Programování v jazyce Ruby

Martin Šín
<martin.sin@zshk.cz>
version 1.0, Duben 2008

Table of Contents

Úvod

V následujících několika článcích se budeme blíže seznamovat s programovacím jazykem Ruby. Cílem tohoto seriálu nebude udělat z vás hotové programátory, ani rozšířit vaši sbírku zářezů na opasku udávající kolik programovacích jazyků znáte. Články jsou jednoduše určeny pro začínající programátory, ať už se tím budou později zabývat na profesionální či amatérské úrovni. Seriál mohou použít také učitelé na všech typech škol k rozšíření znalostí jim svěřených studentů ať už jako hlavní či vedlejší programovací jazyk.

V textu pak nehledejte žádná moudra zkušeného programátora, ani pění chvály na každou novou vlastnost či funkci. Při jeho vytváření jsem vycházel z dostupné dokumentace k programovacímu jazyku Ruby, konkrétně pak z textu Learn to Program, který vytvořil Chris Pine. Vlastní struktura pak bude připadat spíš učebnici a tak se bude stejně jako v originále každý článek zaměřovat na jednu konkrétní část jazyka, zároveň se bude předpokládat znalost předchozích dílů, které na sebe logicky navazují. Jiné znalosti nejsou třeba. Námi vytvořené programy budou určeny pro textovou konzoli, grafickými programy se zabývat nebudeme.

Co to je Ruby a odkud přišel? Krátké povídání naleznete v sekci Programovací jazyky Lukáše Faltýnka.

Protože nemám rád zbytečné otálení, pojďme se nyní rovnou podívat jak se v Ruby pracuje, dnešní kapitola se bude nazývat čísla a ukáže nám jak se provádí základní matematické operace.

Čísla

Ať už jste někdy programovali nebo ne, pro psaní programu je potřeba nějaký textový editor. Tímto editorem se nemyslí editor typu OpenOffice, ale

opravdový textový editor, tím může být např. vim, emacs, mousepad, gedit, kedit, nebo třeba nedávno zmiňovaný editor Geany.

Pokud berete programování opravdu vážně, pak věnujte výběru textového editoru určitou pozornost. Pokud se s ním naučíte opravdu dobře pracovat, pak vám jistě ulehčí vaši práci.

Nyní nastal ten správný čas na vytvoření svého prvního program nazývaného **soucet.rb**. Pravda, běžně se vždy začíná napsáním programu typu "Hello World!", tedy programu, který světu ohlašuje "Ahoj světě!" (tady jsem, já, nový programátor). Na druhou stranu, proč nezačít trochu nezvykle tou snad nejstarší vědou - matematikou.

Example: soucet.rb

```
puts 1+2
```

Soubor uložte a vlastní program pak spusťte v příkazovém řádku následovně:

```
$ ruby soucet.rb
3
```

Znak dolaru $ v předchozím zápisu nezadávejte! Symbol pouze značí příkazový řádek (terminál), ve kterém je potřeba příkaz zadat.

Pro spuštění programu v jazyku Ruby je potřeba nějaký interpretr. Tzn. program, který spustí vámi zapsaný program vytvořený v tomto jazyce. Interpretrem je stejnojmenný program **ruby**, který se standardně nachází v balíčku ruby a v Debian GNU/Linuxu jej nainstalujete zadáním příkazu aptitude install ruby.

Tím jste vytvořili svůj první program, zatím nic složitého, ale počítá to. Ostatně není to právě jedna z věcí, kterou si můžeme představit pod slovem počítač -

počítací stroj? V příkladu jsme použili příkaz jazyka Ruby - je jím **puts**, který se chová tak, že na obrazovku vypíše cokoliv co za ním následuje.

Některá pokročilé textové editory či vývojová prostředí vám umožní spouštět kód přímo z programu. Pokud to váš editor neumí, nebo s ním ještě nejste dobře seznámeni, nezbude vám než program spouštět ručně v konzoli mimo něj. Před spuštěním skriptu nezapomeňte program vždy uložit!

Jednoduchá aritmetika

V následujícím příkladě si můžete vyzkoušet jednoduché operace s čísly.

Example: pocitani.rb

```
puts 1.0 + 2.0
puts 2.0 * 3.0
puts 5.0 - 8.0
puts 9.0 / 2.0
```

Jeho výstupem je

```
3.0
6.0
-3.0
4.5
```

Mezery mezi čísly a operátory nejsou nutné, ale výrazně usnadňují čtení programu.

V příkladu jsme poprvé použili reálná čísla. Reálná čísla se liší od celých čísel na první pohled v tom, že obsahují desetinnou čárku a podíváte-li se pozorněji do výše uvedeného příkladu, pak dokonce uvidíte desetinnou tečku! Ruby stejně jako ostatní programovací jazyky používá pro oddělení desetinné části čísla symbol tečky, takže na to pozor.

Použití desetinné tečky je jednoznačně dáno místními zvyklostmi v některých cizích zemích, zejména těch anglicky hovořících. Jinak řečeno, my máme desetinnou čárku, oni mají desetinnou tečku a nezbývá se s tím než smířit.

Pokud provedete to samé s celými čísly, dostanete podobný výsledek.

Example: pocitani2.rb

```
puts 1+2
puts 2*3
puts 5-8
puts 9/2
```

zobrazí

```
3
6
-3
4
```

Povšimněte si výsledku `9/2 = 4`. Protože dělíte celá čísla, je výsledkem opět celé číslo a to doslova za každou cenu. Ruby v tomto případě zahodí desetinnou část ať už je jakákoliv.

I když na vás mohou předchozí příklady počítání s celými čísly působit legračně a příliš jednoduše, jistě se s nimi setkáte i později a to dokonce častěji než si můžete přát. V programech se často objevují počítání typu "jedna a jedna", "deset bez pěti" nebo "dvakrát dva", tedy na první pohled jednoduché výpočty, ale ve výsledku… ehm, to se budete ještě divit. Prostě umět dobře počítač s celými čísly se opravdu vyplatí.

Nyní nastal ten správný čas na to vyzkoušet složitější příklady. Podívejte se na **pocitani3.rb**.

Example: pocitani3.rb

```
puts 5 * (12-8) + -15
puts 98 + (59872 / (13*8)) * -52
puts 5 * 12-8 + -15
```

a jeho výstup

```
5
-29802
37
```

Tady asi netřeba nějaký zvláštní komentář. Je totiž vidět, že ruby interpretuje přednosti matematických operací tak jak jsme zvyklí (násobení má přednost před sčítáním), nevadí mu záporná znaménka (např. číslo -15) a také můžeme závorkovat dle libosti. Inu závorek není nikdy dost.

Příklady 1

- Kolik hodin má rok?
- Kolik minut má desetiletí?
- Kolik sekund vám je?
- Je mi 1 000 miliónů sekund, kolik mi je?

Tím bychom měli základní operace s čísly hotové, příště se podíváme jak se pracuje s textem.

Text

V dnešním díle se podíváme na to co je a není text a především jak se s ním v Ruby pracuje.

Lépe než o textu, měli bychom hovořit o **textovém řetězci**, protože text je vlastně řetězec po sobě následujících písmen. Pro jednoduchost budeme používat označení **text**, ale pamatujme i na tuto jeho vlastnost a označení.

V následujícím výpisu se nachází příklady textu tak, jak ho zapisujeme v programovacím jazyce Ruby.

```
'Ahoj.'
'5 je moje oblíbené číslo... jaké je vaše?'
'Snoopy řekl #%^?&*@! a pak sletěl ze stolu.'
'     '
''
```

Je vidět, že se text uvozuje pomocí jednoduchých uvozovek a přitom je prakticky jedno co je jimi ohraničeno. Text může obsahovat jedno slovo, celou větu, čísla, mezery i speciální symboly. Poslední text bychom mohli označit pojmem **prázdný text** nebo lépe **prázdný textový řetězec**.

Symbol jednoduché uvozovky najdete na klávesnici vlevo vedle klávesy Enter. Na její napsání je třeba přepnout klávesnici na angličtinu, nebo použít pravou klávesu Alt někdy označovanou též symbolem Alt Gr.

Nyní se dostáváme k programu typu "Hello world!" ("Ahoj světě!"), kterým začíná většina odborných knih uvádějících člověka do světa programování. No a protože jsme už patříme mezi ty zkušenější programátory, tak si ho rovnou trochu vylepšíme:

Example: hello_world.rb

```
puts 'Ahoj světě!'
puts ''
puts 'Měj se.'
```

zobrazí očekávané:

```
Ahoj světě!

Měj se.
```

Pro výpis textu se tedy používá známý příkaz puts, za kterým následuje vlastní text.

Pro výpis prázdného řádku můžete použít nejen výše zmiňovaný příkaz **puts ''**, ale i příkaz **puts** samotný. Je tedy jedno zda za puts následuje prázdný řetězec nebo "nic", v obou případech se na výstup vypíše prázdný řádek.

Aritmetika s řetězci

I když to může působit poněkud záhadným dojmem a možná si teď někdo ťuká i na čelo, v této části budeme text sčítat a násobit. Podívejme se na další příklad:

Example: text2.rb

```
puts 'Sklenku vína' + 'si dám moc rád.'
```

vypíše

```
Sklenku vínasi dám moc rád.
```

V programu **text2.rb** jsme spojili dva texty dohromady. Aby byla věta správně, je potřeba přidat mezeru. To provedete jedním z následujících způsobů:

Example: text3.rb

```
puts 'Sklenku vína ' + 'si dám moc rád.'
puts 'Sklenku vína' + ' si dám moc rád.'
```

```
Sklenku vína si dám moc rád.
Sklenku vína si dám moc rád.
```

Je vidět, že je jedno zda mezeru přidáte na konec prvního řetězce, nebo na začátek druhého řetězce. Rozhodnutí závisí na vás a vašich zvyklostech.

Aby toho nebylo málo, text je možné také násobit.

Example: text4.rb

```
puts 'Popis cesty je následující:'
puts 'doleva ' * 5 + 'doprava ' * 2
```

vypíše

```
Popis cesty je následující:
doleva doleva doleva doleva doleva doprava doprava
```

Vynásobením textového řetězce se provede jeho zopakování.

Text vs. číslo

Jaké jsou tedy odlišnosti mezi textem a číslem? Pro vysvětlení se podívejme na tento příklad:

Example: text5.rb

```
puts  12  +  12
puts '12' + '12'
puts '12  +  12'
puts
puts  2  *  5
puts '2' *  5
puts '2  *  5'
```

```
24
1212
12  +  12

10
22222
2  *  5
```

Pozor, text a číslo nelze libovolně kombinovat. Alespoň ne v tom smyslu, že **s textem nelze počítat** a naopak **číslo se nechová jako text**.

V rozporu s předchozí poznámkou zkusme přesto číslo a text zkombinovat.

Example: text6.rb

```
puts '12' + 12
```

Po spuštění programu se vám zobrazí následující chybové hlášení jazyka Ruby:

```
text6.rb:1:in `+': can't convert Fixnum into String (TypeError)
        from text6.rb:1
```

Česky řečeno, nemohu převést číslo na text a program skončí chybou …

Konečně, ne všechny znaky je možno napsat tak jednoduše jak se může zdát. Pokud bychom např. chtěli zdůraznit veledůležitou informaci o tom, že je nutno zadat uživatelské **'jméno'** a **'heslo'**, musíme to zapsat takto:

Example: text7.rb

```
puts 'Zadejte prosím své uživatelské \'jméno\' a \'heslo\':'
```

```
Zadejte prosím své uživatelské 'jméno' a 'heslo':
```

Symbol jedné uvozovky (') hraje v jazyku Ruby speciální význam a tak je k nim potřeba také přistupovat. Před tyto *speciální symboly* se umísťuje symbol lomítka (\) a tím se jejich význam potlačí.

- Napište text: Jmenuji se 'Igor Hnízdo'.

Příklady 2

- Napište s využitím opakování části textu binární číslo: 1 1111 1111 1111 0000 0001. Nejprve to zkuste s naznačenými mezerami a pak i bez mezer.
- Napište následující výstupu programu:

```
 -------------------------
| Robot 'Karel' říká |
 -------------------------
```

A co dál? V příštím díle si povíme něco o proměnných a přiřazení.

Proměnné a přiřazení

Dnešní povídání bude proti tomu minulému o něco kratší, ale přesto o dost důležitější. Hovořit budeme o něčem, bez čeho se žádný programátor neobejde. Tímto speciálním nástrojem není kafe, ani sluchátka, ale je jím to, co se nazývá *proměnná*. Pojďme se nyní zamyslet nad následujícím programem.

Example: jsem_nejlepsi.rb

```
puts 'Jsem nejlepší na světě.'
puts 'Jsem nejlepší na světě.'
```

Z předchozího dílu je jasné, že tento opravdu jednoduchý program dvakrát vypíše uvedenou větu. Pokud bychom tento výpis opakovali na různých místech programu, nejspíše bychom využili zkratkových kláves CTRL + C a CTRL + V nebo hbitosti našich prstů… Na druhou stranu je vidět, že tento způsob není příliš praktický ani příliš hospodárný.

Za pomoci *proměnné* bychom si předcházející příklad zjednodušili takto:

Example: prom1.rb

```
mujText = 'Jsem nejlepší na světě.'
puts mujText
puts mujText
```

Zde jsme použili proměnnou *mujText*, do které jsme *přiřadili* text. Proměnnou

jsme pak použili ve funkci *puts* čímž došlo k vypsání jejího obsahu. Všimněte si také, že pro *přiřazení* hodnoty (v našem případě textu) do *proměnné* se používá operátor rovná se (=).

Nyní můžete namítnout, že místo dvou řádků máme řádky tři a tím i delší kód programu. Pokud si však uvědomíte, že změnou hodnoty proměnné nahoře se změní i text, který se bude vypisovat všude tam, kde tuto proměnnou používáte, je tato vlastnost k nezaplacení.

Ruby rozlišuje velikost písmen, tzn. je rozdíl mezi proměnnou *mujText*, *MujText*, *mujTEXT*, atd. Všechny tyto proměnné jsou jiné!

Je jistě dobrým zvykem začínat proměnné malým písmenem. Názvy proměnných byste pak měli volit tak, aby byli *samovysvětlující*, tzn. mělo by být na první pohled (při troše té fantazie) patrné, které hodnoty bude daná proměnná obsahovat. Pokud pak chcete vytvářet program, do kterého budou nahlížet i programátoři z celého světa, pak určitě volte proměnné v angličtině. Proměnná kromě písmen může také obsahovat i číslice. Příkladem proměnné tak může být např. *slovo*, *cislo*, *i*, *text1*, atd … Pro velmi usnadnění čtení a pochopení významu velmi dlouhého názvu proměnné pak můžete využít vkládání velkých písmen do jejího názvu, např.: *jednaVelmiDlouhaPromenna*.

Proměnné, které se nemění, tzn. po celou dobu své existence nabývají pouze jednu konkrétní hodnotu, se nazývají *konstanty*. Konstanty značíme obvykle velkými písmeny, např. *AUTOR*, *PODPIS*, *KROK*.

S proměnou můžeme pracovat stejně jako bychom pracovali s hodnotou v ní obsaženou.

Example: prom2.rb

```
jmeno = 'Martin Šín'
puts 'Ty musíš být ' + jmeno
puts 'Říká se, že ' + jmeno + ' je ten nejlepší.'
```

No a konečně nejdůležitější vlastnost proměnných, která je daná mj. už jejich názvem - do proměnné můžete opakovaně přiřazovat:

Example: prom3.rb

```
prohlizec = 'Internet Explorer'
puts 'Nejlepším prohlížečem se stává ' + prohlizec

prohlizec = 'Firefox'
puts 'Nejlepším prohlížečem se stává ' + prohlizec
```

Dosud jsme do proměnných přiřazovali výlučně text, ale do proměnné můžete stejně tak přiřadit i jiné typy objektů, např. čísla:

Example: prom4.rb

```
prom = 'Nějaký řetězec'
puts prom

prom = 3 * (2 + 6)
puts prom
```

Poznámka na závěr - do proměnné můžete přiřadit také jinou proměnnou:

Example: prom5.rb

```
var1 = 8
var2 = var1
puts var1
puts var2

puts ''

var1 = 'osm'
puts var1
puts var2
```

zobrazí

```
8
8

osm
```

```
8
```

Jakmile do proměnné *var2* přiřadíte proměnnou *var1*, přepíše se obsah proměnné *var2* obsahem proměnné *var1*. Tzn. obě proměnné zůstanou nadále nezávislé a změna obsahu proměnné *var1* obsah proměnné *var2* nijak neovlivní. (var je zkratka anglického slova *variable*, česky *proměnná*.)

Tím se dostáváme na závěr dnešního povídání o proměnných. Nezapomeňte si rozmyslet příklad na závěr, jehož řešení nemusí být na první pohled snadné. Tato konstrukce je však typickým a často používaným kusem kódu, který prostě musíte znát. Příště budeme kombinovat a prohlubovat dosud získané znalosti.

Příklady 3

- Vytvořte proměnné *a* a *b*, naplňte je nějakými hodnotami a tyto proměnné vypište. Poté jejich obsah vyměňte. (Návod: Pro výměnu obsahu dvou proměnných budete potřebovat třetí proměnnou, např. *c*.)

Kombinování předchozího

V dnešním povídání se podíváme na to předchozí z trochu jiného úhlu pohledu. Dnes totiž budeme kombinovat něco, co jsme dosud nesměli ani neuměli - čísla a text dohromady. Jinými slovy, budeme počítat s textem a k textu připojovat čísla.

Připomeňme si nejprve problémovou situaci:

```
var1 = 2
var2 = '5'

puts var1 + var2
```

Tento příklad nám vrátí chybu, která je způsobena tím, že dáváme dohromady něco co dohromady dát nelze, tedy číslo a text. Jinak řečeno, počítač neví zda má s hodnotami proměnných pracovat jako s textem a zobrazit výsledek "25", nebo k nim přistupovat jako k číslům a zobrazit řešení "7", jiná možnost není.

Konverze

K tomuto účelu disponuje snad každý programovací jazyk (a Ruby není výjimkou) možností konverze předkládaných hodnot. Tzn. mechanismem, který nám umožní převést text na číslo a číslo na text. V Ruby se tato činnost provádí pomocí funkce *.to_s* (převod na text) a funkce *.to_i* (převod na celé číslo) resp. *.to_f* (převod na reálné číslo). Upravením předchozího příkladu:

Example: konverze.rb

```
var1 = 2
var2 = '5'

puts var1.to_s + var2
puts var1 + var2.to_i
```

dostanete výstup

```
25
7
```

Pojďme se nyní podívat na několik (ne)typických konverzí

Example: konverze2.rb

```
puts '15'.to_f
puts '99.999'.to_f
puts '99.999'.to_i
puts ''
puts '5 je mé oblíbené číslo!'.to_i
puts 'Ptal se tu někdo na číslo 5, nebo ne?'.to_i
puts 'Jak řekla maminka.'.to_f
puts ''
puts 'řetězec'.to_s
puts 3.to_i
```

zobrazí

```
15.0
99.999
99

5
```

```
0
0.0

řetězec
3
```

Je vidět že celé číslo *15* bylo převedeno na číslo reálné, tzn. bylo přidáno jedno desetinné místo. U čísla *99.999* se podle očekávání nic nestalo. V případě převedení čísla *99.999* na celé číslo došlo k zahození desetinné části.

V další části ruby přečetl číslo na začátku textového řetězce, a protože zbytku nerozuměl, tak ho zahodil. V druhém textovém řetězci narazil místo čísla na text a tak se tím dále nezabýval a dosadil si tam číslo *0*. Stejně tak i v posledním převodu s tím rozdílem, že dosadil místo celého čísla *0* reálné číslo *0.0*.

Poslední dvě konverze proběhly podle očekávání.

Jiný pohled na puts

V příkazu *puts* znamená písmeno *s* na konci slovo *string* (řetězec), tzn. *puts* bychom rozepsali jako *put string* neboli *napiš řetězec*. Prakticky to znamená, že příkaz puts převádí cokoliv za ním následuje na textový řetězec. Zároveň ale platí, že pokud se mu nepodaří objekt na text převést, musíte to udělat za něj.

Je tedy jedno zda v programu použijete:

```
puts 20
puts 20.to_s
puts '20'
```

puts vždy vypíše

```
20
20
20
```

Metody *gets* a *chomp*

Zatímco *puts* vypíše textový řetězec, metoda *gets* textový řetězec načte.

Následující program nejprve čeká na vámi zadaný text ukončený klávesou *Enter* a pak tento text vypíše.

Example: gets.rb

```
puts gets
```

A je to! Nyní jsme se naučili získávat vstup od uživatele!

Tímto způsobem bychom mohli vytvořit jednoduchý program, kterému se nejprve představíme a za to nás počítač naším jménem také pozdraví.

Example: pozdrav.rb

```
puts 'Ahoj, jaké je tvoje jméno?'
jmeno = gets
puts 'Ahoj, ' + jmeno + ' je pěkné jméno, rád tě poznávám!'
```

výstup může být např. následující

```
Ahoj, jaké je tvoje jméno?
Martin
Ahoj, Martin
 je pěkné jméno, rád tě poznávám!
```

V čem je problém nyní? Proč počítač pokračoval ve výpisu pozdravu na dalším řádku? Vysvětlení není těžké. Metoda *gets* vrací spolu se zadaným textem i symbol klávesy *Enter*, která byla stisknuta pro odeslání textu. Proto v předchozím příkladě došlo k odsazení textu od sebe. Pro odstranění této informace (znaku nového řádku) se používá metoda *chomp*.

Example: pozdrav2.rb

```
puts 'Ahoj, jaké je tvoje jméno?'
jmeno = gets.chomp
puts 'Ahoj, ' + jmeno + ' je pěkné jméno, rád tě poznávám!'
```

zobrazí podle očekávání

```
Ahoj, jaké je tvoje jméno?
Martin
Ahoj, Martin je pěkné jméno, rád tě poznávám!
```

Také bychom mohli do proměnné *jmeno* přiřadit řetězec i s informací o stisknuté klávese *Enter* a vlastní odstranění odřádkování provést později:

Example: pozdrav22.rb

```
puts 'Ahoj, jaké je tvoje jméno?'
jmeno = gets
puts 'Ahoj, ' + jmeno.chomp + ' je pěkné jméno, rád tě poznávám!'
```

První možnost se však používá častěji. Tím nejpádnějším důvodem jistě bude to, že programátor nemusí na pozdější odstranění znaku nového řádku dál myslet. Pokud si však nejste jisti zda jste nadbytečné znaky odstranili, pak raději použijte metodu chomp znovu.

Příklady 4

- Napište program, který se vás postupně zeptá na vaše jméno a pak i příjmení. Poté uživatele pozdraví celým jménem.
- Napište program, který se zeptá na váš věk a pak tuto informaci převede na měsíce a výsledek zobrazí.
- Napište program, který se vás zeptá na vaše oblíbené číslo. Pak toto číslo zvýší o jedna a předloží ho jako číslo lepší.

A co bude příště? Podíváme se na další metody pro práci s textem a čísly.

Další metody

V minulém díle jsme se setkali s několika metodami určenými k převodu textu na číslo, čísla na text a také k úpravě vstupu uživatele. V dnešní části se podíváme na některé vcelku zábavné možnosti formátování textu a také si rozšíříme naše znalosti matematiky v Ruby.

Zábava s textem

Pojďme se podívat na text a možnosti jeho výstupu. Nejprve však trocha té zábavy. Metoda *reverse* vám totiž umožní otočit vstupní řetězec.

Example: reverse.rb

```
var1 = 'radar'
var2 = 'mississippi'
var3 = 'Dokazete rict tuto vetu pozpatku?'

puts var1.reverse
puts var2.reverse
puts var3.reverse
puts var1
puts var2
puts var3
```

zobrazí

```
radar
ippississim
?uktapzop utev otut tcir etezakoD
radar
mississippi
Dokazete rict tuto vetu pozpatku?
```

Metoda *reverse* si nerozumí s češtinou, takže jí nepoužívejte na slova obsahující diakritiku.

Další metodou pracující s řetězci je metoda *length*, která vrátí délku řetězce včetně mezer.

Example: length.rb

```
puts 'Jaké je vaše celé jméno?'
jmeno = gets.chomp
puts 'Věděli jste, že je vaše jméno \'' + jmeno + '\' dlouhé ' + jmeno.le
```

zobrazí

```
Jaké je vaše celé jméno?
Martin Sin
Věděli jste, že je vaše jméno 'Martin Sin' dlouhé 10 znaků?
```

Opět pozor na češtinu, znak obsahující diakritiku se počítá jako znaky 2.

Mezi další metody patří např. *upcase* (změna písma na velké), *downcase* (změna písma na malé), *swapcase* (inverze velikosti písma dle zadaného slova), *capitalize* (první písmeno bude velké).

Example: fancy.rb

```
slovo = 'ABBrakaDaBRa'

puts slovo
puts slovo.upcase
puts slovo.downcase
puts slovo.swapcase
puts slovo.capitalize
```

zobrazí

```
ABBrakaDaBRa
ABBRAKADABRA
abbrakadabra
abbRAKAdAbrA
Abbrakadabra
```

Na závěr se ještě podívejme na možnosti formátování textu. Pro umístění textu na střed obrazovky se používá metoda *cetner*. Použití metody je stejné jako v následujícím příkladu.

Example: center.rb

```
sirka = 80
puts('Přišel jsem,'.center(sirka))
puts('viděl jsem,'.center(sirka))
puts('zvítězil jsem.'.center(sirka))
```

zobrazí

```
                                  Přišel jsem,
                                   viděl jsem,
                                 zvítězil jsem.
```

80 znaků na řádek je jakýmsi standardem, můžete samozřejmě zvolit i nižší hodnotu. Pokud však nastavíte hodnotu vyšší, nemusí se to správně zobrazovat na všech terminálech.

Pokud vám vadí přílišný počet závorek, pak můžete za metodu center umístit místo závorky mezeru. Význam je stejný, viz dále.

Pro zarovnání textu vlevo a vpravo se používají metody *ljust* a *rjust*. Za použití mezery místo závorky bude příkaz puts vypadat takto:

Example: zarovnani.rb

```
sirka = 40
text = '--> Text <--'

puts text.ljust sirka
puts text.center sirka
puts text.rjust sirka
puts text.ljust (sirka/2) + text.rjust (sirka/2)
```

zobrazí

```
--> Text <--
              --> Text <--
                           --> Text <--
--> Text <--               --> Text <--
```

Poslední příkaz vlastně vytvoří textový řetězec o šířce 20 znaků a vloží do něj *text* zarovnaný vlevo a další textový řetězec o šířce 20 znaků, do kterého vloží text zarovnaný vpravo. Nakonec oba řetězce spojí dohromady a vypíše je na obrazovku.

Více matematiky

Pojďme se nejprve podívat na takové matematické operace jako je mocnina (**), či odmocnina. Možná znáte také zbytek po celočíselném dělení (*%*) (modulo).

Example: matematika.rb

```
puts 5**2
puts 5**0.5
puts 7/3
puts 7%3
puts 365%7
```

zobrazí

```
25
2.23606797749979
2
1
1
```

Zatímco *5**2* znamená "5 na druhou", *5**0.5* je "druhá odmocnina z pěti", nebo-li "5 na jednu polovinu". *7%3* (sedm modulo třemi) je pak jedna, protože: *7 = 3 * 2 + 1*.

(Ne)jen pro zajímavost můžete zkusit také metodu *abs*, která se stará o výpočet absolutní hodnoty daného čísla.

Example: absolut.rb

```
puts ((5-2).abs)
puts ((2-5).abs)
```

zobrazí podle očekávání

```
3
3
```

Náhodná čísla

Čas od času je potřeba vygenerovat nějaké náhodné číslo. K tomu slouží funkce *rand*. Zavoláte-li funkci *rand* bez dalších parametrů, vrátí vám reálné číslo mezi 0 a 1. Bude-li za číslem následovat celočíselná hodnota, vrátí vám číslo ležící mezi číslem 0 a touto *hodnotou - 1*.

Example: random.rb

```
puts rand
puts rand
puts rand
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(1))
puts(rand(1))
puts(rand(1))
puts(rand(99999999999999999999999999999999999999999999))
puts('Podle předpovědi, bude pršet na '+rand(101).to_s+'%.')
```

zobrazí

```
0.594956941672168
0.181544980290849
0.0925772999883406
58
19
75
0
0
0
173608196998218282957848353262484394467716 1
Podle předpovědi, bude pršet na 47%.
```

Ještě jednou a jen pro jistotu - funkce *rand(číslo)* zobrazí náhodné číslo v rozmezí od 0 po hodnotu *číslo - 1*.

Někdy můžete požadovat, aby program použil náhodně generovaná čísla znova (např. při vytvoření dočasných souborů a jejich opětovném otevření). Pak můžete použít funkci *srand*.

Example: srand.rb

```
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts ''
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
```

zobrazí

```
24
35
36
58
70

24
35
36
58
70
```

Objekt Math

Objekt *Math* obsahuje celou řadu matematických funkcí, ukažme si alespoň tyto

Example: math.rb

```
puts(Math::PI)
puts(Math::E)
puts(Math.cos(Math::PI/3))
puts(Math.tan(Math::PI/4))
puts(Math.log(Math::E**2))
puts((1 + Math.sqrt(5))/2)
```

zobrazí

```
3.14159265358979
2.71828182845905
0.5
1.0
2.0
1.61803398874989
```

Co se myslí objektem? Objektem může být dům, auto, matematika nebo pěkná slečna (jinými slovy - snad cokoliv vás napadne). Důležité je si uvědomit, že objekt zastřešuje určitou sadu metod a vlastností, které jsou jeho nedílnou součástí. Stejně jako součástí matematiky je výše uvedená konstanta *pí*, *Eulerovo číslo e*, *odmocnina*, atd.

Příklady 5

- Napište program, který nahradí vašeho šéfa. Program se vás zeptá co chcete a jako odpověď vám napíše *Cože to chcete? **CHCI PŘIDAT** !! Máte padáka!.* Tedy převede váš text na velká písmena a zareaguje jako sám velký šéf…
- Napište program, který vytvoří obsah. Tedy tuto strukturu:

```
                Obsah

Kapitola 1:  Čísla                    strana 1
Kapitola 2:  Písmena               strana 10
Kapitola 3:  Proměnné           strana 25
```

- Napište matematickou sadu programů, které vám spočítají obsah a obvod čtverce a obdélníku, přeponu a velikosti zbývajících úhlů v pravoúhlém trojúhelníku a konečně také obvod a obsah kruhu.

Protože toho už umíme docela dost, příště se podíváme na řízení běhu programu, tzn. budeme učit program "se rozhodovat" a také se naučíme používat cykly.

Řízení běhu programu

Dnes to bude náročné, budeme vytvářet programy, které se budou sami rozhodovat. Naštěstí pro nás, toto rozmýšlení nebude tak složité jako v běžném životě. Také se podíváme na cykly, které nám umožní opakovat určité částí programu aniž bychom je museli psát znovu. Takže pusťme se do toho.

Podmínky

Stejně jako se musíme rozhodovat my, koná tato rozhodnutí během svého běhu i většina počítačových programů. Rozhodnutí jsou činěna na základě vyhodnocení určitého výrazu a stanovení zda je či není pravdivý, tedy zda *je pravda* (*true*) nebo *není pravda* (*false*).

Example: je_pravda.rb

```
puts 1 > 2
puts 2 > 1

puts ''

puts 5 >= 5
puts 5 <= 4

puts ''

puts 1 == 1
puts 2 != 1
```

odpovědí programu je

```
false
true
```

```
true
false

true
true
```

Tedy platí, že 1 není větší než 2, 2 je větší než 1, 5 je větší nebo rovno než 5 a 5 není menší nebo rovno než 4. Poslední dvě podmínky značí, porovnání, resp. nerovnost dvou čísel. Tedy je pravda, že se číslo 1 rovná číslu 1 a také platí, že číslo 2 se nerovná číslu 1.

Porovnání == resp. *!=* se v programování používají velmi často. Běžně je potřeba zjistit, zda se nějaká proměnná rovná či nerovná námi hledané či sledované hodnotě.

Zatímco zápis *a = b* znamená do *a* přiřaď *b*, výsledkem zápisu *a == b* je hodnota *true* nebo *false*.

Také můžete porovnávat dva textové řetězce

Example: porovnani.rb

```
puts 'kocka' < 'pes'
puts 'kocka' > 'pes'
puts 'KOCKA' > 'pes'
```

zobrazí

```
true
false
false
```

V případě textových řetězců se porovnává jejich umístění v abecedě.

Platí, že písmena velké abecedy se nachází před písmeny malé abecedy a tak např. *Zebra* se nachází před *autem*.

> Pokud nevíte, zda budou slova začínat malými či velkými písmeny, můžete použít pro sjednocení velikosti metody *downcase*, *upcase* a *capitalize*.

Vyhodnocení podmínky

Nyní, když jsme schopni určit co je a není pravda, můžeme na základě těchto znalostí činit závěry. K tomu slouží konstrukce *if* podmínka ... *end* (*jestliže* podmínka pak udělej ... *konec*).

Example: pozdrav3.rb

```
puts 'Ahoj, jak se jmenuješ?'
jmeno = gets.chomp
puts 'Ahoj ' + jmeno + '!'
if jmeno == 'Martin'
   puts 'To je ale pěkné jméno!'
end
```

zobrazí

```
Ahoj, jak se jmenuješ?
Martin
Ahoj Martin!
To je ale pěkné jméno!
```

V případě, že zadáte jiné jméno, pak bude výstupem např. následující:

```
Ahoj, jak se jmenuješ?
Adam
Ahoj Adam!
```

Jistě bychom dříve či později chtěli provést také odpovídající činnost v situaci kdy nebude počáteční podmínka splněna. K tomu slouží klíčové slovo *else* (*jinak*).

Example: pozdrav4.rb

```
puts 'Ahoj, jak se jmenuješ?'
jmeno = gets.chomp
puts 'Ahoj ' + jmeno + '!'
```

```
if jmeno == 'Martin'
   puts 'To je ale pěkné jméno!'
else
   puts 'Nechceš se nechat přejmenovat?'
end
```

zobrazí v případě nesplnění podmínky

```
Ahoj, jak se jmenuješ?
Adam
Ahoj Adam!
Nechceš se nechat přejmenovat?
```

Podmínky lze do sebe také vnořovat. Tyto programy jsou na první pohled trochu složitější ale díky správnému strukturování kódu se čtou celkem snadno.

Example: pozdrav5.rb

```
puts 'Ahoj, jak se jmenuješ?'
jmeno = gets.chomp
if jmeno == jmeno.capitalize
    puts 'Ahoj ' + jmeno + '!'
else
   puts 'Myslel jsi ' + jmeno.capitalize + ', že?'
   puts 'Umíš vůbec správně napsat své jméno?'

   odpoved = gets.chomp
   if odpoved.downcase == 'ano'
        puts 'Aspoň, že tak..'
   else
        puts 'Přestaň si ze mě dělat srandu!'
   end
end
```

zobrazí např.

```
Ahoj, jak se jmenuješ?
adam
Myslel jsi Adam, ze?
Umíš vůbec správně napsat své jméno?
ano
```

```
Aspoň, že tak..
```

Kód vašeho programu byste měli psát s ohledem na provádění pozdějších změn. Neměl by působit dojmem šifry, ale měl by být příjemným čtením po nedělním obědu či před spaním.

Cykly

Cykly představují opakující se smyčku, která skončí až v okamžiku kdy to dovolíme.

Example: cyklus.rb

```
prikaz = 'Zadejte příkaz'

while prikaz != 'konec'
    puts prikaz
    prikaz = gets.chomp
end

puts 'Na shledanou'
```

zobrazí

```
Zadejte příkaz
help
help
?
?
konec
Na shledanou
```

Tento cyklus je tedy uvozen klíčovým slovem *while* na jedné straně a *end* na straně druhé. Česky řečeno: *dokud platí podmínka* dělej ... *konec*. Cyklus tedy končí v okamžiku kdy je podmínka nesplněna.

Pozor na nekonečné cykly (cykly, které nikdy neskončí)! Někdy se nám tyto cykly mohou hodit, jindy značně ztěžují hledání chyb. Nekonečný cyklus ukončíte stiskem kláves *CTRL+C*.

Trocha logiky

Pojďme nyní zkombinovat to co jsme se dozvěděli v předchozí části a podívejme se na následující příklad

Example: logika.rb

```
puts 'Jak se jmenujete?'

jmeno = gets.chomp

puts 'Ahoj ' + jmeno + '.'

if jmeno == 'Martin'
   puts 'To je opravdu pěkné jméno.'
else
   if jmeno == 'Adam'
      puts 'To je opravdu pěkné jméno.'
   end
end
```

Tento zápis je jistě v pořádku, ale jak je vidět, není příliš praktický ani úsporný. Celé si to můžeme zkrátit pomocí podmínky *or* (česky *nebo*).

Example: logika2.rb

```
puts 'Jak se jmenujete?'

jmeno = gets.chomp

puts 'Ahoj ' + jmeno + '.'

if (jmeno == 'Martin' or jmeno == 'Adam')
      puts 'To je opravdu pěkné jméno.'
end
```

Platí následující poučka: *Co nemá programátor v hlavě, to musí mít v prstech.* Tuto větu lze i zobecnit …

Mezi další logické operátory patří *and* (*a*) a *not* (*negace výroku*). Jak tyto operátory fungují můžete vidět v dalším příkladě.

Example: logika3.rb

```
jaJsemMartin = true
jsemZelenej = false
jimMaso = true
mamRadBehani = false

puts (jaJsemMartin and jimMaso)
puts (jaJsemMartin and jsemZelenej)
puts (jsemZelenej and jimMaso)
puts (jsemZelenej and mamRadBehani)
puts
puts (jaJsemMartin or jimMaso)
puts (jaJsemMartin or jsemZelenej)
puts (jsemZelenej or jimMaso)
puts (jsemZelenej or mamRadBehani)
puts
puts (not jaJsemMartin)
puts (not jsemZelenej)
```

zobrazí

```
true
false
false
false

true
true
true
false

false
true
```

- Napište program, který bude počítat ovečky až do 200 …

Příklady 6

- Napište program, který po zadání počátečního a koncového roku napíše všechny přestupné roky, které byly mezi těmito roky. (Pro jednoduchost předpokládejme, že je přestupný rok beze zbytku dělitelný číslem 4.)
- Napište variantu programu *yes*. Tento program se standardně chová tak, že na každý řádek napíše písmeno *y* a vypisuje ho tak dlouho, dokud uživatel nestiskne kombinaci kláves *CTRL+C*. (Použijte nekonečnou smyčku.)
- Napište program, který se vás bude ptát na jméno a po jeho napsání vás pozdraví. Program skončí zadáním slova *konec*.

A co bude příště? Příště se podíváme na pole. Ne, nemusíte si chystat pracovní oděv, řeč bude o poli prvků ať už jsou jeho prvky jakékoliv.

Pole

Dnešní téma zní jednoduše, budeme se totiž zabývat polem a ničím jiným než polem. Co se ovšem tímto polem vlastně rozumí a především jak se s ním pracuje a jaké jsou jeho výhody? Pojďme se do toho pustit.

Pole si můžete představit jako řadu prvků spojených dohromady. Tyto prvky jsou uvozeny znakem *hranaté závorky* a odděleny čárkou. Polem je např.:

```
[]
[5]
['Ahoj', 'Na shledanou']

promenna = 'text'   # toto není pole, ale proměnná
[89.9, promenna, [true, false]]
```

První pole je *prázdné*, druhé obsahuje číslici 5, třetí pole obsahuje dva textové řetězce a konečně poslední pole obsahuje číslo, hodnotu proměnné *promenna* a posledním prvkem pole je pole obsahující pravdivostní hodnoty *true* a *false*.

Vzpomínáte na prázdný textový řetězec (*''*) ? Tak proč bychom nemohli mít prázdné pole (*[]*), že?

Pojďme se nyní podívat jak se s jednotlivými prvky pole vlastně pracuje

Example: pole.rb

```
jmena = ['Martin', 'Honza', 'Alena']

puts jmena
puts
puts jmena[0]
puts jmena[1]
puts jmena[2]
puts jmena[3]
```

zobrazí

```
Martin
Honza
Alena

Martin
Honza
Alena
nil
```

K jednotlivým prvkům pole se tedy přistupuje prostřednictvím čísla (pozice) v poli. Jen je důležité pamatovat na to, že se jednotlivé prvky pole číslují od 0 a zároveň hodnota *nil* nás informuje o tom, že se již v poli žádný další prvek nenachází (v našem případě, že prvek pole s pořadovým číslem 3 neexistuje).

Hodnota *nil* je speciální objekt a znamená konec pole.

Metoda *each*

Metoda *each* vám umožní provést něco (cokoliv chcete) s každým prvkem pole. Zkusme následující příklad.

Example: pole2.rb

```
jazyky = ['Čeština', 'Angličtina', 'Němčina']

jazyky.each do |jazyk|
   puts 'Můj nejoblíbenější jazyk je ' + jazyk + '!'
   puts 'Máte ho také rádi?'
end
```

zobrazí

```
Můj nejoblíbenější jazyk je Čeština!
Máte ho také rádi?
Můj nejoblíbenější jazyk je Angličtina!
Máte ho také rádi?
Můj nejoblíbenější jazyk je Němčina!
Máte ho také rádi?
```

Je vidět, že se každá z hodnot pole *jazyky* v cyklu postupně dosadí do proměnné *jazyk*, kterou pak dál používáme pro výpis daného jazyku.

Podobně vypadá následující konstrukce, která nám umožní zopakovat nějaký kus kódu X-krát. Pozor, přitom se ale nejedná o pole, ale pouze metodu (objektu) celého čísla:

Example: iterace.rb

```
3.times do
  puts 'Mám tě rád.'
end
```

zobrazí

```
Mám tě rád.
Mám tě rád.
Mám tě rád.
```

Pokročilá práce s polem

Podívejme se nyní na další metody, které nám usnadní provádění většiny běžných operací, které někdy můžeme chtít s polem provést.

Metody *join* a *to_s*

Example: potraviny.rb

```
potraviny = ['mléko', 'maso', 'pivo']

puts potraviny
puts
puts potraviny.to_s
puts
puts potraviny.join(', ')
puts
puts potraviny.join('  :)  ') + '  8)'

200.times do
  puts []
end
```

zobrazí

```
mléko
maso
pivo

mlékomasopivo

mléko, maso, pivo

mléko  :)  maso  :)  pivo  8)
```

Několik poznámek k předchozímu programu

- *puts potraviny* vytiskne položky pole potraviny pod sebe
- *puts potraviny.to_s* spojí položky pole dohromady, tedy je napíše jako jedno slovo
- *puts potraviny.join(', ')* přidá ke každé položce pole text uvedený v závorce
- *puts potraviny.join(' :) ') + ' 8)'* si můžete pro snadnější pochopení představit jako řetězec *puts potraviny.join(' :) ')* PLUS řetězec *' 8)'*
- konečně poslední cyklus neprovede nic, přesněji řečeno 200-krát vypíše prázdné pole, tedy nevypíše nic

Metody *push*, *pop*, *last*, *length*

Zkusme tento příklad

Example: oblibene.rb

```
oblibene = []

oblibene.push 'víno a brambůrky'
oblibene.push 'stránky Microsoftu'

puts oblibene[0]
puts oblibene.last
puts oblibene.length

puts oblibene.pop
puts oblibene
puts oblibene.length
```

zobrazí

```
víno a brambůrky
stránky Microsoftu
2
stránky Microsoftu
víno a brambůrky
1
```

a nyní vysvětlení

- metoda *push* přidává do pole další položky, položky jsou přidávány postupně jak přijdou, nová položka je přidána na konec pole
- metoda *last* zobrazí poslední položku v poli
- metoda *length* nám řekne kolik prvků pole obsahuje (pozor, index posledního prvku v poli je o jedničku menší!)
- metoda *pop* pak vrací položku z konce pole a zároveň jí z pole odebere

Pamatujte, že metody *push* a *pop* mění obsah pole. Metoda *push* prvky do pole přidává a metoda *pop* je odebírá!

Příklady 7

- Napište program, který od nás bude na vstupu čekat neomezené množství slov (jedno slovo na řádek). Jakmile pak stisknete klávesu <Enter> na prázdném řádku, ukončí vstup slov a vypíše námi napsaná slova opačně, tzn. od posledního slova po první slovo, které jsme zadali.
- Upravte předchozí příklad tak, že zadaná slova setřídíte, použijte metodu *sort*. Její syntaxe je snadná - *pole.sort* vrátí setříděný seznam prvků pole.
- Přepište program, ve kterém jste vytvářeli Obsah nějaké knihy. Jednotlivé kapitoly uchovejte jako prvky pole.
- Napište program, který vytvoří pole 20 náhodných dvouciferných čísel. Zajistěte, aby se čísla uvedená v poli neopakovala!

Příště se podíváme na psaní vlastních metod. Jak sami uvidíte, usnadníme si tím psaní delších programů a zároveň zpřehledníme vlastní kód programu.

Psaní vlastních metod

Jak bylo řečeno na závěr minulého článku, metody vám usnadní psaní delších programů a zároveň zpřehlední vlastní zdrojový kód. Také vám umožní zapomenout jednou provedené a někdy i značně složité konstrukce a v další části programu se na ně jen *odvolávat*.

Vynechejme nyní trochu únavné teorie a podívejme se rovnou na následující příklad, ve kterém budeme uživateli pokládat celou spoustu otázek. Pro jednoduchost, nás nebudou zajímat jeho odpovědi, pouze budeme kontrolovat, zda uživatel napsal ano nebo ne.

Example: dotaznik.rb

```
odpoved_ok = false

while (not odpoved_ok)
  puts 'Máš rád zvířata?'
  odpoved = gets.chomp.downcase

  if (odpoved == 'ano' or odpoved == 'ne')
        odpoved_ok = true
  else
        puts 'Zadejte "ano" nebo "ne"'
  end
end

odpoved_ok = false
```

```
while (not odpoved_ok)
  puts 'Už jsi někdy jedl psa?'
  odpoved = gets.chomp.downcase

  if (odpoved == 'ano' or odpoved == 'ne')
        odpoved_ok = true
  else
        puts 'Zadejte "ano" nebo "ne"'
  end
end

odpoved_ok = false

while (not odpoved_ok)
  puts 'A co vlka, vlka jsi měl?'
  odpoved = gets.chomp.downcase

  if (odpoved == 'ano' or odpoved == 'ne')
        odpoved_ok = true
  else
        puts 'Zadejte "ano" nebo "ne"'
  end
end

odpoved_ok = false

while (not odpoved_ok)
  puts 'A medvěda jsi jedl?'
  odpoved = gets.chomp.downcase

  if (odpoved == 'ano' or odpoved == 'ne')
        odpoved_ok = true
  else
        puts 'Zadejte "ano" nebo "ne"'
  end
end
```

Je vidět, že tento postup není příliš praktický. Ne, že bychom tímto způsobem program nevytvořili, ani nemůžeme říct, že by nefungoval tak, jak má. Je ale hned na první pohled vidět, kolik práce nás jeho vytvoření stálo a přitom se se určité části programu pouze pravidelně opakují…

K tomu nám *ruby* umožňuje vytvářet a především používat vlastní metody. Zkusme vytvořit následující jednoduchou metodu, kterou označme *pozdrav*.

Example: link:priklady/pozdrav6.rb[pozdrav6.rb)

```
def pozdrav
   puts 'Ahoj!'
end

pozdrav
pozdrav
puts 'Jak se vede?'
pozdrav
```

zobrazí

```
Ahoj!
Ahoj!
Jak se vede?
Ahoj!
```

V příkladu jsme definovali vlastní metodu *pozdrav* a tu pak několikrát použili. No není to pěkné?

Pokud bychom chtěli metodu *pozdrav* změnit, stačí jenom upravit její definici nahoře a všude tam, kde metodu používáme se použije její nová aktualizovaná verze. Tzn. již žádné hledání v programu a úpravy typu Najít & Nahradit.

Parametry metod

Vraťme se nyní k úvodnímu příkladu. Abychom si mohli tento program zjednodušit, potřebovali bychom metodě předávat nějaký parametr obsahující naši otázku. I na toto *ruby* pamatuje:

Example: dotaznik2.rb

```
def otazka text
  odpoved_ok = false

  while (not odpoved_ok)
     puts text
```

```
      odpoved = gets.chomp.downcase

      if (odpoved == 'ano' or odpoved == 'ne')
         odpoved_ok = true
      else
         puts 'Zadejte "ano" nebo "ne"'
      end
   end
end

otazka 'Máš rád zvířata?'
otazka 'Už jsi někdy jedl psa?'
otazka 'A co vlka, vlka jsi měl?'
otazka 'A medvěda jsi jedl?'
```

Tím se nám program zjednodušil nejen svým rozsahem, ale také svým obsahem. Zdrojový kód programu by určitě neměl působit dojmem nějaké šifry!

Lokální proměnné

Lokální proměnné jsou takové proměnné, které se objevují uvnitř metody a jsou vidět (dostupné) pouze v této metodě a nikde jinde. Zkusme následující příklad.

Example: umocneni.rb

```
def umocneni cislo
        vysledek = cislo * cislo
        puts 'mocnina čísla ' + cislo.to_s + ' je ' + vysledek.to_s
end

umocneni 5
puts vysledek
```

zobrazí

```
mocnina čísla 5 je 25
umocneni.rb:7: undefined local variable or method `vysledek' for main:Obj
```

Je vidět, že proměnná *vysledek* existuje pouze v metodě *umocnění* a tak s ní není možno mimo tuto metodu dál pracovat. Na druhou stranu, to samé platí i pro proměnné definované mimo jakoukoliv metodu (tuto část programu můžeme nazývat např. hlavním programem). Zkusme tuto ukázku

Example: lokalni_promenne.rb

```
def likvidator promenna
   promenna = nil
   puts 'CHA, a máš po proměnné!'
end

promenna = 'Nějaký velmi dlouhý a důležitý text'
likvidator promenna
puts promenna
```

zobrazí

```
CHA, a máš po proměnné!
Nějaký velmi dlouhý a důležitý text
```

tedy, obsah proměnné *promenna* zůstal zachován.

Tato vlastnost platnosti proměnných v programu je velmi důležitá a proto si jí dobře promyslete!

Návratové hodnoty metod

Přestože jsou globální i lokální proměnné navzájem neviditelné, často potřebujeme výsledek nějaké funkce dosadit do nějaké proměnné. K tomuto (a nejen k tomu) slouží návratové hodnoty metod. Zkusme následující

Example: navratova_hodnota.rb

```
def metoda
   puts 'Včera jsem byl doma.'
   puts 'Dělal jsem úkoly do školy.'
   'A koukal na televizi'
```

```
end

metoda
puts

pokus = metoda
puts pokus
puts

pokus2 = puts pokus
puts pokus2
```

zobrazí

```
Včera jsem byl doma.
Dělal jsem úkoly do školy.

Včera jsem byl doma.
Dělal jsem úkoly do školy.
A koukal na televizi

A koukal na televizi
nil
```

Takže co se to vlastně stalo

- první volání metody *metoda* zobrazilo výstup přesně tak jak bychom očekávali
- druhým voláním metody *metoda* jsme přiřadili do proměnné pokus její návratovou hodnotu. Tou je vždy **poslední proměnná či hodnota** uvedená na závěr metody
- poslední volání *puts pokus* ukazuje na jednu zajímavou vlastnost samotné metody puts, která spočívá v tom, že metoda posílá jako poslední hodnotu *nil*, která nás informuje o konci textu.

Na závěr zkusme trochu složitější program, kterému zadáme nějaké číslo v rozmezí od 0 po 99 a program nám napíše, jak bychom toto číslo vyslovili.

Example: rekni_cislo.rb

```
def rekni_cislo cislo
 cislice = ['nula', 'jedna', 'dva', 'tri', 'ctyri', 'pet', 'sest', 'sedm'
 desitky = ['deset', 'dvacet', 'tricet', 'ctyricet', 'padesat', 'sedesat'
```

```
  vyslov = ''

  desitek = cislo/10

  if (desitek !=  0)
     vyslov = desitky[desitek-1]
     if (cislo != 10*desitek)
        vyslov = vyslov + cislice[cislo-10*desitek]
     end
  else
     vyslov = vyslov + cislice [cislo]
  end

  vyslov
end

puts rekni_cislo(5)
puts rekni_cislo(8)
puts rekni_cislo(50)
puts rekni_cislo(87)
puts rekni_cislo(66)
puts rekni_cislo(10
```

program zobrazí

```
pet
osm
padesat
osmdesatsedm
sedesatsest
deset
```

Metodu *funkce* s parametrem v podobě čísla *5* můžete zavolat *funkce 5* i *funkce(5)*. Obojí syntaxe je možná.

Příklady 8

- V předchozím příkladu je chyba a program nebude správně zapisovat čísla 11, 12, 13, … 19. Opravte program tak, aby fungoval správně, nebo

napište svou vlastní verzi, ve které bude chyba odstraněna.

- Upravte první příklad tohoto cvičení tak, aby nejenom používal metody, ale také si *zapamatoval* výsledné hodnoty. Pro uchování hodnot můžete použít proměnné nebo lépe *pole*. Výsledek pak najednou vypište, odpovědi od sebe oddělte symbolem středník.
- Napište program, který vám spočítá faktoriál nějakého čísla. Faktoriál čísla *n*, tedy *n!*. Faktoriál se spočítá takto: *n! = n(n-1)(n-2)(n-3)…1*, tedy např. *5! = 5.4.3.2.1 = 120*. Pro výpočet faktoriálu je potřeba volat funkci pro výpočet faktoriálu *rekurzivně*. ;-)

Příště se podíváme na třídy. Nebudeme se sice vracet do školních lavic, ale podíváme se zblízka na to, jak jsou konstruovány objekty a jak se s nimi v *ruby* pracuje.

Třídy

Musím se přiznat, že práce s objekty a třídami těchto objektů patří mezi mé neoblíbenější činnosti. Třídy vám umožní vytvořit svůj vlastní objekt a ten dál používat, vylepšovat, prostě s ním pracovat. S třídami, aniž bychom si to uvědomili, jsme se setkali už mnohem dříve a dneska se na ně podíváme podrobněji.

Objektem může být cokoliv, auto, dům, člověk, počítač, robot, svět… Zde se fantazii meze nekladou. Třída pak zastřešuje skupinu objektů stejných vlastností (např. všechna auta mají motor, kapotu, určitou spotřebu, atd.).

Nejprve se seznamme s několika třídami, které jsou v ruby definovány. Základní metodou v podstatě každé třídy je metoda *new*, která vytvoří nový objekt dané třídy. Ukažme si to na následujícím příkladu:

Example: tridy.rb

```
a = Array.new + [12345]
b = String.new + 'Ahoj'
c = Time.new

puts 'a = ' + a.to_s
puts 'b = ' + b.to_s
puts 'c = ' + c.to_s
```

zobrazí

```
a = 12345
b = Ahoj
c = Wed Jun 11 13:36:17 +0200 2008
```

A co jsme to vlastně vytvořili?

- objekt *pole* (*Array*) a pak do něj přidali jedenu položku
- objekt *řetězec* (*String*) a přidali do něj text
- objekt *čas* (*Time*), který se automaticky inicializoval aktuální informací o času a datu dle nastavení počítače.

Umět říct kolik je hodin patří bezesporu mezi základní dovednosti žáků základní školy, pokud umíte s časem i pracovat, můžete nějaký ten čas i ušetřit. Podívejme se tedy na třídu *Time* zblízka.

Třída čas

Zkusme několik základních příkladů. S časem je možno počítat:

Example: cas.rb

```
cas = Time.new
cas2 = cas + 60  # o minutu déle

puts cas
puts cas2
```

zobrazí

```
Wed Jun 11 14:06:03 +0200 2008
Wed Jun 11 14:07:03 +0200 2008
```

Čas můžete pomocí metody *mktime* nastavit na vámi zvolenou hodnotu.

Example: cas2.rb

```
puts Time.mktime(2000,1,1)
puts Time.mktime(1970, 2, 1, 10, 0)
```

zobrazí

```
Sat Jan 01 00:00:00 +0100 2000
Fri Jan 02 10:00:00 +0100 1970
```

Čas lze také porovnávat s jiným časem a určit, který čas je starší nebo naopak novější. Konečně, časy můžete mezi sebou odečítat a určit tak kolik doby (sekund) od té doby uteklo.

Třída *Hash*

O třídě *Hash* jsme dosud nemluvili, patří však mezi třídy o kterých bychom měli vědět. *Hash* se podobá poli, ale oproti němu nejsou položky uspořádány lineárně za sebou (nejsou vzestupně očíslovány počínaje 0). *Hash* se hodí především pro neuspořádanou skupinu prvků.

Neuspořádaná skupina prvků je taková skupina prvků, ve které *nezáleží na pořadí*.

Example: hash.rb

```
poleBarev = []  #  to samé jako metoda Array.new
hashBarev  = {}  #  to samé jako metoda Hash.new

poleBarev[0]         = 'červeně'
poleBarev[1]         = 'zeleně'
poleBarev[2]         = 'modře'
hashBarev['text']  = 'červeně'
hashBarev['čísla']  = 'zeleně'
hashBarev['klíčová slova'] = 'modře'
```

```
poleBarev.each do |barva|
  puts barva
end

puts

hashBarev.each do |vyuziti, barva|
  puts vyuziti + ':  ' + barva
end
```

zobrazí

```
červeně
zeleně
modře

text:  červeně
čísla:  zeleně
klíčová slova:  modře
```

V *hashi* je každý prvek označen nějakým symbolem, v našem případě jím je slovo, ale mohl by to být také znak nebo číslice.

Rozšíření třídy

V předchozí kapitole jsme měli příklad, který vyslovil číslo tak jak se čte. Zkusme nyní něco podobného, ale s tím rozdílem, že rozšíříme stávající třídu *celých čísel* (*Integer*) o další metodu, kterou nazveme např. *to_czech* (*do_cestiny*). (Celý příklad si pro jednoduchost zápisu velmi zjednodušíme. ;-))

Example: rozsireni_tridy.rb

```
class Integer

   def to_czech
     if self == 5
        cesky = 'pet'
     else
        cesky = 'padesatpet'
```

```
      end

      cesky
    end

end

puts 5.to_czech
puts 55.to_czech
```

zobrazí

```
pet
padesatpet
```

Pro označení objektu jsme použili jeho identifikátor v podobě klíčového slova *self*. Tato proměnná se vždy odkazuje na *sebe sama*, tedy objekt, ve kterém se nacházíme.

Vytváření tříd

Zatím jsme třídy jenom používali či rozšiřovali. Pojďme se podívat jak třídu vytvořit, jak vlastně vzniká. Zkusme následující jednoduchou třídu.

Example: trida.rb

```
class Kostka

   def hod
     1 + rand(5)
   end

end

hody_kostkou = [Kostka.new, Kostka.new, Kostka.new]

hody_kostkou.each do |kostka|
   puts kostka.hod
end
```

zobrazí např.

```
5
5
1
```

Třída je uvozena klíčovým slovem *class*. Dál v ní můžete vytvářet její vlastní metody pomocí klíčového slova *def*.

Proměnné instancí

Instance je *terminus technikus*, který není ničím jiným než konkrétním objektem dané třídy, tzn. objektem s kterým pracujeme a který má vlastnosti nějaké třídy.

Proměnné instancí jsou dostupné v celé třídě. Abychom je odlišili od lokálních proměnných, začínají znakem zavináč @.

Example: promenne_trid.rb

```ruby
class Kostka

   def hod
      @cislo = 1 + rand(5)
   end

   def vysledek
      @cislo
   end

end

kostka = Kostka.new

kostka.hod
puts kostka.vysledek
puts kostka.vysledek
```

```
kostka.hod
puts kostka.vysledek
puts kostka.vysledek
```

zobrazí

```
2
2
3
3
```

Celý proces házení kostkou bychom si pak mohli zjednodušit využitím metody *initialize*, která se provede automaticky při vytvoření daného objektu.

Example: promenne_trid2.rb

```
class Kostka

   def initialize
       hod
   end

   def hod
      @cislo = 1 + rand(5)
   end

   def vysledek
      @cislo
   end

end

puts Kostka.new.vysledek
puts Kostka.new.vysledek
```

tento příklad zobrazí např. toto:

```
5
4
```

Pokud bychom nyní chtěli uživateli zamezit v tom, aby mohl s kostkou házet,

tzn. aby se kostka inicializovalo pouze při svém vytvoření, můžeme metodu *hod* označit jako *soukromou* (*private*). Tím zajistíme, že k této metodě nebude možno *zvenku* přistupovat.

Example: promenne_trid3.rb

```
class Kostka

   def initialize
       hod
   end

   def vysledek
      @cislo
   end

   private

   def hod
      @cislo = 1 + rand(5)
   end
end

kostka = Kostka.new.vysledek
puts kostka

kostka.hod
```

zobrazí

```
4
promenne_trid3.rb:21: undefined method `hod' for 4:Fixnum (NoMethodError)
```

Tedy třída *Kostka* metodu *hod* nezná!

Pozor cokoliv uvedete za slovem *private* bude jenom soukromé pro danou třídu!

Příklady 9

- Zjistěte kolik vám bude za bilión sekund od vašeho narození. (Pro provedení výpočtu si zkuste zjistit v kolik hodin jste se narodili.)
- Napište program, který se vás zeptá kdy jste se narodili, z tohoto data vypočítá kolik vám je let a dodatečně vám popřeje za každý den vašich narozenin, který jste již měli.
- Napište program - simulátor automobilu s děravou nádrží, kterému na začátku zadáte požadovanou vzdálenost a automobil sám si bude sledovat stav své nádrže a v případě potřeby natankuje benzín. (Pro stanovení dojezdu automobilu na plnou nádrž použijte nějakou vámi předem danou hodnotu v kilometrech plus určitou vzdálenost, která není předem známa a jejíž výpočet necháte na funkci *rand*.) Použijte třídy!

Příště nás čeká poslední díl našeho seriálu, pro zachování určité nádechu tajemna si pouze řekněme, že se bude jmenovat "Bloky a Proc". Co to znamená, a jak to spolu souvisí, se dozvíte již příště.

Bloky a Procs

Jedni to zatracují, jiní to obdivují, názory na následující techniku programování se liší. Ke které skupině programátorů se přikloníte vy, to už záleží jen na vás, oč se jedná a jak se s tím pracuje, to vám ukáže poslední díl seriálu, který je věnovaný programovacímu jazyku Ruby.

Stručně řečeno, Ruby umožňuje vzít blok zdrojového kódu (kódu mezi klíčovými slovy *do* a *end*), umístit ho do nějakého objektu (nazývaný *Proc*), uložit ho v proměnné nebo vložit do metody a spustit tento kus kódu kdekoliv chcete a především kolikrát chcete.

Podobá se to druhu určité metody s rozdílem, že není umístěna v objektu, ale sama o sobě je objektem, který můžete umístit snad kamkoliv.

Example: proc.rb

```
pozdrav = Proc.new do
   puts "Ahoj!"
end

pozdrav.call
pozdrav.call
pozdrav.call
```

zobrazí

```
Ahoj!
Ahoj!
Ahoj!
```

Blokům můžete předávat parametry, stejně jako metodám.

Example: proc2.rb

```
mamRad = Proc.new do |dobrota|
        puts 'Mam *rad* ' + dobrota + '!'
end

mamRad.call 'pivo'
mamRad.call 'vino'
```

zobrazí

```
Mam *rad* pivo!
Mam *rad* vino!
```

Zatím se to dost podobá metodám tak je známe, takže pojďme zkusit něco jiného, vložme blok do nějaké metody.

Vkládání bloků do metod

Example: proc3.rb

```
def osloveni rec
   puts 'Všichni se prosím ztište, chci vám říct něco důležitého'
```

```
   rec.call
   puts 'Tak to bylo vše, děkuji'
end

ahoj = Proc.new do
   puts 'Dobrý den'
end

nashledanou = Proc.new do
   puts 'Mějte se pěkně'
end

osloveni ahoj
puts
osloveni nashledanou
```

zobrazí

```
Všichni se prosím ztište, chci vám říct něco důležitého
Dobrý den
Tak to bylo vše, děkuji

Všichni se prosím ztište, chci vám říct něco důležitého
Mějte se pěkně
Tak to bylo vše, děkuji
```

V příkladu jsme vytvořili dva bloky *ahoj* a *nashledanou*. Vytvořené bloky se zavolají pomocí metody *call*.

Metody vracející *Proc*

Nyní budeme provádět následující "kouzla".

Example: proc4.rb

```
def vypocet proc1, proc2
   Proc.new do |x|
     proc2.call(proc1.call(x))
   end
end
```

```
umocnit = Proc.new do |x|
    x * x
end

secist = Proc.new do |x|
    x + x
end

secistPakUmocnit = vypocet secist, umocnit
umocnitPakSecist = vypocet umocnit, secist

puts secistPakUmocnit.call(5)
puts umocnitPakSecist.call(5)
```

zobrazí

```
100
50
```

V příkladu jsme vytvořili blok *umocnit* a *secist*, dál metodu *vypocet*, které předáváme dva parametry (třídy *Proc*) a ty v této metodě spustíme jejich jménem tak, jak jsme zvyklí.

Pokud se vám nyní ježí vlasy hrůzou, je to normální. Buď si zvyknete, nebo Ruby zavrhnete. ;-)

Vytvoření bloku z metody

Následující příklad je trochu záludný a tak si ho dobře promyslete.

Example: proc5.rb

```
class Array

def polozky &neco
   jeSude = true

   self.each do |objekt|
```

```
        if jeSude
          neco.call objekt
        end

       jeSude = (not jeSude)
    end
end

end

['jablko', 'hruška', 'švestka', 'pomeranč'].polozky do |ovoce|
   puts 'Takže vy máte rádi ' + ovoce
end
```

zobrazí

```
Takže vy máte rádi jablko
Takže vy máte rádi švestka
```

Program vybere z pole položky umístěné na místě 0, 2, 4, … v poli a vypíše je. Volání *neco.call* nás přenese do hlavního programu a provede vypsání textu na obrazovku. Všimněte si, že volaný text je vymezen direktivou *do* a *end* tak jak jsme zvyklí. Takto vytvořený blok je nutno umístit na závěr všech parametrů metody a vždy mu předchází symbol *&*.

Pokud není výše popsaná technika jasná, podívejte se na další příklad a k tomuto se vraťte později.

Example: mereni_casu.rb

```
def zpracovani popis, &blok
    zacatek = Time.now

    blok.call

    trvani = Time.now - zacatek

    puts popis + ':  ' + trvani.to_s + 's.'
end

zpracovani '25 000 sčítání' do
```

```
  cislo = 1

  25000.times do
     cislo = cislo + cislo
  end

  puts 'Číslo má ' +  cislo.to_s.length.to_s + ' míst.'
end

zpracovani '1 000 000 sčítání' do
  cislo = 1

  1000000.times do
     cislo = cislo + 1
  end

  puts 'Číslo má ' + cislo.to_s.length.to_s + ' míst.'
end
```

zobrazí např.

```
Číslo má 7526 míst.
25 000 sčítání:  0.221906s.
1 000 000 sčítání:  1.183532s.
```

Délku druhého čísla si raději nevypisujte, jinak byste mohli ztratit některé iluze o počítačích… Matematicky řečeno - vzniklé číslo je strašně moc dlouhé.

Vlastní příklad si myslím nepotřebuje další vysvětlení, nejen že se dobře čte, dobře i funguje a tak by měl správný program vypadat.

Platí to i opačně, co je složité přečíst, to je složité i napsat. ;-)

Příklady 10

- Napište metodu, která vezme nějaký blok a zavolá ho každou hodinu, která dnes uplynula. Pro zjištění aktuální hodiny můžete použít funkci *Time.now.hour*.

A to je konec.

Version 1.0
Last updated 2008-11-29 17:42:51 CEST