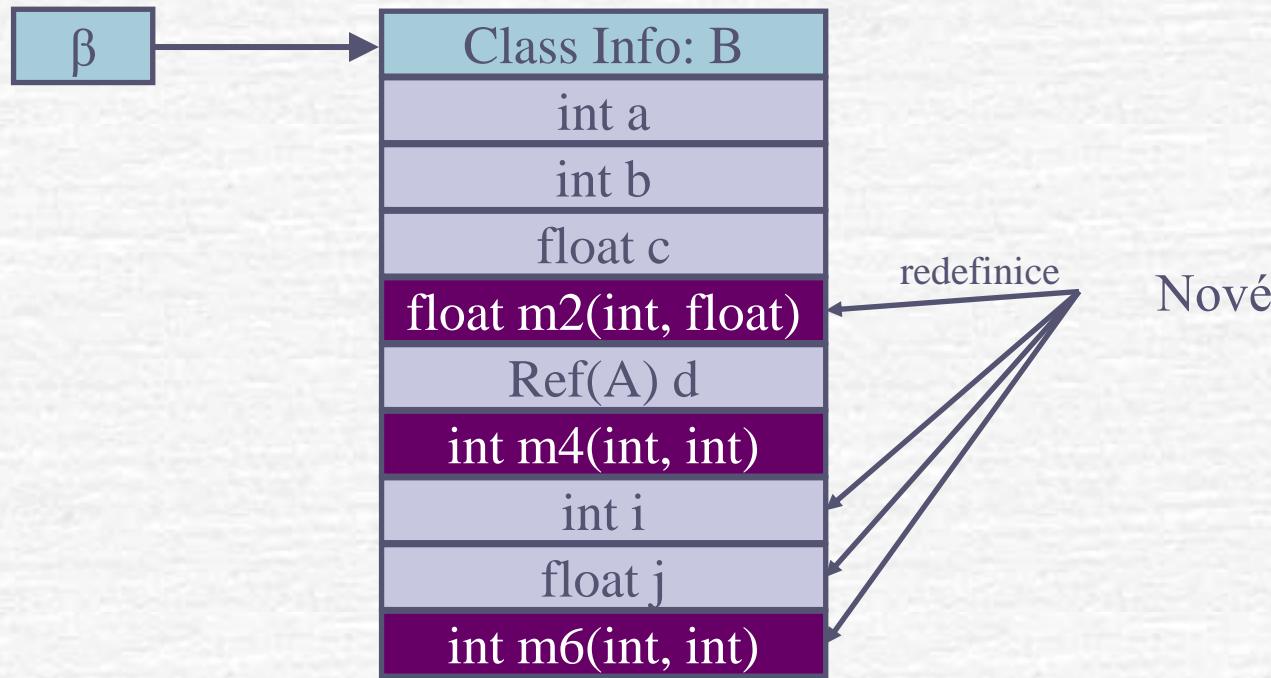
Zodpovědnost za správu zásobníku volání:



Příklad: Jednoduchá dědičnost

```
subclasse B of A is
    attr int i;
    static int f5(int);
    attr float j;
overriden polymorphic float m2(int, float);
overriden nonpolymorphic float g3(float);
private:
    polymorphic int m6(int, int);
endOfClass
```

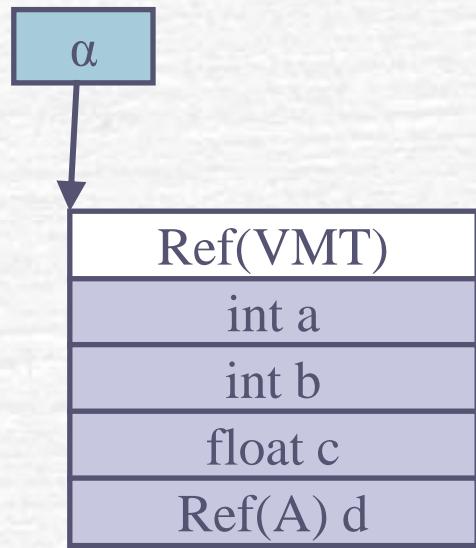
Uložení instance v paměti: Návrh 1



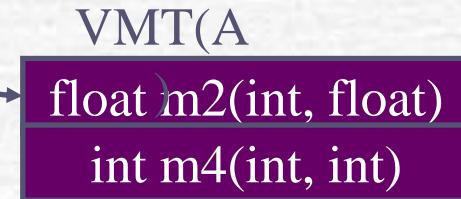
Problémy návrhu 1

- ☞ Každá instance třídy A (nebo B) nese stejnou sadu odkazů na metody
 - Reference na třídu(y)?
 - Uložení třídy v paměti za běhu?
 - Možné řešení: Každý *Extent* sdílí sadu metod
- ☞ Přístup k původním metodám při jejich redefinici (např. invokace $\beta.m2(...)$;)
 - potřeba vynucení hledání metody v nadtřídě
 - klíčové slovo/funkce: **super**, **base**, **super()**

Uložení instance v paměti: Návrh 2

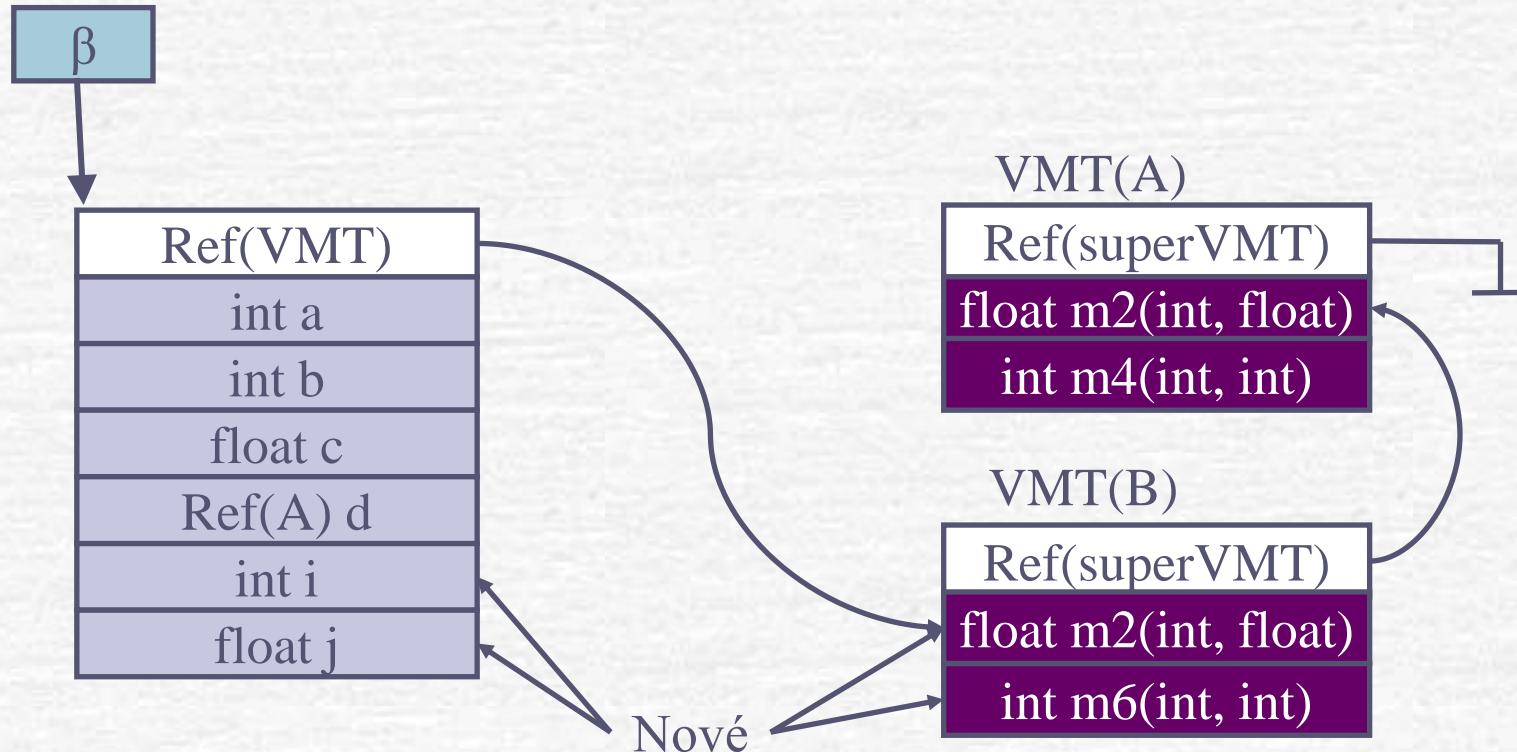


Tabulka (virtuálních) metod
((*Virtual*) Method Table, VMT/vtable) =
sada polymorfních metod dostupných
pro instanci



Odkaz na kód!

Uložení instance v paměti: Návrh 3

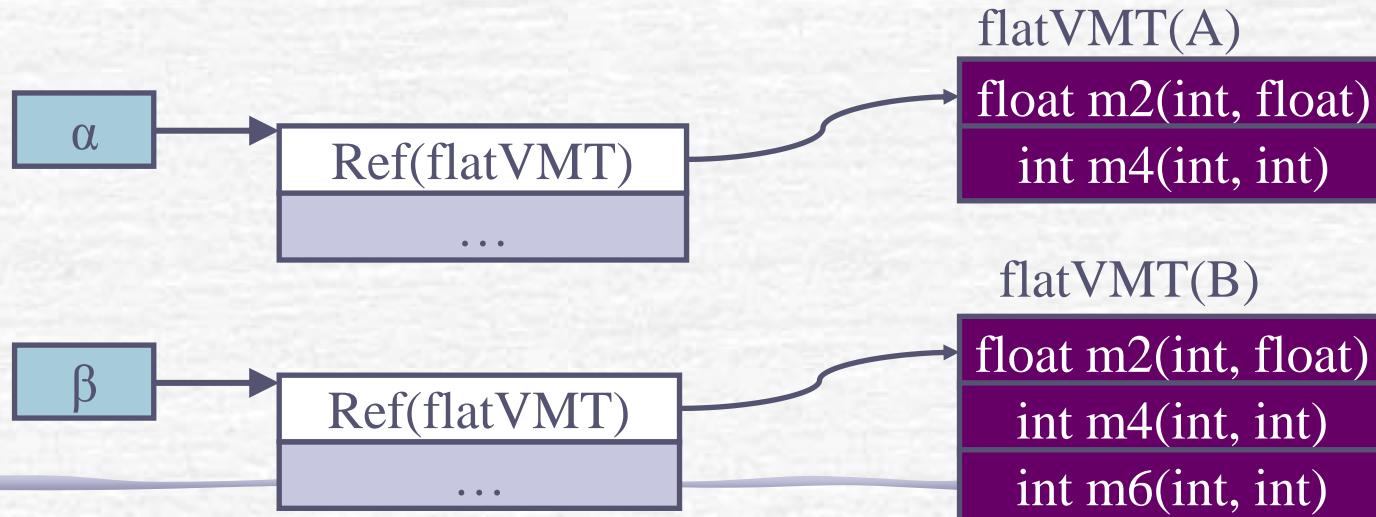


Problém toho návrhu:
přístup k neredefinovaným metodám

Uložení instance v paměti: finální, optimální?

Staticky typované OÖJ

- Každá třída „plochou“ VMT \Rightarrow méně dereferencí, stejné offsety
- Řešení *super*: využije statickou adresu VMT předka
- Neredefinovaná polymorfní metoda \Rightarrow nepolymorfní?



Vytváření instancí (Instanciace)

- ➊ Klíčové slovo **new** × Třídní zpráva
 - 1. Přiděl paměť pro instanci dané třídy
 - Většinou na haldě (× na zásobníku)
 - 2. Invokuj **konstruktor** (self, parametry)
 - Speciální metoda pro inicializaci atributů
 - Volání konstruktorů dle relace dědičnosti?
 - Statický × virtuální (externí/kopírovací)
 - Kovariantní návratový typ (redefinice vrací instanci podtřídy)

Případová studie (PHP 8: instanciace)

Truck.php

```
class Truck extends Car {  
    private float $load = 0;  
    public static int $count = 0;  
    function __construct(string $man = null,  
                          int $fuel = 0) {  
        parent::__construct($man, $fuel);  
        echo "Creating ".static::$count++. "th  
        ".__CLASS__;  
    }  
    function loadup(float $weight):static {  
        $this->load += $weight;  
        return $this;  
    }  
    //overriden  
    function drive(float $dist):static {  
        $this->fuel -= min($this->fuel,  
                           $dist/100 * 30); return $this;  
    }  
}
```

test_Truck.php

```
$skoda = new Car("Škoda");  
echo $skoda->refuel(10 /*1*/)  
    ->drive(150 /*km*/)  
    ->asString();  
  
$truck = new Truck("Tatra", 100);  
echo $truck->drive(100)  
    ->asString();  
  
  
if (rand(0,10) > 5)  
    $v = new Car("BMV", 40);  
else  
    $v = new Truck("Ford", 80);  
// receiver unknown until runtime  
echo $v->drive(100)  
    ->asString();
```

Rušení instancí/objektů

- ➊ Implicitní (*Garbage Collector*; Správce paměti)
 - Po zrušení všech referencí na objekt, je v objektové paměti nedosažitelný
 - problém cyklických referencí
 - časově náročné, neznámá doba sběru (explicitně?)
 - Případné volání finalizační metody (negarantováno)
 - Uvolnění paměti
- ➋ Explicitní (hlídá programátor, klíčové slovo **delete**)
 - Specialní metoda – **destruktur**
 - Automatické volání destruktorů dle dědičnosti?
 - Po provedení se objekt uvolní z paměti

Řízení toku programu (*flow control*)

Hybridní OOJ:

- Mix:

- Procedurální: příkazy volání funkce, větvení, iterace
- OO: instanciace, přiřazení, (polymorfní) invokace metody

Polymorfní invokace = polymorfismus

- Pozdní vazba (*late binding*) při invokaci polymorfních (virtuálních) metod

Třída jako typ

- B je podtřída A \wedge var β : InstanceOfType(A)
 $\Rightarrow \text{value}(\beta) \in \text{Extent}(A)$
- **Zahrnutí typu B do typu A (*subsumption*)** =
Pokud B je podtřída třídy A, lze instanci třídy B využít kdekoliv je očekávána instance třídy A. Pak statický přístup k β přes protokol A.
- Proto ve staticky typovaných OOJ nelze rušit položky.

Příklad: Polymorfní invokace metody

```
✓ static Main.work(o : InstanceOfType(A))  
    ...  
    o.g3(x, y); // nonvirtual in A and B  
    o.m2(y);    // overridden in B  
    ...  
end;  
✓ α = new A;  
β = new B; // subclass of A  
✓ ... Main.work(α) ...  
... Main.work(β) ...
```

Příklad: Režie invokace o.m2()

...

příprava a uložení parametrů
(vč. self-parametru)

...

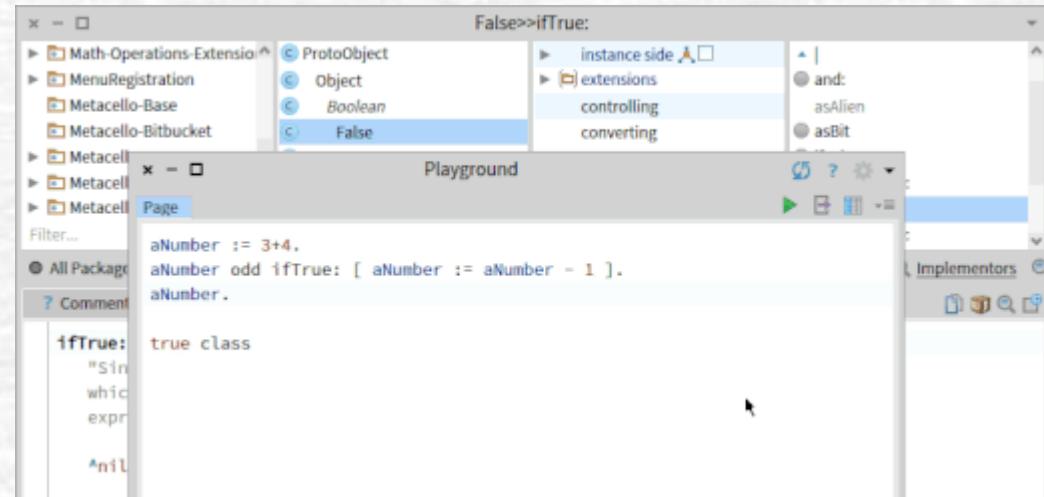
Load	Reg, [addr(o)]
Add	Reg, #offset_VMT // optimal.
Load	Reg, [Reg]
Add	Reg, #offset_m2_in_VMT
Call	[Reg]

Polymorfní invokace: Diskuze

- Invokace **Main.work (a)** nezajímavé
- Invokace **Main.work (b)** ⇒ invokace redefinované **m2 ()** (provede jiný kód)
 - Modifikace (vylepšení původního **m2 ()**)
 - Specializace/vylepšení původního algoritmu
 - Přes self-parametr přístup k novým položkám
 - Změna!
 - Nebezpečí úplné logické změny!

Vývojová prostředí (IDE)

- ✓ Podpora OOJ v IDE
- ✓ Zaměření na program a zdrojový kód → zaměření na instance/objekty
- ✓ Interaktivita = možnost explorativního programování (objekty dostupné v IDE)
- ✓ Vizualizace (např. diagram tříd)



Případová studie: Smalltalk (I)

- Definice tříd přes GUI IDE (Pharo, Squeak)
- Instanciace (třídní zpráva třídě):

- $t := Time now.$

- Blok kódu je také objekt:

- $b := [:arg | arg + 3].$

- Zaslání zprávy s jedním parametrem:

- $b value: 4.$

příjemce

zpráva

nepojmenovaný blok

formální parametr

argument

Případová studie: Smalltalk (II)

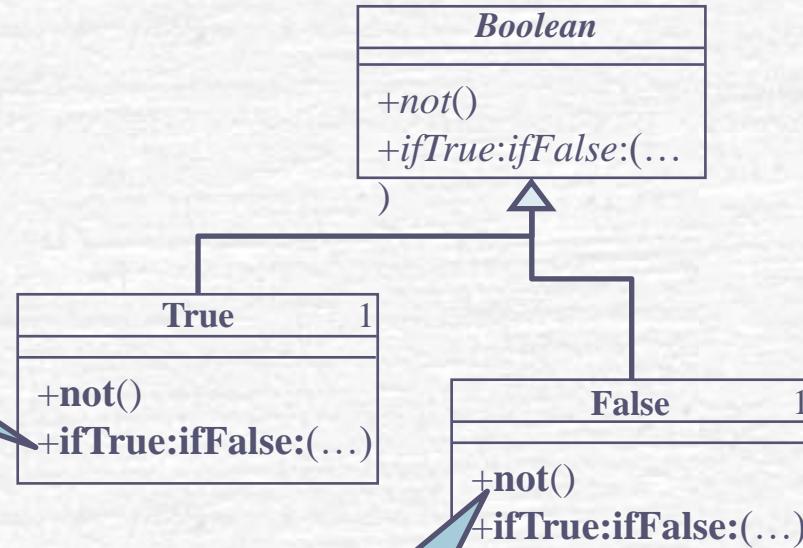
✓ true **class**.

⇒ True

✓ false **class**.

⇒ False

ifTrue: tb ifFalse: fb
 ^tb value.



✓ false **not** **ifTrue:** [^'Ahoj', ' svete!']
ifFalse: [^1+1].

⇒ ???

not
 ^true.

Případová studie: Smalltalk (III)

- ➊ Selektor zprávy je také objekt:
 - *mess* := #sin.
 - 20 **perform**: *mess*.
- ➋ Dotaz do objektové databáze:
 - (#(2 3 6 5 4 7 8) **select**: [:*x* | *x* > 5])
collect: [:*c* | *c* raisedTo: 20].



Případová studie: Smalltalk Pharo (IV)

PLACE
STAMP
HERE

"A method that illustrates every part of Smalltalk method syntax except primitives. It has unary, binary, and keyword messages, declares arguments and temporaries; accesses a global variable (but not an instance variable), uses literals (array, character, symbol, string, integer, float), uses the pseudo variables true, false, nil, self, and super, and has sequence, assignment, return, and cascade. It has both zero argument and one argument blocks."

<https://www.pharo.org>

Běhová prostředí OOJ

✓ Kompilované (OS, CPU)

- Binární kód

✓ Částečně interpretované (VM)

- Bajtkód, Virtuální stroj (*Virtual Machine*)
- Optimalizace pomocí JIT komplikace
- Multiplatformní (jednodušší portabilita)

✓ Interpretované (interpret OOJ)

- Opakuje „*front-end*“ analýzu zdrojového textu
- Vysoká reflektivita (přístup k metadatům)

Reflektivita (*Reflection*)

- ✓ systém zahrnuje kauzálně propojené struktury popisující sebe sama (alespoň částečně)

Dvě úrovně:

- a) Zkoumání vnitřní reprezentace (*Introspection*)
- b) (Kauzální) změna vnitřní reprezentace (*Intercession*)

- ✓ Dva typy

- **Strukturální** – pracuje se statickými strukturami (balíčky, třídy, metody)
- **Behaviorální** – pracuje s prováděním programu (invokace metody, přiřazení, zásobník volání)

Reflektivita – příklad ve Smalltalku

- ➊ Změna nadtídy jiné třídy (strukturální)
 - MyClass **superclass**: String.
- ➋ „Bakalář na FIT je pohůdka“ (strukturální)
 - Student **allInstancesDo**: [:stud |
stud **becomeForward**: (Bachelor **from**: stud)].
- ➌ Modifikace zásobníku volání (behaviorální)
 - | a |
a := 6 * 9.
thisContext **tempNamed**: #a **put**: 42.
Transcript **show**: a.

Činnost překladače

- Modulární OOJ – zvýšení složitosti překladače o dodržování ABI
 - *Linker* – typicky velké množství modulů
- Potřeba lokálních tabulek symbolů pro analýzu definic tříd (položek, vazeb)
- Zavádění jmenných prostorů pro snížení rizika kolize jmen

Lexikální a syntaktická analýza

- ➊ Podobná modulárním a blokově strukturovaným jazykům
 - Např. Python – bílé znaky součástí syntaxe
- ➋ Další části překladu jsou náročnější
⇒ požadavek na efektivní implementaci
(rozumně rychlou)

Sémantická analýza

Extrémně náročná

- Někdy i několik průchodů
 - Např. odstranění deklarací \Rightarrow víceprůchodová analýza jazyka Java
 - Komplikovanější typový systém (viz později)

Explicitní přetypování objektů

- Např. přetypování instance na typ podtřídy \Rightarrow typová kontrola možná až za běhu

Činnost interpretu

- Typicky překlad do mezikódu + interpretace
- Interaktivní konzole
 - Pracovní prostor (*workplace, shell*) pro „živé“ objekty
 - Mizí rozdíl vývojového a běhového prostředí
 - Často čistý OO model (objektová paměť)
 - Permanentní interpretace (po metodách/třídách)
 - Možnost interní optimalizace, ale z pohledu uživatele je OOJ formálně čistý

Pár doporučení na závěr

- ☞ OOP – mocné paradigma
 - Nepodceňovat ani nepřečeňovat
- ☞ Možnost tvořit složité konstrukce
- ☞ OOJ nabízí velkou variabilitu
 - statické × dynamické
- ☞ Zvyšuje se riziko špatného návrhu
 - dekompozice, řešení modularity

K zapamatování

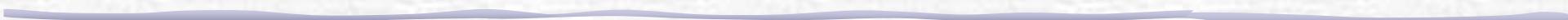
- ☛ Třída, třídní dědičnost a další pojmy
- ☛ Způsoby vytváření a rušení objektů
- ☛ Reprezentace objektů v paměti
 - Tabulka virtuálních metod (VMT)
 - Brzká a pozdní vazba

Cvičení

- 👉 Analyzujte rozdíl mezi modulárním a objektově-orientovaným paradigmatem:
 - Kolik času Vám zabere vytvořit modul pro vybraný ADT? (bez použití vestavěných ADT)
 - Je možné jako parametry využít libovolné druhy (typy) dat nebo jsou omezeny? (objekty/struktury/primitivní typy)
 - Jaké máte zkušenosti s výměnou modulů s kolegy? Jak dobře se Vám modul používá? Je modul dobré navržen (protokol, ...)?
- 👉 Vyzkoušejte některou implementaci jazyka Smalltalk
 - www.pharo.org, www.squeak.org



Typový systém objektově-orientovaných jazyků



Cíl přednášky

- ⌚ Vyžádaná dědičnost vs skutečné typy
- ⌚ Vícenásobná dědičnost
- ⌚ Rozhraní
- ⌚ Role
- ⌚ Perzistence

Typový systém

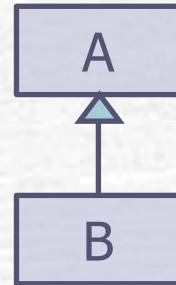
typicky

- ➊ Typový systém = způsob určování a kontroly typů (včetně operací nad nimi)
- ➋ Kategorie OOJ dle určení typu proměnné
 - Typované (typ dán typovou anotací/inferencí; např. `Car a = new Superb()`)
 - Netypované (typ dán hodnotou; např. `p = Employee()`)
- ➌ Kategorie OOJ dle doby typové kontroly
 - Staticky typované = většina kontrol při překladu/linkování (např. Java, C++, C#)
 - Dynamicky typované = kontroly až za běhu (např. Python, PHP, Smalltalk, Ruby)

Typy × Třídy

- ➊ Typ = množina validních hodnot a operací nad nimi
- ➋ Třída = množina vnitřních stavů a operací nad nimi
 - včetně dědičnosti a polymorfismu
 - Vlastní třída instance (tj. sloužící pro vznik této instance)
- ➌ Typy jsou obecnější a dynamičtější koncept
 - Podtyp B typu A = $\text{Members}(A) \subseteq \text{Members}(B) \wedge$ stejné chování navenek
 - Podtřída není podtypem, dojde-li např. ke změně chování (překrytím, změnou viditelnosti veřejné položky)
 - Podtyp nemusí být podtřídou, např. dvě nesouvisející třídy implementují stejné položky
 - Praktický pohled OOJ: Třída určuje typ nehledě na redefinici

$$\text{B je podtřída A} \Rightarrow (\text{B je podtyp A} \Leftrightarrow \text{Members}(A) \subseteq \text{Members}(B))$$



Přístupy pro určení typu

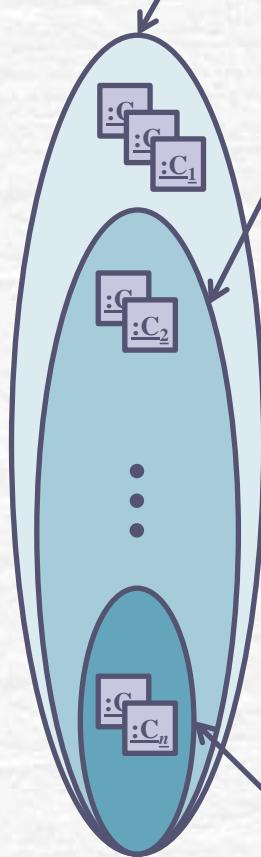
1. Typ daný jménem (*nominal typing*)
 - Např. (jednoduchá) dědičnost (staticky)
 - Vyžádaná dědičnost (*required inheritance*)
2. Typ daný výčtem položek (*structural typing*)
 - Rozpoznávám skutečný (pod)typ (dynamicky)
3. Kombinace (výčet jmen)
 - Např. vícenásobná dědičnost tříd nebo rozhraní (staticky) nebo systémy s rolemi (dynamicky)

Typ daný jménem

- ➊ Typ daný jménem třídy a relací dědičnosti
- ➋ Zahrnutí: Mějme „funkci“ f s parametrem přijímajícím instance třídy A, pak parametr může být:
 - instance třídy A (\Rightarrow stejného typu)
 - instance lib. podtřídy A (\Rightarrow kompatibilního podtypu)
- ➌ Třídní dědičnost = částečné uspořádání na třídách:
 - reflexivita (nikoli syntakticky, $A(A)$), tranzitivita
 - antisymetrie (žádný cyklus, $A(B), B(A)$)

Typ daný jménem

$\text{Extent}(C_1) = \text{instance vlastních tříd } C_1, C_2, \dots, C_n$



Podtřída × Podtyp

$\text{Extent}(C_2) = \text{instance vlastních tříd } C_2, \dots, C_n$

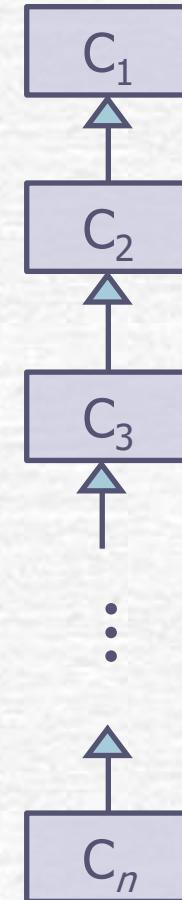
- Třída C_i je přímou podtřídou C_{i+1} , $1 < i \leq n$
- Třída C_n je podtřídou (dědí od) třídy C_1
- Třída C_n je podtyp třídy C_1

$\text{Members}(C_1) \subseteq \text{Members}(C_n)$

×

$\text{Extent}(C_n) \subseteq \text{Extent}(C_1)$

$\text{Extent}(C_n) = \text{instance vlastní třídy } C_n$

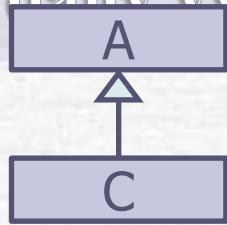


Vyžádaná dědičnost

- ➊ Příklad: `static f(p : InstanceOfType(B or C))`
 - Dědičnost pro „sdílení“ jména místo implementace
- ➋ Pro zajištění kompatibilního typu instance musíme použít dědičnost
 - Dědičnost zajišťuje existenci položek (atributů a polymorfních metod)
 - Využíváno (statickou) typovou kontrolou
- ➌ Např. Java (bez rozhraní), Delphi, ..., C++ (bez vícenásobné dědičnosti)

Skutečné (pod)typy

- ➊ Zjištění typu prozkoumáním položek instance
 - Typicky nás zajímá pouze potřebný podtyp
- ➋ Test implementace potřebného podtypu
 - Během překladu (potřeba staticky typovaný jazyk)
 - Např. Jazyk Go (rozhodnutí o implementaci „interface“ objektem dle seznamu skutečně implementovaných metod)
 - Za běhu
 - Tzv. *Duck (dynamic) typing*
 - Polymorfismus nezávislý na třídní dědičnosti
 - Např. Smalltalk, Python, Objective-C, (beztřídní SELF)



Příklad

- ☞ **class A is**
 int d1;
 float d2;
 float m(int);
endOfClass
- ☞ **subclass C of A is**
 int d3;
 float m(int);
 int g(float);
endOfClass

- ☞ **class B is**
 float d2;
 int d1;
 int h(float);
 float m(int);
endOfClass

Příklad: Srovnání

```
static p(o : InstanceOfType(A)) ;  
static q(o : InstanceOfType(C)) ;  
  
    α = new A;  
    β = new B;  
    γ = new C;
```

✓ Vyžádaná dědičnost

- ✓ call p(α) OK
- call p(β) *ERROR*
- call p(γ) OK
- ✓ call q(β) *ERROR*
- call q(γ) OK

✓ Skutečné podtypy

- ✓ call p(α) OK
- call p(β) OK
- call p(γ) OK
- ✓ call q(β) *ERROR?*
- call q(γ) OK

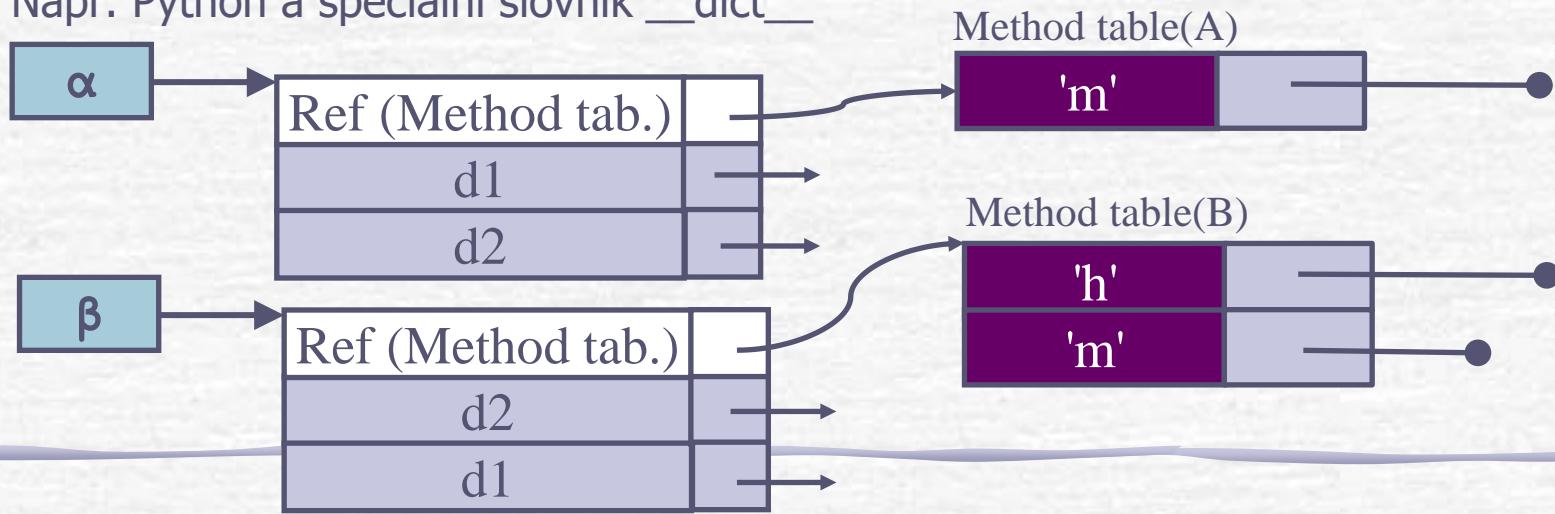
Implementace skutečných podtypů

Jak najít požadovanou položku?

- ☛ Nezávislost na třídě ⇒ různé pořadí položek v objektech, ale kód volání metod je statický
⇒ Nutný **jednotný přístup** pro všechny objekty a nutnost řešit **až za běhu!**
- ☛ Interní rutiny běhového prostředí pro
 - přístup k (veřejným) atributům
 - %set/get<object, attribute_name>
 - přístup k (veřejným; polymorfním) metodám
 - %getMethodAddress<object, method_name>

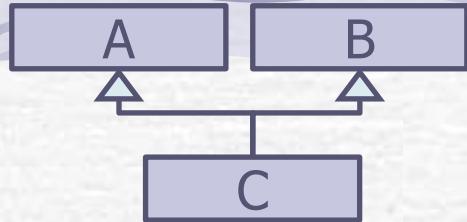
Důsledky pro implementaci

- Typicky (částečně) interpretované a dynamicky typované OOJ
- Hledání reakce na zprávu dle selektoru
 - Atributy jako slovník *atribut* → *hodnota*
 - Tabulka metod jako slovník *selektor* → *metoda*
 - Např. Python a speciální slovník `__dict__`



Vícenásobná (třídní) dědičnost

- Zděděná třída má více jak jednu **přímou** nadřídu.
- Např. C++, Python, CLOS
- Diskutabilní vlastnost OOJ:
 - Častý výskyt v OOA a OOD
 - Často není při programování nutná
 - Problémy s implementací

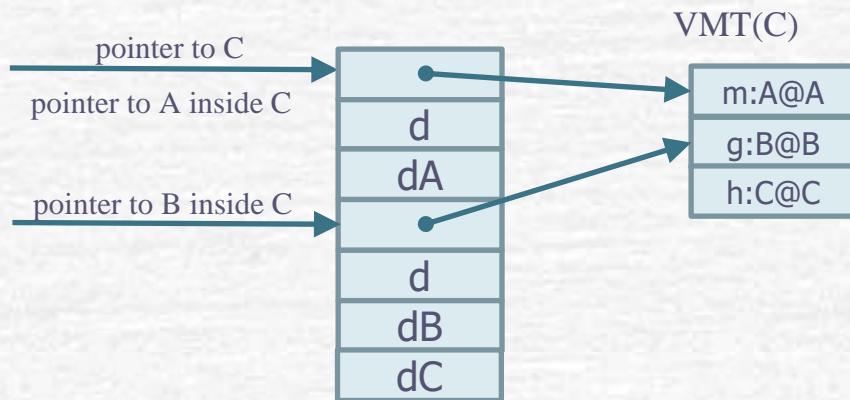


Všechny metody
jsou polymorfní!

Příklad

- ☞ **class A is**
 int d;
 float dA;
 float m(int);
endOfClass
- ☞ **class B is**
 int d;
 float dB;
 float m(int);
 int g(float);
endOfClass

- ☞ **subclass C of A, B is**
 int dC;
 int h(int);
endOfClass



Instance vlastní třídy C

Problémy vícenásobné dědičnosti

- ➊ Když (přímé) nadtíry obsahují položky stejného jména a typu
- ➋ Když (přímé) nadtíry obsahují položky stejného jména, ale různých typů
- ➌ Inicializace instancí
 - Pořadí volání konstruktorů
- ➍ Uložení instancí v paměti
 - Instance třídy C (C je přímá podtřída A i B) lze využít kdekoli očekávám instance tříd A nebo B.

Problémy: Metody I

Metody stejného jména a typu v definici nové podtřídy

- Zakázáno (např. SELF – kontrola za běhu)
- Skrytí (např. A::m() přístupná jen v A, ne v C)
- První nalezený výskyt (např. Python)
- Povinná redefinice
 - Plná kvalifikace (např. C::m(){ ...A::m()... })
 - Problém přístupu k instanci třídy C jakoby to byla instance třídy B (tj. budu volat ...A::m()...)

Problémy: Metody II

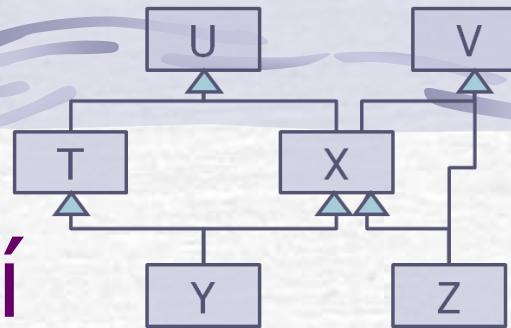
- ➊ Metody stejného jména a různých typů v definici nové podtřídy
 - Zakázáno
 - Povolit jejich souběžnou existenci:
 - Implementace přetěžování metod (*overloading*)
 - Další zátěž překladače, spojovače, podpory běhového prostředí

Problémy: Atributy

▀ Atributy stejného jména a typu

- Zakázáno
- Skrytí (např. A::d přístupný jen v A, ne v C)
- Sloučení
 - Jediný výskyt (vazba přes jméno, např. Python)
 - Replikace atributu v paměti (kompilované)
 - Problém konzistence: Instance C použita jako by to byla instance A \Rightarrow změna v A::d by se měla promítnout i v B::d

▀ Stejnojmenné atributy různých typů jsou zakázány



Inicializace instancí

1. Implicitní volání konstruktorů nadtíříd:

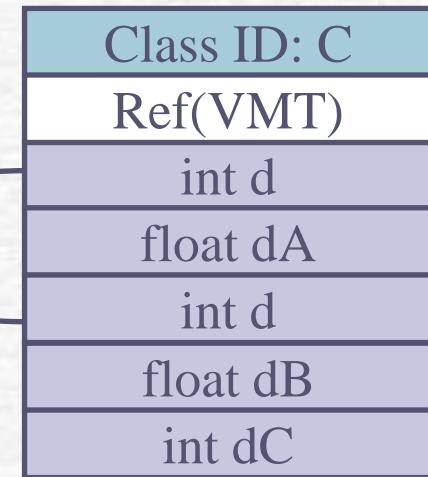
- ✓ Využití pořadí zápisu nadtíříd ve zdrojovém textu a prohledávání do hloubky (DFS)
⇒ **acyklické (opačné) pořadí volání konstruktorů**
Např. new Y ⇒ U(), T(), V(), X(), Y()

2. Explicitní (uživatelem definované):

- Nebezpečí uváznutí (*dead-lock*) či vynechání některého konstruktoru

Uložení instancí v paměti

- ⌚ Není univerzální řešení
 - Závisí na konkrétním OOJ
- ⌚ “Položky za sebou”?
 - Duplicace položek stejného jména a typu \Rightarrow interní rutiny běhového prostředí pro zachování konzistence sloučení
- ⌚ Diamant v hierarchii?
 - virtuální dědičnost



Rozhraní (*Interfaces*)

- ☞ Schéma deklarující sadu metod
- ☞ Objekt/třída implementující/realizující toto rozhraní musí tyto metody implementovat

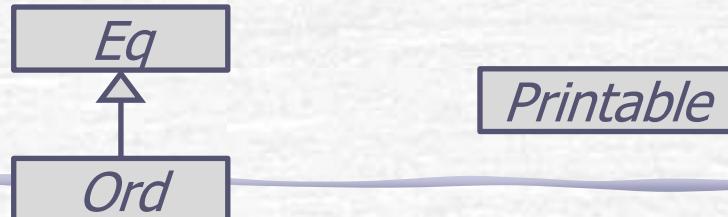
- ☞ Např. Java (zpopularizováno), C#, D, PHP 5
- ☞ Možnost i deklarace atributů?
 - Teoreticky ano, ale většinou nepodporováno
 - **Abstraktní třída**
 - Možnost částečné implementace (min. 1 abstraktní metoda)
 - Nelze vytvářet instance; Nelze vícenásobně dědit

Vícenásobná dědičnost rozhraní

- 🕒 Hierarchie vícenásobné dědičnosti rozhraní (částečné uspořádání na rozhraních)
 - Realizace rozhraní pak musí implementovat i metody zděděné z nadrozhraní.
- 🕒 Většinou kombinace s jednoduchou třídní dědičností
 - Třída implementuje libovolný počet rozhraní
 - Včetně metod deklarovaných v „nad-rozhraních“

Příklad: Definice rozhraní v Javě

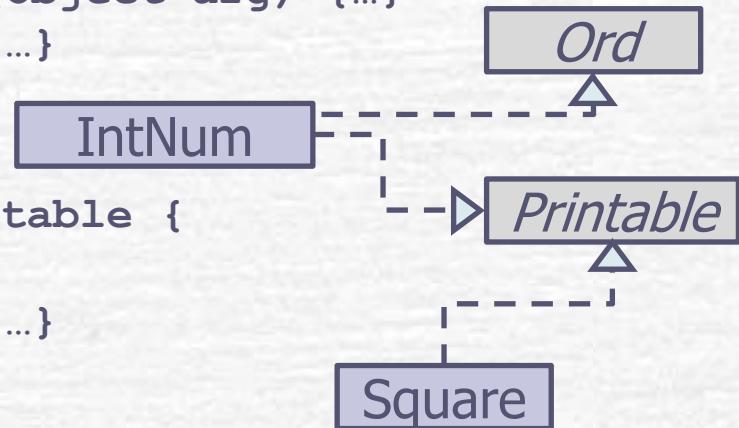
- ➊ interface Eq {
 bool isEqual(Object arg); //implicitly public
}
- ➋ interface Ord extends Eq { //skrytý self-parametr
 bool lessThan(Object arg);
}
- ➌ interface Printable {
 void print();
}



Příklad: Realizace & užití rozhraní v Javě

```
class IntNum implements Ord, Printable {  
    ...  
    public bool isEqual(Object arg) {...}  
    public bool lessThan(Object arg) {...}  
    public void print() {...}  
}
```

```
class Square implements Printable {  
    ...  
    public void print() {...}  
}
```



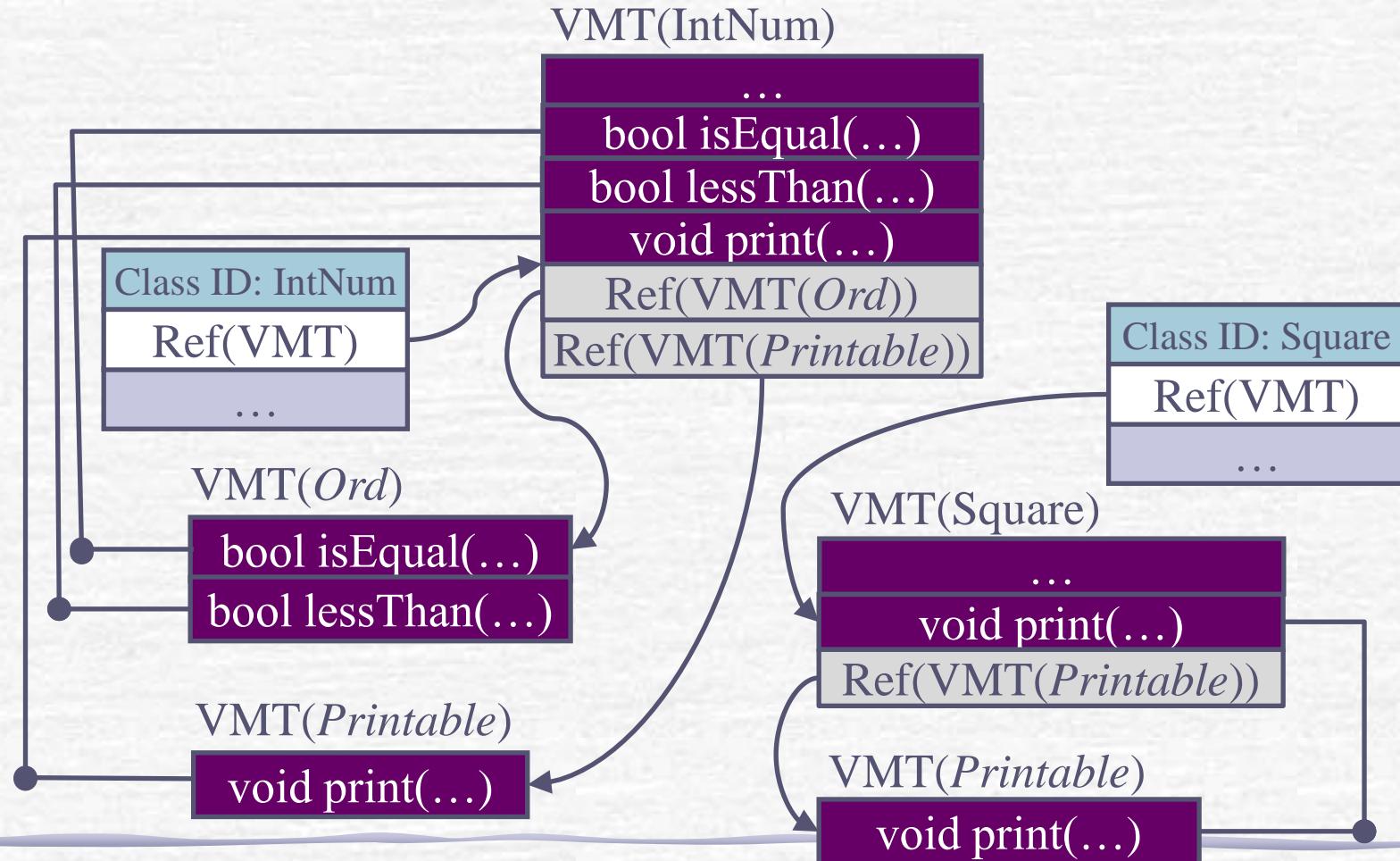
Rozhraní a polymorfismus

- Na místě parametru vyžadujícího rozhraní lze použít instanci libovolné třídy implementující toto rozhraní
- Rozhraní jako typ proměnné
 - `Printable sq = new Square();`
- Např. Generická funkce pro řazení libovolné kolekce
 - `static sort(interface Ord[]);`

Implementace Rozhraní

- Jednodušší než vícenásob. třídní dědičnosti
- Problémy k řešení
 - Vazba přes jméno, nutnost slovníků (Python)
 - Kontrola výčtu implementovaných rozhraní
 - Výběr správné VMT (Java)
 - Interní rutiny %get/setMethodAddress<...>
 - komplilovaná vazba přes jméno

Implementace Rozhraní - Návrh



Rozhraní objektu × Protokol

- Rozhraní (koncept) = pojmenovaný seznam metod k implementaci
- Rozhraní objektu = množina zpráv, kterým se objekt zavazuje rozumět
- Protokol (obecně) = postup hledání reakce na zprávu
- Protokol (alternativně) = množina VŠECH zpráv, kterým objekt rozumí
 - Určen množinou zděděných a nových položek (včetně implementovaných Rozhraní)

Perzistence objektů

A. Perzistentní objekt (instance) = přežívá dobu běhu aplikace; při dalším spuštění aplikace je opět k dispozici (se stejným stavem i **identitou** jako při vypnutí aplikace)

Co není perzistence?

- Ukládání snímků celé objektové paměti
- Ukládaní/načítání objektů na aplikační úrovni

B. Transientní (dynamický) objekt = neperzistentní

Implementace perzistence I

■ Klíčové slovo (*persist*) pro deklarativní označení perzistentních objektů (např. Intersystems Caché)

a) Jen hodnoty atributů objektů

- Ukládání/načítání stavu pomocí „knihovny“
- Metody nutno řešit jinak (problém přidávání dříve neznámých rolí)
- Teoreticky nečistý model perzistence

Implementace perzistence II

b) Ukládáme položky objektu (příp. třídu)

- Typicky využíváme (semi-)interpretovaných systémů (podpora VM)
- Problémy u komplikovaných systémů:
 - Speciální knihovny pro načítání spustitelného kódu (ala zavaděč aplikace v OS)
 - Velmi závislé na cílové platformě

K zapamatování

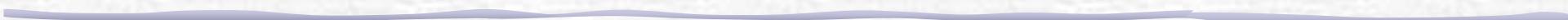
- ☛ Vyžádaná dědičnost, skutečné podtypy
- ☛ Problémy vícenásobné dědičnosti
- ☛ Implementace Rozhraní
- ☛ Systémy s rolemi
 - Vícetypové objekty, operace nad nimi
 - Perzistence

Cvičení

- ↗ Porovnejte Rozhraní a vícenásobnou dědičnost tříd. Obhajujte jedno před druhým.
- ↗ Promyslete možnost implementace systému s rolemi v některém staticky typovaném jazyce (např. Java nebo C++).



Beztrídní objektově-orientované jazyky



Cíl přednášky

- ⌚ Charakteristika
- ⌚ Jazyk SELF
- ⌚ Dynamická dědičnost
- ⌚ Explorativní programování

Charakteristika beztřídních OOJ

- Vytváří objekty klonováním prototypů
- Dědičnost = sdílení položek mezi objekty
 - Realizace pomocí delegace = zpráva přeposlaná rodiči nese i původního příjemce
 - Dynamická dědičnost = možnost měnit rodiče za běhu
 - Vícenásobná = více přímých rodičovských objektů (*parent*)
- Většinou interpretované systémy
 - Propojení vývojového a běhového prostředí, tzv. inkrementální interaktivní (explorativní) programování
- Př. beztřídních OOJ (založených na prototypech):
 - SELF, Slate, Isaac (kompilovaný), JavaScript, Lua, ...

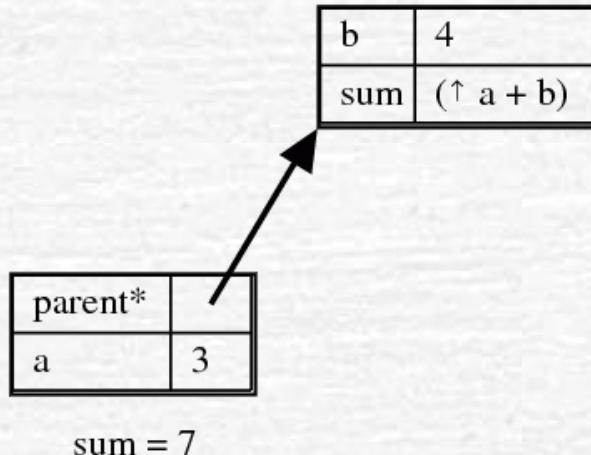
Případová studie: SELF (I)

Objekt

- Seznam položek, tzv. sloty

Slot

- Jméno + odkaz:
 - na datový objekt
 - na objekt s metodou
 - na rodičovský objekt (např. `parent*`)



Video ukázka SELFu: <http://www.youtube.com/watch?v=NYEKO7JcmLE>

Objektový model jazyka Self: <https://www.abclinuxu.cz/blog/squeaker/2018/12/objektovy-model-jazyka-self>

Případová studie: SELF (II)

>Data

- čtení (zpráva `name`), zápis (zpráva `name:`)

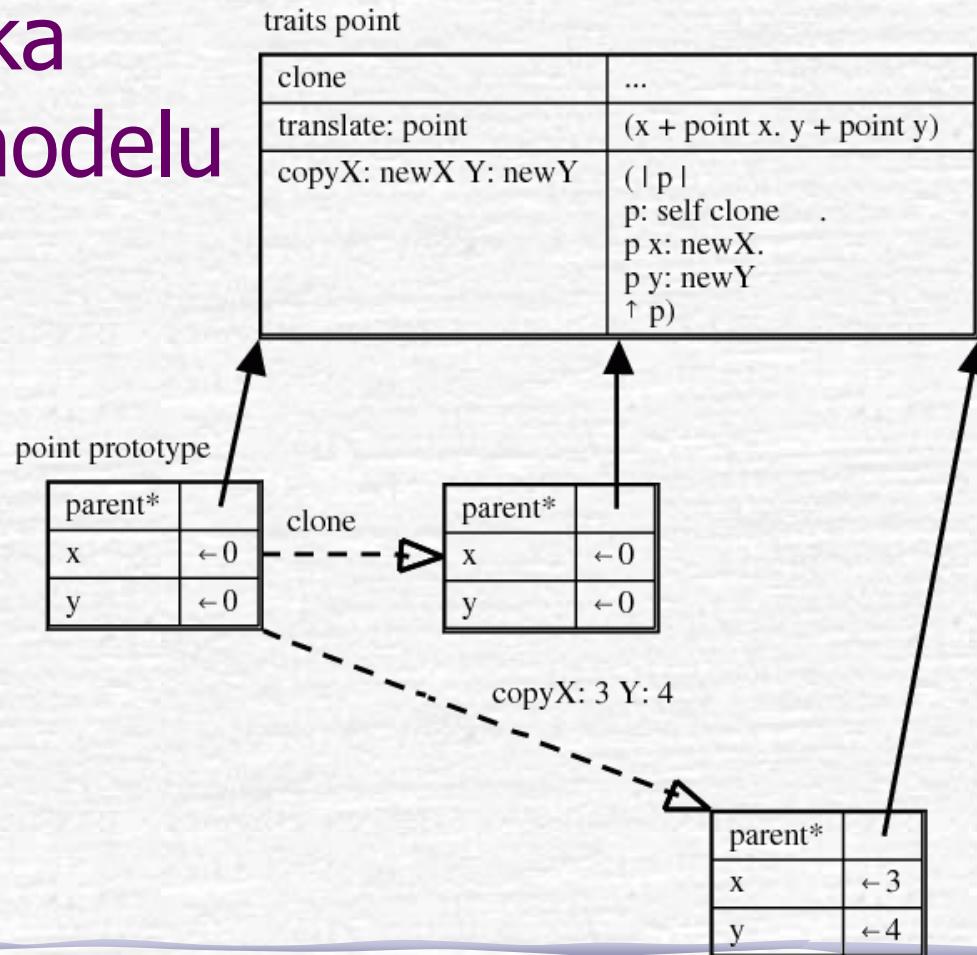
Metoda

- Objekt metody s jejím kódem
- Při invokaci klonuji ⇒ **aktivační objekt** (a.o.)
 - sloty pro parametry metody a lokální proměnné
 - a.o. tvoří jmenný prostor (kontext invokace)
- Chybí klíčové slovo `self`, implicitní
 - a.o. obsahuje rodičovský slot na hostující objekt

SELF: „Třídní“ model

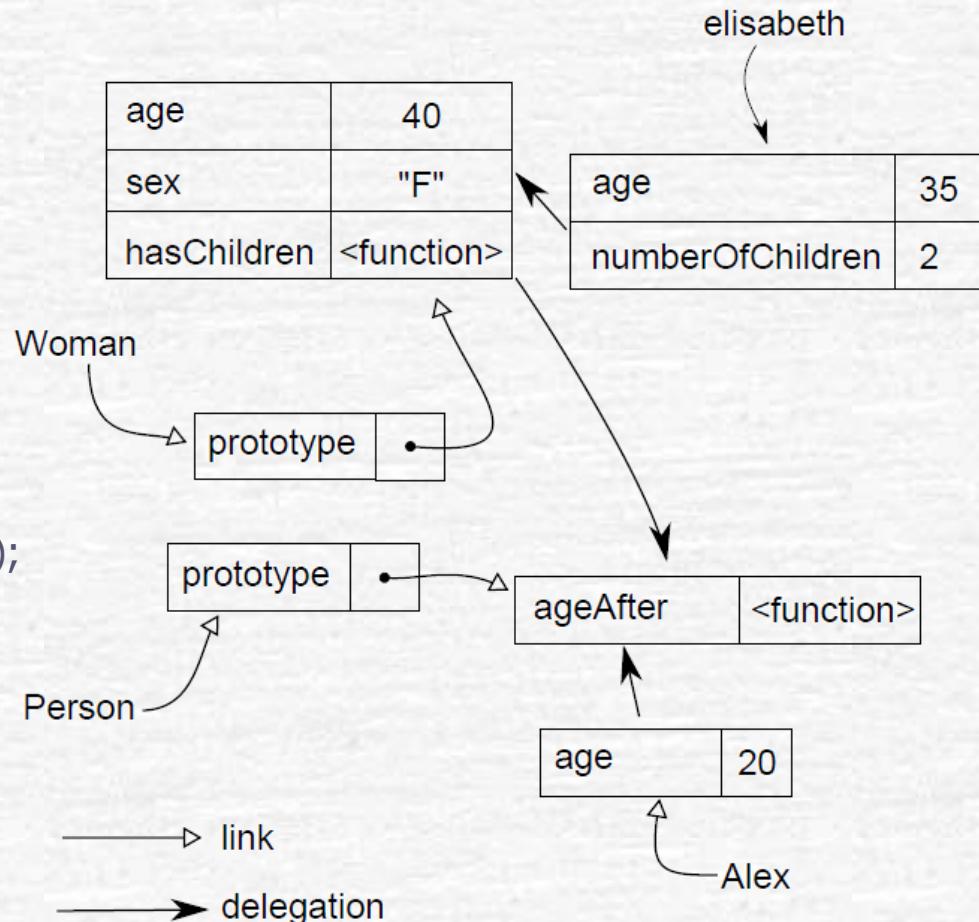
- „Třída“, tzv. *rys* (*traits*) = objekt pouze se sdílenými metodami a rodičovskými sloty pro delegaci, tj. sdílení implementace
- Prototyp
 - Obsahuje datové sloty pro atributy a jejich implicitní hodnoty, tj. sdílení reprezentace datového typu
 - Deleluje na rysy (simulující rodičovské „třídy“)
 - přístup ke sdíleným „instančním“ metodám
- Vytvoření nové „instance“
 - Naklonuje prototyp (mělká kopie ⇒ vlastní odkazy)

SELF: Ukázka „třídního“ modelu



Případová studie: JavaScript

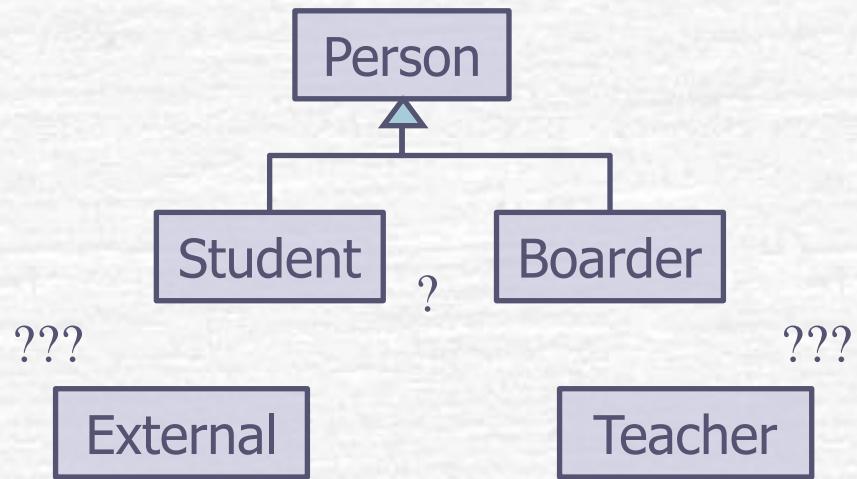
```
function Person(age) {  
    this.age = age };  
  
var Alex = new Person(20);  
  
Person.prototype.ageAfter =  
    function (time) { ... };  
  
function Woman(a,n) {  
    { this.age = a;  
    this.numberOfChildren = n};  
  
Woman.prototype = new Person(40);  
Woman.prototype.sex="F";  
Woman.prototype.hasChildren  
    = function () { return ... };  
  
var elisabeth =  
    new Woman(35, 2);
```



Systémy s rolemi

- ☞ Vícetypový objekt = objekt má **najednou** více typů, tzv. **role** (např. více přímých předků) a **dynamicky** tyto role nabývá/pozbývá bez nutnosti změny své identity.
- ☞ Použití:
 - OO databáze (ACID vlastnosti)
 - interpretované systémy s dlouhou dobou života objektů (podpora např. v SELF, Python, JavaScript)

Motivační příklad



Řešení vícenásobnou dědičnosti?

- mnoho kombinací
- nepředvidatelnost budoucnosti

Nové požadavky:

- Menza chce umožnit stravování i externistů
- Ne každý student chodí do menzy
- Někteří uživatelé se stanou strávníky později

Operace s vícetypovými objekty

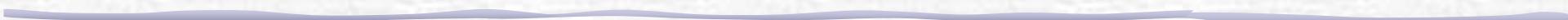
Dodatečné požadavky na operace za běhu:

- ⌚ Objekt spravuje svou identitu (role), ne třída
 - 1) Operace vytvoření objektu (*create*)
- ⌚ Pohled na objekty skrze specifickou roli (typ)
 - 2) Přetypování (*cast*)
- ⌚ Změna objektem podporovaných rolí
 - 3) Přidání role (*bind*)
 - 4) Odebrání role (*unbind*)

K zapamatování

- ☛ Jazyk SELF: typy slotů
- ☛ Prototyp, klonování
- ☛ Rys, sdílení implementace
- ☛ Dynamická dědičnost – delegace

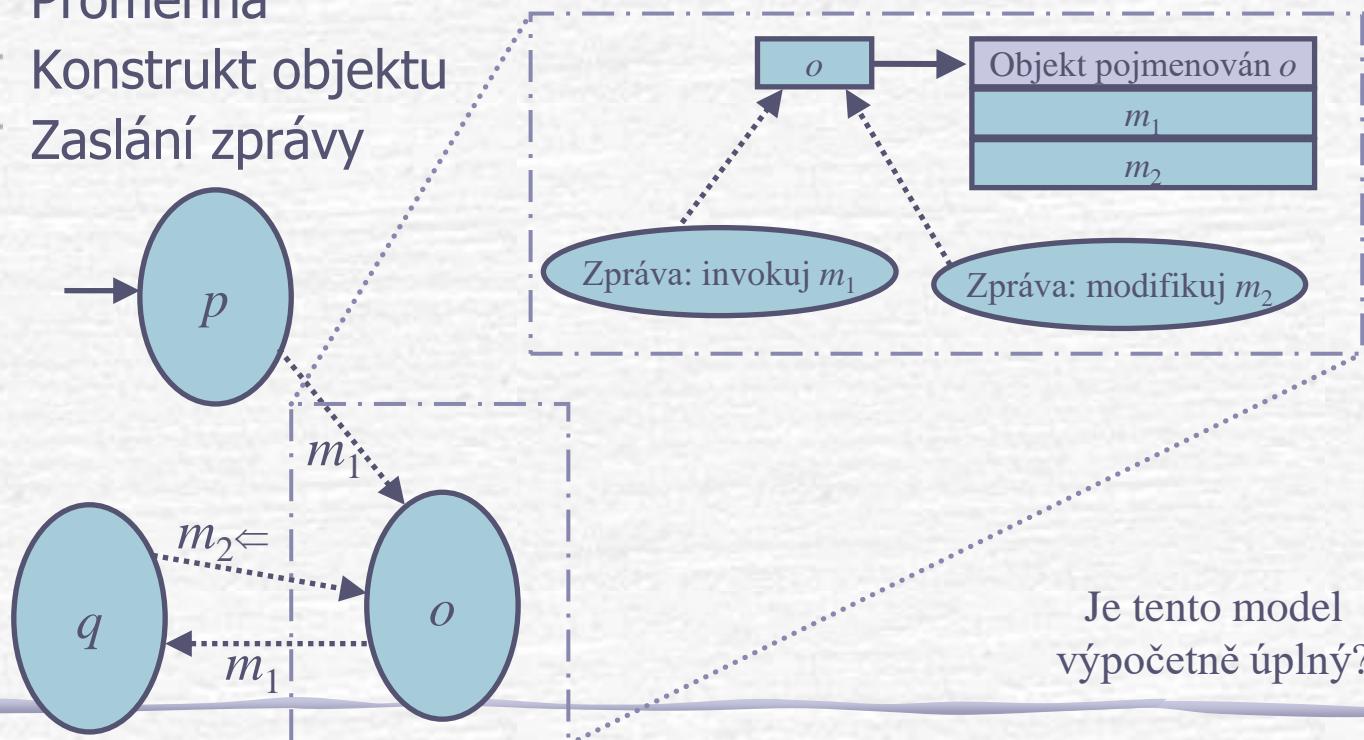
- ☛ <http://www.selflanguage.org/>



Teoretický objektově- orientovaný jazyk: ς -kalkul

Minimální OO model

- ⌚ Proměnná
- ⌚ Konstrukt objektu
- ⌚ Zaslání zprávy



Příklad formální báze OOP

- ς -kalkul – popisuje minimální model OOP
 - program obsahuje **jediný výraz** (příp. pomocné definice)
 - výsledek každého ς -výrazu je objekt
 - deklarativní, beztypový
 - formálně i sémantika jazyka \Rightarrow dokazatelnost korektnosti
 - ς je varianta řeckého písmene „sigma“

ζ -kalkul – Syntaxe

• ζ -výraz je výraz tvaru:

- Proměnná (slovo z písmen a číslic anglické abecedy)

- x

- Objekt (seznam metod se ζ -parametrem)

- $[l_1 = \zeta(x_1)e_1, l_2 = \zeta(x_2)e_2, \dots, l_n = \zeta(x_n)e_n]$

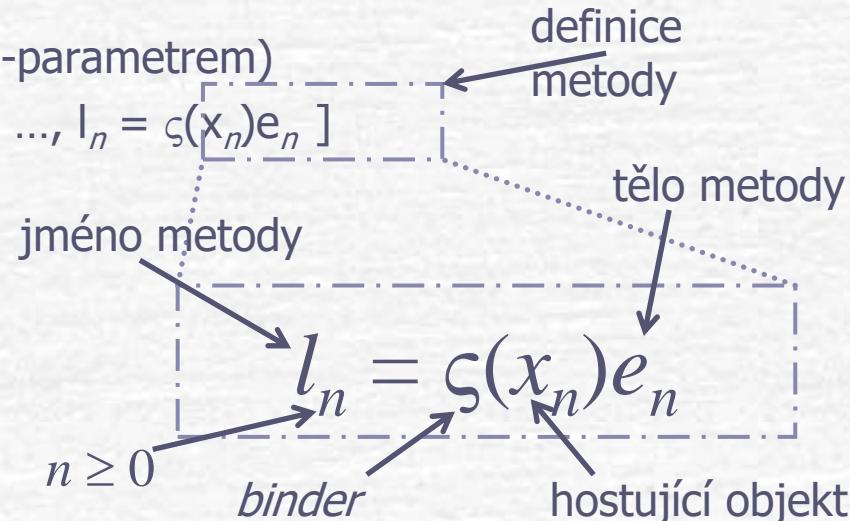
- Invokace metody

- e.l

- Modifikace metody

- $e.l \Leftarrow \zeta(x)e'$

- Závorky



ς -kalkul – Příklady

- Prázdný objekt

[]

- Objekt se dvěma metodami

[$I = \varsigma(x)[], m = \varsigma(y)y$]

- Invokace metody

[$m = \varsigma(x)x.n, n = \varsigma(y)y.m$].n

- Modifikace metody

[$m = \varsigma(y)y$].m \Leftarrow $\varsigma(x)x.m$

ζ -kalkul – Model výpočtu

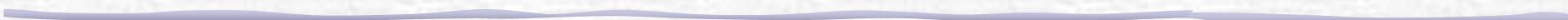
- ☞ ζ -výraz pro invokaci nebo modifikaci metody postupně redukuji (\hookrightarrow) na neredukovatelný ζ -výraz
- ☞ Redukce invokace metody
 - ☞ Např. $[n=\zeta(x)x].n \hookrightarrow [n=\zeta(x)x]$
- ☞ Redukce modifikace metody
 - ☞ Např. $[m=\zeta(y)[]].m \Leftarrow \zeta(x)x \hookrightarrow [m=\zeta(x)x]$

Redukce – Příklady

- ☞ $[l = \varsigma(x)[]]$
- ☞ $[m_1 = \varsigma(z)[], m_2 = \varsigma(y)y.m_1].m_1 \Leftarrow \varsigma(x)[k = \varsigma(x)[]]$
 - ↪ $[m_1 = \varsigma(x)[k = \varsigma(x)[]], m_2 = \varsigma(y)y.m_1]$
- ☞ $[m = \varsigma(y)[], n = \varsigma(x)x.m].n$
 - ↪ $[m = \varsigma(y)[], n = \varsigma(x)x.m].m$
 - ↪ $[]$



Ostatní nejen objektově-orientované koncepty



Cíl přednášky

- ➊ Výjimky
- ➋ Šablony

Výjimky (*Exceptions*)

- ☞ Výjimka = softwarově/hardwarově detekovatelný neočekávaný/chybový stav, který je typicky třeba ošetřit
- ☞ Návrh práce s výjimkami?
 - Jak *vyvolat* výjimku?
 - Jak a kde specifikovat *ošetřující kód*?
 - Vazba vyvolané výjimky a ošetřujícího kódu? *Změna modelu výpočtu!*
 - Předávání informací do ošetřujícího kódu?
 - Pokračování/návrat k normálnímu modelu výpočtu?

Návrh práce s výjimkami

Produkční kód:

```
...  
block {  
    ...  
    vyvolán → píkaz;  
    ...  
}
```

vazba výjimka → ošetření,
propagace?

Ošetřující kód:

when X

{

}

when Y

{

}



Běžné řešení

Reprezentace

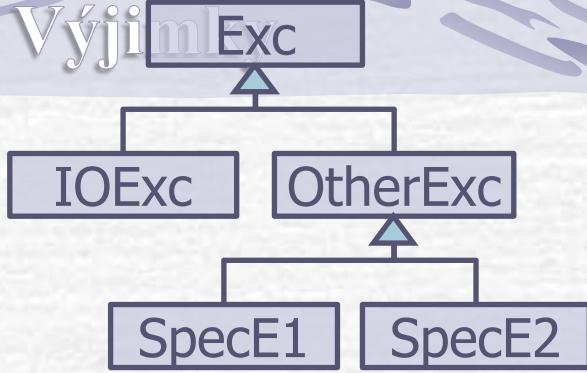
- Chybový kód
- Struktura/objekt/**instance třídy** obsahující informace o vzniklé chybě
 - využití třídní hierarchie pro kategorizaci

Běžné řešení dnes:

- Vyvolání výjimky: *throw/raise <výjimka>*
 - Nastane dočasná změna modelu výpočtu
- Ošetření výjimky: *try – catch/except – finally*
- Správa zdrojů: *using* (C#), *with* (Python 3), *try(...)* (Java 7)

Třídní hierarchie výjimek

- ☞ Typicky existence kořenové třídy hierarchie (*Exception*)
- ☞ Blok(y) *catch* specifikuje třídu(y), jejíž instance odchytává (včetně instancí podtříd)
 - Nutnost efektivní implementace testu třídy (nadtrídy) instance výjimky
- ☞ Neošetřená výjimka ⇒ propagace do obalujícího *try*
 - *v téže metodě/funkci*
 - *ve volající (nepřímo) metodě/funkci*
- ☞ Objekt výjimky nese informace potřebné pro její ošetření → možnost dislokace ošetření
 - neOO řešení: chybové kódy + globální proměnné



Příklad

Kontext o1.m1():

```

try:
    ...
    o2.m2()
    ...
catch SpecE1:
    ... handle error ...
catch OtherExc:
    ... handle error ...
...
  
```

Kontext o2.m2():

```

...
o3.m3()
...
if c:
    e = new E
    e.set(...)
    raise e
...
  
```

Kontext o3.m3():

```

try:
    ...
raise new .....(...)
...
catch IOExc as e:
    ... handle error ...
    raise //again
finally:
    ... do always ...
...
  
```

Generický polymorfismus a šablony

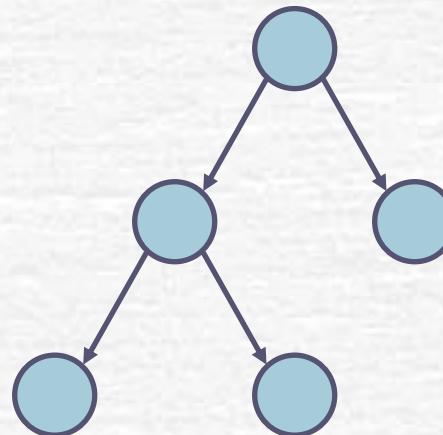
Generický polymorfismus

- Statický (typový parametr = hodnotou je typ)
 - Šablony (*templates*) – textové, např. C++
 - Generiky (*generics*) – řízení implicitního přetypování, např. Java
- Dynamický – polymorfní metody (pozdní vazba)
- Ad hoc – přetěžování operátorů

Motivační příklad: Binární strom

Uzly mohou nést data:

- čísla
- řetězce
- objekty, ...



Jak zajistíme?

- Homogenita uzelů stromu (dat jeho uzelů)

OO Řešení bez šablon: Návrh

- Pro uzly společná abstraktní nadtířida
- Pro manipulaci se stromem bez přístupu k datům – OK
- Např.
 - `subclass Item of Object is ...;`
 - `Item * i = (Item) tree.getObject(); ...`
- Nevýhody:
 - Potřeba společné (abstraktní) nadtířidy pro uzly
 - API umožňuje porušení homogeneity

Šablonové řešení

- ✓ **Šablona** = blok kódu (třída, funkce, metoda, ...) využívající „falešný typ“, který bude později nahrazen konkrétním typem, až bude znám (rozgenerován)
- ✓ **Typová proměnná** – hodnotou je datový typ
 - Používá se pro parametrizaci šablony
- ✓ Rozgenerování/instanciacie šablony až při použití (dosazen konkrétní typ za typovou proměnnou)
 - Šablony v C++: rozgenerování před komplikací („textově“)
 - Generiky v Java 5.0/C# 2.0: viz řešení bez šablon s doplněním implicitních konverzí za běhu

Případová studie: C++ (I)

```
template<class T>
class Node {
public:
    Node(Node<T> *the_parent = NULL );
    T item;
    int number_of_items;
    Node<T> *parent;
    Node<T> *left_child;
    Node<T> *right_child;
};
```

Případová studie: C++ (II)

```
template<class T> class BTREE {  
public:  
    BTREE();  
    ~BTREE();  
    void Insert(T item);  
    ...  
private:  
    bool r, l; int node_count;  
    Node<T> *root;  
    void Insert(T item, Node<T> *tree, Node<T> *parent);  
    void List_PreOrder(Node<T> *tree, Node<T> *parent);  
    ...  
// Instanciace šablony:  
    BTREE<int> * treeOfIntegers = new BTREE<int>();
```

K zapamatování

- ☛ Tok řízení při vyvolání výjimky
- ☛ Reprezentace výjimky jako objektu
- ☛ Generický polymorfismus

Cvičení

- Najděte na internetu nějaký článek o implementaci výjimek a prostudujte jej. Zkuste nastudovat fungování výjimek v jiném jazyce než C++ a Java.
- Srovnejte možnosti práce s generikami v Javě a šablonami v C++. Identifikujte výhody a nevýhody každého přístupu.
- Vyzkoušejte si sestavit několik správných ζ -výrazů v ζ -kalkulu.