KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Michal Krupka
krupka.inf.upol.cz
michal.krupka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 3 ⋄ poznámky k přednášce

1. Objektově orientované programování jako paradigma

verze z 18. září 2023

1 Přehled základních programovacích paradigmat

Nacházíme se v polovině čtyřsemestrálního kurzu o paradigmatech programování. To je vhodná chvíle na podrobnější vysvětlení, co paradigmata programování jsou a popis těch základních, se kterými jste se už setkali, i nástin dalších, se kterými se setkáte později. Poté přikročíme k úvodním informacím o důležitém paradigmatu, kterým se budeme zabývat tento semestr: paradigmatu objektovému.

Paradigmata programování jsou obvykle charakterizována jako programovací **styly**. V tomto úvodu se pokusíme o přesnější popis. Kdo se chce s tématem seznámit podrobněji, bude tento stručný úvod konfrontovat s vlastními zkušenostmi a podívá se také do jiných zdrojů.

Programovací jazyk je prostředkem komunikace s počítačem. Programovací jazyky jsou přizpůsobeny lidskému uvažování a každý programovací jazyk vyžaduje více nebo méně náročný překlad do strojového kódu — jazyka, kterému rozumí zase počítač.

Při vytváření programu programátor potřebuje pracovat s nějakou představou, modelem toho, co to je počítačový program, co se děje, když je na počítači vykonáván, a jakým způsobem v jeho zdrojovém kódu počítači říkáme, co má dělat. Tento model nesouhlasí s reálným průběhem výpočtu na počítači (kterým je vykonávání jednoduchých instrukcí procesorem); jeho smyslem je usnadnit programátorovi uvažování o programu způsobem, který je pro něj přirozený a který se hodí k popisu postupu řešení daného problému. Takový model, více nebo méně formálně popsaný, a vtělený do programovacího jazyka, se nazývá **paradigma programování**.

Například: k řešení problému jednoduchým algoritmem se hodí představa programu jako posloupnosti příkazů s podmínkami, cykly, prací s proměnnými včetně jejich nastavování a jednoduchou rekurzí (procedurální paradigma). Simulaci reality nebo třeba vektorovou grafiku budeme spíše modelovat sadou samostatných objektů, které mají atributy a kód řídící jejich chování, a mezi sebou komunikují (objektové paradigma). Algebraický nebo symbolický výpočet se hodí popsat jako posloupnost aplikací matematických funkcí (funkcionální paradigma). Požadované informace z databáze zase získáme prostým popisem požadovaného výsledku, aniž bychom vůbec stanovovali, jak k němu má program dojít (dotazovací paradigma

relačních databází).

K popisu programu podle různých paradigmat lze samozřejmě alespoň částečně použít libovolný programovací jazyk. K úspěšnému vytvoření programu je ale nezbytné, aby bylo možné se v použitém programovacím jazyce snadno podle principů zvoleného paradigmatu vyjadřovat. Programovací jazyky nabízejí různé jazykové konstrukce, které to umožňují.

Programovací jazyky jsou obvykle zaměřeny na jedno konkrétní paradigma. Proto běžně hovoříme o funkcionálních, objektově orientovaných a dalších jazycích. V praxi se obvykle používají jazyky, ve kterých lze kromě jejich hlavního částečně nebo úplně používat i jiná paradigmata. Takovým jazykem je i Common Lisp, což jej činí vhodným k použití v tomto předmětu. To posiluje i fakt, že díky pokročilé práci s makry do něj lze do jisté míry dodávat další paradigmata, která v něm původně nebyla (tuto jeho vlastnost je ostatně možno také chápat jako další zvláštní paradigma).

Používaná paradigmata programování se obvykle dělí do dvou skupin: na paradigmata **imperativní** a **deklarativní**.

Imperativní paradigmata

Program v imperativním paradigmatu je v každý moment svého vykonávání dán svým **stavem**. Tím je především souhrn všech proměnných (lokálních a globálních) a jejich hodnot. U jazyků nejnižší úrovně se místo proměnných pracuje přímo s registry procesoru a obsahem paměti. Program pracuje tak (když pomineme externí vedlejší efekt, jako je např. tiskový výstup), že hodnoty zjišťuje a nastavuje. Zdrojový kód se skládá z **příkazů** na změnu stavu a příkazů řídících běh programu. Imperativně napsaným programem počítači sdělujeme, jakým postupem má dojít k požadovanému výsledku. Každý vyšší programovací jazyk ovšem kromě imperativních prvků obsahuje i prvky deklarativní (aritmetické operace, volání funkcí...), ty ale nikdy nepřevažují a nebývají tak chápány.

Čisté imperativní paradigma. Program je založen v podstatě jen na výše uvedených základních principech imperativních paradigmat. Obvykle také může obsahovat **podprogramy**, což je část kódu, na kterou může program přejít z různých míst. Aby se po ukončení podprogramu řízení vrátilo na správnou adresu, je třeba si ji při volání podprogramu zapamatovat. Vnořené volání podprogramů včetně rekurzivního je umožněno pomocným zásobníkem. Základní jazyky: strojové kódy procesorů, assemblery (ty navíc obvykle obsahují makra), základní verze jazyka BASIC.

Zásobníkové paradigma. Příbuzné předchozímu, v programu ovšem centrální roli hraje zásobník nebo zásobníky; místo práce s proměnnými (či jinam uloženými hodnotami) se pracuje s hodnotami uloženými na zásobníku. Jazyky obvykle používají postfixovou notaci. Se zásobníkovým paradigmatem jsme se setkali minulý semestr. Základní jazyk: FORTH. Dále: PostScript, runtimy Javy, Pythonu, .NETu.

Procedurální paradigma. Nepoužívá zásobník explicitně a nezatěžuje programátora režií kolem volání podprogramů. Těm se říká **procedury**, je u nich navíc možné volání s parametry a návratové hodnoty — procedurám s návratovými hodnotami se také říká **funkce**. Základní jazyky: Algol, Pascal, C.

Strukturované paradigma. Varianta procedurálního s vynucenou větší disciplínou psaní kódu: program se dělí na tzv. **struktury**, což jsou bloky kódu s jediným začátkem a koncem. Z bloků (včetně cyklů) je zakázán předčasný výskok, instrukce `GOTO` neexistuje. Ve volnější verzi používáno ve většině moderních jazyků. Základní jazyk: Pascal.

Objektové paradigma. Běžící program se skládá ze samostatných a nezávislých jednotek zvaných **objekty**. Objekty v sobě mohou sjednocovat jak roli **modulů**, tedy částí programu implementujících nějakou ucelenou funkčnost a poskytujících k ní rozhraní, tak **abstraktních datových struktur**, které za běhu programu vznikají a zanikají; a koncepčně (na rozdíl od obyčejných datových struktur) kromě dat obsahují i funkce, které s nimi pracují. S objekty pracujeme pomocí tzv. **zasílání zpráv**. Na přijetí zprávy reaguje objekt autonomně spuštěním svého kódu. Objekty bývají definovány jako instance **tříd**, které jsou uspořádány do hierarchie podle principu **dědičnosti**. Většina moderních programovacích jazyků objektové paradigma používá jako své hlavní, ale stále více zahrnuje i další paradigmata. Takovým jazykům říkáme **objektově orientované**. Jazyky: Simula 67 (první objektově orientovaný), SmallTalk (první čistě objektový). Objektově orientované: C++, Java, C#, Python. V dalších jazycích lze objektové paradigma použít, ale nejsou na něm postaveny (Common Lisp).

Prototypové objektové paradigma. Objekty se nevytvářejí jako instance tříd (ty v tomto paradigmatu neexistují), ale jako kopie nebo klony (mělké kopie) jiných objektů sloužících jako **prototypy**. Dědičnost je obvykle založena na vztahu potomek–předek mezi objektem a jeho prototypem. Jazyky: Self, JavaScript.

Paralelní programování. V paralelním programu je kód vykonáván (zdánlivě nebo skutečně) souběžně na více procesorech současně. Tím vznikají pro programátora nové problémy, neznámé z běžných, sekvenčních programů: obecně se nemůžeme spolehnout, že proměnná bude mít hodnotu, kterou jsme do ní uložili, protože může být změněna jiným procesem. Podle našeho vymezení nejde o programovací paradigma; paralelní program lze vytvořit v libovolném paradigmatu. Některé jazyky ale obsahují speciální konstrukce pro paralelní programování (např. paralelní cyklus) a pro paralelní programování existují i specializované jazyky. V těchto případech lze o paradigmatu hovořit.

Deklarativní paradigmata

Na rozdíl od imperativních paradigmat, u kterých program chápeme jako souhrn **příkazů**, kterými stroji říkáme, jak vypočítat žádaný výsledek, ideou deklarativních paradigmat je zdrojovým kódem charakterizovat vlastnosti výsledku a nechat k němu program dojít vlastním způsobem. Místo **jak** k výsledku dojít tedy říkáme pouze, **co** má výsledkem být.

U tří paradigmat dále uvádíme i jednoduché příklady.

Funkcionální paradigma. Zdrojový kód programu lze chápat jako kompozici funkcí bez vedlejšího efektu (tzv. čisté funkce, pure functions). Funkcionální program nepracuje vůbec se stavem. Opakovaného výpočtu se dosahuje pomocí rekurze a používáním funkcí vyššího řádu. Za běhu programu lze vytvářet nové funkce, obvykle anonymní. Funkcionální programovací jazyky často používají líné vyhodnocování, parciální aplikaci funkcí, currying, pattern matching. Mívají pevné teoretické základy (λ-kalkul) a používají pokročilé teoretické koncepty (monády). Dobře napsaný funkcionální program (ve vhodném jazyce) má deklarativní charakter. Např. popisujeme podmínku, kdy leží prvek v seznamu, nikoli postup, jak to zjistit:

```
(defun in? (elem list)
  (and (not (null list))
       (or (eql elem (first list))
           (in? elem (rest list)))))
```

(v jazyce s pattern matchingem by definice vypadala ještě lépe). Jazyky: Haskell (čistě funkcionální), jazyky rodiny Lispu (založené na funkcionálním paradigmatu).

Logické paradigma. Program je soubor známých informací o nějakém problému ve tvaru **faktů** nebo **pravidel** (která jsou např. v Prologu tvaru *`závěr :- předpoklady`*). Výpočet zjišťuje (způsobem, který obvykle nemusíme znát), za jakých podmínek lze zadanou hypotézu (**dotaz**) odvodit jako důsledek faktů a pravidel. Teoretickým základem logického paradigmatu je predikátová logika. Příklad: následujícím faktem a pravidlem lze definovat, kdy může být hodnota prvkem seznamu (seznam píšeme ve tvaru `[`*`první_prvek`*`|`*`zbytek`*`]`):

```
member(X, [X|_]).
member(X, [_|Tail]) :- member(X, Tail).
```

Vhodnými dotazy lze zjistit, zda leží prvek v seznamu, jak musí prvek vypadat, aby ležel v seznamu, ale také třeba, jak musí vypadat seznam, aby obsahoval daný prvek. Jazyk: Prolog.

Dotazovací paradigma relačních databází. Před vznikem relačních databází musel programátor s databázemi pracovat pomocí obecných imperativních programovacích jazyků. Relační databáze jsou postaveny na přesné matematické teorii (databáze je v podstatě soustava matematických relací, se kterými se pracuje pomocí relační algebry), což vede k tomu, že požadovaný výsledek lze charakterizovat podobně, jako se v matematice zapisují množiny, aniž by se muselo systému diktovat, jak jej má nalézt. Jazyk, pomocí kterého s databází pracujeme, pak dokonce nemusí být výpočetně úplný (program se nemůže zacyklit). Následující program v jazyce SQL pracuje se seznamem (**relací**) `person` n-tic obsahujících položky `age` a `name`. Jako výsledek vrátí seznam (opět relaci) čísel splňujících zadanou podmínku.

```
SELECT age
FROM person
WHERE name = 'Adam';
```

Jazyk: SQL.

Další rysy programovacích jazyků

Kromě základních paradigmat u programovacích jazyků rozlišujeme i další charakteristiky: líné vyhodnocování, možnost vytvářet lexikální uzávěry, automatickou správu paměti, slabý nebo silný typový systém, statické nebo dynamické typování. Jazyky (přesněji jejich implementace) mohou být interpretované nebo kompilované, moderně se používají i další, hybridní přístupy. Známe také jazyky dynamické. Podrobnosti o těchto a dalších rysech programovacích jazyků si můžete zjistit sami.

2 Objekty

V objektově orientovaném programování je běžící program složen ze samostatných a nezávislých jednotek, zvaných *objekty*.[1] Každý objekt je zodpovědný za určitou část činnosti programu. Koná ji na základě pokynů, které mu udělujeme zasíláním *zpráv*. V reakci na přijetí zprávy objekt vykoná požadovanou činnost. Ta se obvykle týká jeho vlastního obsahu.

[1] Prvním objektově orientovaným jazykem byl jazyk Simula 67, vytvořený v 60. letech minulého století v Norském výpočetním středisku v Oslu. Velmi důležitým objektovým programovacím jazykem je SmallTalk, který vznikl začátkem 70. let v Xerox Palo Alto Research Center. Objektový systém Common Lispu (Common Lisp Object System, CLOS), kterému se budeme věnovat v tomto textu, vznikl ze starších objektových systémů CommonLOOPS a MIT Flavors. Common Lisp je historicky prvním standardizovaným programovacím jazykem používajícím objektově orientovaný přístup (ANSI standard). Jedním z nejvýznamnějších jazyků používajících objektové paradigma je jazyk C++, který významně napomohl rozšíření objektového programování do praxe. Většina běžných moderních programovacích jazyků dnes používá objektový přístup: Java, C#, Python, Swift, JavaScript (ten ovšem s jedním podstatným rozdílem proti všem předchozím uvedeným, ke kterému se dostaneme ve druhé části semestru).

Zopakujme znovu dvě základní charakteristiky objektů: *samostatnost* a *nezávislost*.

Samostatnost a nezávislost objektů

Samostatnost objektů znamená, že jim stačí říct, *co* mají dělat, není nutné specifikovat *jak*. Objekt by měl být pro programátora, který ho používá, co nejjednodušší k použití, měl by od něj vyžadovat co nejmenší znalosti a odstínit ho od nepodstatných technických detailů řešení problému.

Nezávislost Objekt je tím nezávislejší, čím méně ke své práci vyžaduje zajištění vnějších podmínek. Nezávislost objektů umožňuje jejich *znovupoužitelnost*, neboli zachování funkčnosti po přenesení do jiného prostředí (programu).

Tyto dvě charakteristiky si dobře zapamatujte. V budoucnu je můžeme používat k posuzování kvality napsaného programu.

Zprávy zasílané objektům jsou identifikovány jménem. Součástí zprávy mohou být i argumenty, podobně jako u volání funkce. Výsledkem zaslání zprávy může být (podobně jako u volání funkce) vrácená hodnota.

Na přijetí zprávy objekt reaguje tak, že spustí kód, který vykoná činnost, jež je po něm zprávou požadována. Tomuto kódu se říká *metoda*. Metoda je zvláštní druh funkce, má jméno a parametry. Každý objekt může mít k dispozici více metod, podle toho, které zprávy může přijímat. Po přijetí zprávy se spustí metoda stejného jména, jaké má přijímaná zpráva, s těmi argumenty, se kterými byla zpráva poslána. Metody mají stejně jako funkce návratovou hodnotu (hodnoty). Ta je nakonec výsledkem zaslání zprávy.

Metody objektu chápeme jako kód, který k němu patří (tuhle věc ještě za chvíli upřesníme); různé objekty mohou k obsluze téže zprávy používat různé metody.

Objekty lze chápat jako abstraktní datové struktury s tím dodatkem, že funkčnost objektů (metody) se chápe jako jejich součást ve smyslu principu samostatnosti uvedeném před chvílí.

Data uložená v objektu se nazývají jeho *vnitřní stav*. V reakci na přijetí zprávy může objekt svůj vnitřní stav změnit. To je kromě vrácení hodnoty další efekt, který může zaslání zprávy objektu mít.

Data v objektech jsou (podobně jako u struktur v jazyce C) rozdělena do pojmenovaných položek, kterým budeme říkat *sloty* (v jiných jazycích se často používá pojem *atribut*). Z pohledu uživatele objektu je ale jedno, jak je vnitřní stav v objektu reprezentován, protože uživatel s objektem komunikuje pouze zasíláním zpráv. Nezajímá ho, jak je napsaná metoda, která se v reakci na přijetí zprávy spustí, ani jak a s jakými daty uvnitř objektu pracuje.

Bariéra mezi vnitřním stavem objektu a jeho rozhraním

Při psaní programu musíme důsledně odlišovat mezi autorem metod a vnitřní stavby objektu a *uživatelem objektu*, tj. programátorem, který objekt ve svém programu používá, a to bez ohledu na to, že to může být tentýž člověk. Uživatel objektu nemusí znát informace o implementaci objektu a nemá je používat. Neměl by také být nucen dělat činnosti, které by měl objekt zvládnout sám.

Tento důležitý princip rozebereme podrobně později.

3 Třídy

Podobně jako jiné hodnoty, i objekty mohou být různých typů. V objektově orientovaném programování se pro základní typy objektů používá pojem *třídy*. Na tomto místě uvedeme zjednodušenou definici třídy, kterou v dalších částech rozšíříme.

Aby dva objekty patřily téže třídě, musí splňovat tyto podmínky:

1. musí obsahovat stejnou sadu slotů, tedy stejný počet slotů stejných názvů (hodnoty těchto slotů však mohou být různé),
2. musí obsahovat stejné metody.

Definice třídy ve zdrojovém textu programu tyto dva údaje (kromě názvu třídy) uvádí. Třídu lze tedy chápat, jako popis objektu: obsahuje jednak seznam názvů jeho slotů a jednak definici všech jeho metod. Při běhu programu pak třída slouží jako předloha k vytváření nových objektů. A konečně, jak už jsem napsal nahoře, třída se také chápe jako datový typ, tedy množina objektů.

Objekt, který patří třídě, se nazývá její *instancí*. Chceme-li v programu vytvořit objekt, musíme mít pro něj definovánu třídu a v programu objekt vytvořit jako její instanci.

4 Třídy a instance v Common Lispu

V Common Lispu, přesněji v jeho objektovém systému CLOS, se nové třídy definují pomocí makra `defclass`, které specifikuje seznam slotů třídy, a pomocí makra `defmethod`, které slouží k definici metod instancí třídy.

Nové objekty se vytvářejí pomocí funkce `make-instance`. Ke čtení hodnoty slotu objektu slouží funkce s názvem `slot-value`. Symbol `slot-value` také definuje **místo**.

Zprávy se v Common Lispu objektům zasílají pomocí stejné syntaxe, jakou se v tomto jazyce volají funkce.

Makro `defclass`

Zjednodušená syntax makra `defclass` je následující:

```
(defclass name () slots)

  name: symbol (nevyhodnocuje se)
  slots: seznam symbolů (nevyhodnocuje se)
```

Symbol *name* je název nově definované třídy, symboly ze seznamu *slots* jsou názvy slotů této třídy.

Prázdný seznam za symbolem *name* je součástí zjednodušené syntaxe. V dalších kapitolách, až se dozvíme více o třídách, ukážeme, co lze použít místo něj.

Příklad: třída `point`

Definice třídy `point`, jejíž instance by obsahovaly dva sloty s názvy `x` a `y`, by vypadala takto:

```
(defclass point ()
  (x y))
```

Funkce `make-instance`

Definovali-li jsme novou třídu (zatím bez metod, k jejichž definici se dostaneme vzápětí), měli bychom se naučit vytvářet její instance. V Common Lispu k tomu používáme funkci `make-instance`, jejíž zjednodušená syntax je tato:

```
(make-instance class-name)

  class-name: symbol
```

Funkce `make-instance` vytvoří a vrátí novou instanci třídy, jejíž jméno najde ve svém prvním parametru. Všechny sloty nově vytvořeného objektu jsou neinicializované a každý pokus získat jejich hodnotu (funkcí `slot-value`) skončí chybou.

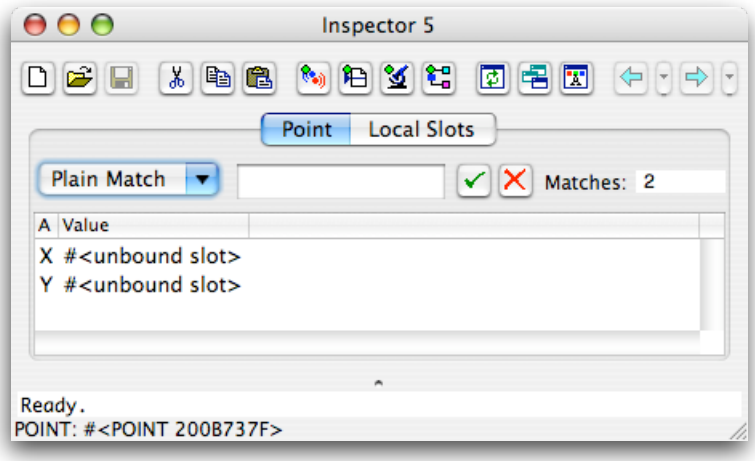
Příklad: vytváření a prohlížení instancí

Vytvoření nové instance třídy `point` z předchozího příkladu:

```
CL-USER 2 > (make-instance 'point)
#<POINT 200DC0D7>
```

Pokud není jasné, proč jsme ve výrazu `(make-instance 'point)` symbol `point` kvotovali, je třeba si uvědomit, že `make-instance` je funkce a zopakovat si základy vyhodnocovacího procesu v Common Lispu.

Výsledek volání není příliš čitelný, v prostředí LispWorks si jej ale můžeme prohlédnout v inspektoru (Obr. 1).



Obrázek 1: Neinicializovaná instance třídy `point` v inspektoru

Text `#<unbound slot>` u názvů jednotlivých slotů znamená, že sloty jsou neinicializované. Můžeme jim ale pomocí prostředí LispWorks zkusit nastavit hodnotu. Klikneme-li na některý ze zobrazených slotů pravým tlačítkem, můžeme si v objevivší se nabídce vybrat volbu "Slots->Set..." tak, jak je znázorněno na Obrázku 2 a novou hodnotu slotu nastavit.

Funkce `slot-value`

K programovému čtení hodnot slotů slouží funkce `slot-value`, k jejich nastavování symbol `slot-value` v kombinaci s makrem `setf`. Syntax je následující:

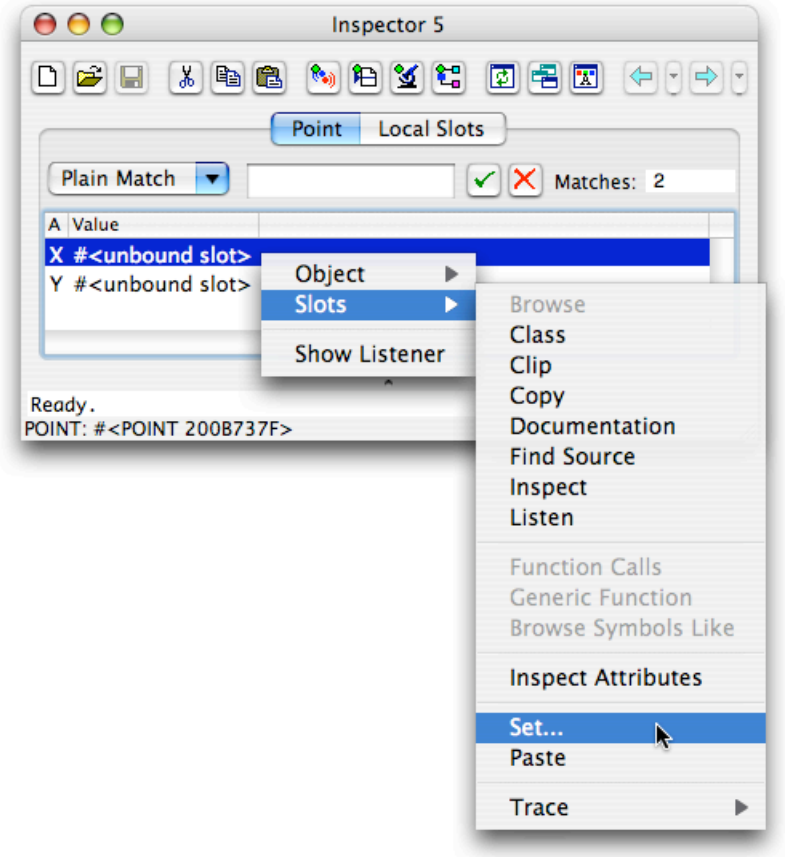
`(slot-value` *`object slot-name`*`)`

`object:` objekt
`slot-name:` symbol

Příklad: práce s funkcí `slot-value`

Vytvořme instanci třídy `point` a uložme ji do proměnné `pt`:

```
CL-USER 2 > (setf pt (make-instance 'point))
#<POINT 216C0213>
```



Obrázek 2: Nastavování hodnoty slotu v inspektoru

Nyní zkusme získat hodnotu slotu `x` nově vytvořené instance:

```
CL-USER 3 > (slot-value pt 'x)

Error: The slot X is unbound in the object #<POINT 216C0213>
(an instance of class #<STANDARD-CLASS POINT 200972AB>).
  1 (continue) Try reading slot X again.
  2 Specify a value to use this time for slot X.
  3 Specify a value to set slot X to.
  4 (abort) Return to level 0.
  5 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :?
for other options
```

Vidíme, že došlo k chybě; slot `x` není v nově vytvořeném objektu inicializován. Z chybového stavu se dostaneme napsáním `:a` a zkusíme hodnotu slotu nejprve nastavit:

```
CL-USER 4 : 1 > :a

CL-USER 5 > (setf (slot-value pt 'x) 10)
10
```

Nyní již funkce `slot-value` chybu nevyvolá a vrátí nastavenou hodnotu:

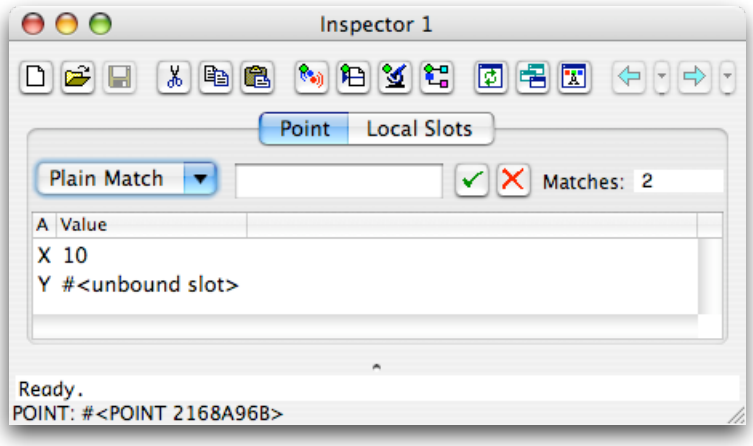
```
CL-USER 6 > (slot-value pt 'x)
10
```

Nové hodnoty slotů lze také ověřit pomocí inspektoru (Obr 3).

K práci se sloty ještě dodejme, že používáním funkce `slot-value` ke čtení a nastavování hodnoty slotů vlastně porušujeme výše uvedený princip, že pro komunikaci s objekty se používají výhradně zprávy. K tomuto problému se dostaneme v další kapitole.

Makro `defmethod` a posílání zpráv

Zbývá vysvětlit, jak se v Common Lispu definují metody objektů a jak se objektům posílají zprávy. Jak jsme již řekli, všechny instance jedné třídy mají stejnou sadu metod. Metody jsou zvláštní druh funkce, proto se definují podobně. V Common Lispu je k definici metod připraveno makro `defmethod`. Jeho syntaxe (ve zjednodušené podobě, jak ji uvádíme na tomto místě), je stejná jako u makra `defun` s tou výjimkou, že u prvního parametru je třeba specifikovat jeho třídu. Tím se metoda definuje pro všechny instance třídy.



Obrázek 3: Instance třídy `point` po změně hodnoty slotu `x`

```
(defmethod message ((object class) arg1 arg2 ...)
  expr1
  expr2
  ... )

  message: symbol
  object: symbol
  class: symbol
  argi: symbol
  expri: výraz
```

Symbol *class* určuje třídu, pro jejíž instance metodu definujeme, symbol *message* současně název nové metody i název zprávy, kterou tato metoda obsluhuje. Výrazy *expr1*, *expr2* atd. tvoří *tělo metody*, které stejně jako tělo funkce definuje kód, který se provádí, když je metoda spuštěna. Symbol *object* je v těle metody navázán na objekt, jemuž byla zpráva poslána, symboly *arg1*, *arg2*, atd. na další argumenty, se kterými byla zpráva zaslána.

Jak již bylo řečeno, syntax zasílání zprávy je stejná jako syntax volání funkce:

```
(message object arg1 arg2 ...)

  message: symbol
  object: výraz
  argi: výraz
```

Symbol *message* musí být názvem zprávy, kterou lze zaslat objektu vzniklému vyhodnocením výrazu *object*. Zpráva je objektu zaslána s argumenty, vzniklými vyhodnocením výrazů *arg1*, *arg2* atd. Stejně jako u volání funkce jsou výrazy *object*, *arg1*, *arg2* atd. vyhodnoceny postupně zleva doprava.

Zasílání zprávy v Common Lispu má stejnou syntax jako volání funkce a ve skutečnosti opravdu o volání funkce jde. Jedná se o funkce zvláštního druhu, kterým se říká *generické funkce*. Tento fakt je dobré mít na paměti, protože vývojové prostředí o generických funkcích někdy (třeba při hlášení chyb) mluví. Ke konci semestru se ke zvláštnostem objektového systému Common Lispu dostaneme.

V mnoha jiných programovacích jazycích by se výše uvedené zaslání zprávy zapsalo takto:

```
object.message(arg1 arg2 ...)
```

Příklad: polární souřadnice

Řekněme, že potřebujeme zjišťovat polární souřadnice bodů. Správný objektový způsob řešení této úlohy je definovat nové zprávy, které budeme bodům k získání těchto informací zasílat.

Definujme tedy nové metody pro třídu `point`: metodu `r`, která bude vracet vzdálenost bodu od počátku (první složku jeho polárních souřadnic), a metodu `phi`, která bude vracet odchylku spojnice bodu a počátku od osy x (tedy druhou složku polárních souřadnic bodu)

Metoda `r` počítá vzdálenost bodu od počátku pomocí Pythagorovy věty:

```
(defmethod r ((point point))
  (let ((x (slot-value point 'x))
        (y (slot-value point 'y)))
    (sqrt (+ (* x x) (* y y)))))
```

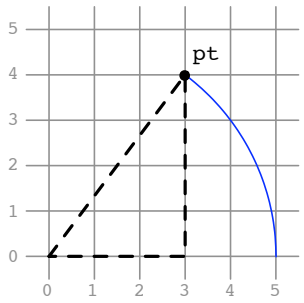
Pokud nechápete, co znamená `(point point)` v této definici, podívejte se znovu na syntax makra `defmethod`. Zjistíte, že první položkou tohoto seznamu je symbol, na nějž bude při vykonávání těla metody navázán příjemce zprávy, zatímco druhou položkou je název třídy, pro jejíž instance metodu definujeme. Že jsou oba tyto symboly stejné, nevadí, v Common Lispu mohou být názvy proměnných a tříd stejné.

Po zaslání zprávy `r` bodu bychom tedy měli obdržet jeho vzdálenost od počátku. Vytvořme si na zkoušku instanci třídy `point` a nastavme jí hodnoty slotů `x` a `y` na `3` a `4`:

```
CL-USER 8 > (setf pt (make-instance 'point))
#<POINT 200BC6A3>

CL-USER 9 > (setf (slot-value pt 'x) 3
                  (slot-value pt 'y) 4)
4
```

Vytvořený objekt reprezentuje geometrický bod, který je znázorněn na Obrázku 4.



Obrázek 4: Bod o souřadnicích $(3, 4)$

Nyní zkusme získat vzdálenost tohoto bodu od počátku zasláním zprávy `r` naší instanci (připomeňme, že zprávy se objektům zasílají stejnou syntaxí jakou se volají funkce, tedy výraz `(r pt)` znamená zaslání zprávy `r` objektu `pt`):

```
CL-USER 10 > (r pt)
5.0
```

To správně, jelikož $\sqrt{3^2 + 4^2} = 5$.

Podobně definujme metodu `phi` (pochopení vyžaduje trochu matematických znalostí: k výpočtu používáme komplexní čísla):

```
(defmethod phi ((point point))
  (phase (complex (slot-value point 'x)
                  (slot-value point 'y))))
```

Další zkouška:

```
CL-USER 11 > (phi pt)
0.9272952
```

Tangens tohoto úhlu by měl být roven 4/3 (viz Obrázek 4):

```
CL-USER 12 > (tan *)
1.3333333
```

Příklad: nastavování polárních souřadnic

Definujme ještě metody pro nastavení polárních souřadnic bodu. Na rozdíl od předchozích budou tyto metody vyžadovat zadání argumentů. Vzhledem k tomu, že každá z nich mění obě kartézské souřadnice bodu současně, bude užitečné napsat nejprve metodu pro současné nastavení obou polárních souřadnic.

```
(defmethod set-r-phi ((point point) r phi)
  (let ((complex (* (cis phi) r)))
    (setf (slot-value point 'x) (realpart complex)
          (slot-value point 'y) (imagpart complex)))
  point)
```

Metody `set-r` a `set-phi` tuto metodu využijí (přesněji řečeno, zprávu `set-r-phi` zasílají):

```
(defmethod set-r ((point point) value)
  (set-r-phi point value (phi point)))

(defmethod set-phi ((point point) value)
  (set-r-phi point (r point) value))
```

Metody `set-r-phi`, `set-r` a `set-phi` vracejí vždy jako výsledek parametr `point`. Tento přístup budeme volit ve všech metodách, které mění stav objektu: vždy budeme jako výsledek vracet měněný objekt. Důvodem je, aby šlo objektu měnit více hodnot v jednom výrazu:

```
(set-r (set-phi pt pi) 1)
```

Nyní můžeme instancím třídy `point` posílat zprávy `set-r-phi`, `set-r` a `set-phi` a měnit tak jejich polární souřadnice. Vyzkoušejme to tak, že našemu bodu `pt` pošleme zprávu `set-phi` s argumentem `0`. Tím bychom měli zachovat jeho vzdálenost od počátku, ale odchylka od osy x by měla být nulová.

Zaslání zprávy `set-phi` s argumentem `0`:

```
CL-USER 13 > (set-phi pt 0)
#<POINT 200BC6A3>
```

Test polohy transformovaného bodu:

```
CL-USER 14 > (slot-value pt 'x)
5.0

CL-USER 15 > (slot-value pt 'y)
0.0
```

Výsledek je tedy podle očekávání (nová poloha bodu je na druhém konci modrého oblouku na Obrázku 4).

5 Inicializace slotů nových instancí

Ukažme si ještě jednu možnost makra `defclass`. V předchozích odstavcích jsme si všimli, že když vytvoříme novou instanci třídy, jsou všechny její sloty neinicializované a při pokusu o získání jejich hodnoty před jejím nastavením dojde k chybě. To se někdy nemusí hodit. Proto makro `defclass` stanovuje možnost, jak specifikovat počáteční hodnotu slotů nově vytvářené instance.

V obecnější podobě makra `defclass` je jeho syntax následující:

```
(defclass name () slots)

  name: symbol (nevyhodnocuje se)
  slots: seznam (nevyhodnocuje se)
```

Prvky seznamu *slots* mohou být symboly nebo seznamy. Je-li prvkem symbol, je jeho význam takový, jak již bylo řečeno, tedy specifikuje název slotu instancí třídy, který není při vzniku nové instance inicializován. Je-li prvkem tohoto seznamu seznam, musí být jeho tvar následující:

```
(slot-name :initform expr)

  slot-name: symbol
  expr: výraz
```

V tomto případě specifikuje symbol *slot-name* název definovaného slotu. Výraz *expr* je vyhodnocen pokaždé při vytváření nové instance třídy a jeho hodnota je do příslušného slotu instance uložena.

Sloty nově vytvořených instancí je dobré vždy hned inicializovat. Kromě uvedeného způsobu se později dozvíme další, vhodný na složitější situace.

Konzistence nově vytvořených instancí

Nově vytvořené instance by měly být rovnou v konzistentním stavu, tj. měly by splňovat všechny předpoklady kladené na instance dané třídy.

Toto pravidlo je užitečné z pohledu uživatele. K vytvoření nové funkční instance třídy mu stačí zavolat funkci `make-instance`. Nemusí provádět žádné další inicializace. Pro autora třídy znamená dodržení pravidla víc práce, ale tak to má být: programátor se má vždy snažit usnadňovat uživateli práci i za tu cenu, že jemu to práci přidělá.

Příklad: třída `point` s inicializací slotů

Základní předpoklad o bodech je bezesporu ten, že mají nějaké kartézské souřadnice. Proto upravíme definici třídy `point` tak, aby byly sloty `x` a `y` nových instancí inicializovány na hodnotu `0`:

```
(defclass point ()
  ((x :initform 0)
   (y :initform 0)))
```

Jak můžeme snadno zkusit, sloty nových instancí jsou nyní inicializovány:

```
CL-USER 1 > (setf pt (make-instance 'point))
#<POINT 20095117>

CL-USER 2 > (list (slot-value pt 'x) (slot-value pt 'y))
(0 0)
```

Příklad: třída `circle`

Nyní definujeme další třídu, jejíž instance budou reprezentovat geometrické útvary. Bude to třída `circle`. Jak známo, geometrie kruhu je určena jeho středem a poloměrem. Proto budou mít instance této třídy dva sloty. Slot `center`, který bude obsahovat instanci třídy `point` a slot `radius`, který bude obsahovat číslo. Každý z těchto slotů bude při vytvoření nové instance automaticky inicializován. Naším předpokladem kladeným na všechny kruhy totiž je, že jejich středem má být bod a poloměrem číslo.

```
(defclass circle ()
  ((center :initform (make-instance 'point))
   (radius :initform 1)))
```

Teď již necháme na čtenáři, aby si sám zkusil vytvořit novou instanci této třídy a prohlédl její sloty.

V následujícím příkladu ukážeme ještě několik chybových hlášení, se kterými se můžeme ve vývojovém prostředí LispWorks setkat.

Příklad: chybová hlášení

Pokud pošleme zprávu objektu, který pro ni nemá definovánu metodu (obsluhu této zprávy), dojde k chybě. Můžeme si to ukázat tak, že pošleme zprávu `phi` instanci třídy `circle`:

```
CL-USER 3 > (phi (make-instance 'circle))
```

```
Error: No applicable methods for #<STANDARD-GENERIC-FUNCTION PHI
21694CFA> with args (#<CIRCLE 216C3CF3>)
  1 (continue) Call #<STANDARD-GENERIC-FUNCTION PHI 21694CFA> again
  2 (abort) Return to level 0.
  3 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :? for
other options
```

V tomto hlášení o chybě je třeba všimnout si hlavně textu „No applicable methods“, který znamená, že jsme posílali zprávu objektu, který pro ni nemá definovanou obsluhu (metodu).

Vzhledem k tomu, že syntax zasílání zpráv je v Common Lispu stejná jako syntax volání funkce či aplikace jiného operátoru, nemohou se zprávy jmenovat stejně jako funkce, makra nebo speciální operátory. Proto následující definice vyvolá chybu (`set` je funkce Common Lispu):

```
CL-USER 5 > (defmethod set ((point point) coord value)
              (cond ((eql coord 'x) (set-x point value))
                    ((eql coord 'y) (set-y point value))))

Error: SET is defined as an ordinary function #<Function SET
202D54A2>
  1 (continue) Discard existing definition and create generic
function
  2 (abort) Return to level 0.
  3 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :? for
other options
```

Pokud bychom se pokusili poslat objektu zprávu, pro niž jsme nedefinovali metodu pro žádnou třídu, Common Lisp vůbec nepochopí, že se snažíme poslat zprávu, a bude volání interpretovat jako použití neexistujícího operátoru:

```
CL-USER 7 > (fí (make-instance 'circle))

Error: Undefined operator FÍ in form (FÍ (MAKE-INSTANCE (QUOTE
CIRCLE))).
  1 (continue) Try invoking FÍ again.
  2 Return some values from the form (FÍ (MAKE-INSTANCE (QUOTE
CIRCLE))).
  3 Try invoking something other than FÍ with the same arguments.
  4 Set the symbol-function of FÍ to another function.
```

```
  5 Set the macro-function of FÍ to another function.
  6 (abort) Return to level 0.
  7 Return to top loop level 0.

Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template or :? for
other options.
```

Otázky a úkoly na cvičení

1. Vysvětlete základní pojmy objektově orientovaného programování: objekt, třída, instance, zpráva, metoda, slot.

2. Definujte třídu `triangle`, jejíž instance budou představovat trojúhelníky v rovině. Trojúhelníky budou obsahovat sloty `vertex-a`, `vertex-b`, `vertex-c` uchovávající vrcholy trojúhelníka jako body.

3. Definujte následující metody pro třídu `triangle`: Metoda `vertices` vrátí seznam vrcholů, metoda `perimeter` vrátí délku obvodu a metoda `right--triangle-p` zjistí, zda je trojúhelník pravoúhlý (zde musíte použít určitou malou toleranci).

4. U metody `perimeter` z předchozího příkladu jste si zřejmě napsali nástroj na výpočet délky úsečky o zadaných koncových bodech. Napsali jste ho jako funkci? Nebo metodu? Proč?

5. Elipsu v rovině lze zadat pomocí dvou ohnisek a délky hlavní poloosy. Druhou poloosu už lze vždy dopočítat. Definujte třídu `ellipse`, která bude obsahovat sloty `focal-point-1`, `focal-point-2` a `major-semiaxis`. Sloty slouží k uchovávání dvou bodů, které jsou ohnisky elipsy, a čísla, které je délkou hlavní poloosy. Volitelně můžete napsat metody pro přístup k těmto slotům (viz také následující příklad). Dále můžete ve třídě definovat další slot (sloty) na uchovávání potřebných informací o elipse.

6. Definujte pro třídu `ellipse` metody `major-semiaxis` a `minor-semiaxis`, které vrátí délku hlavní (to je vždy ta delší) a vedlejší poloosy

7. Napište metodu `to-ellipse` třídy `circle`, která vrátí elipsu stejného tvaru, jako kruh, jenž je příjemcem zprávy.