Ruby tutoriál

Jaromír Hradílek / **Blackened**

~

oficiální homepage: *http://blackened.wz.cz*
verze: 11.8.2005

Obsah:

1. ÚVOD

1.1 Co to je Ruby?

Přes všechny své kvality není Ruby v našich končinách až tak známá a je tedy dost možné, že jste o ní dosud neslyšeli, nebo slyšeli jen velmi málo. Co je tedy Ruby?

Ruby je interpretovaný skriptovací jazyk. Díky své jednoduché syntaxi je poměrně snadný k naučení, přesto však dostatečně výkonný, aby dokázal konkurovat svým známějším bratříčkům Pythonu a Perlu. Na rozdíl od nich je však plně objektově orientovaný (doslova vše v Ruby je objekt).

1.2 K čemu je to dobré?

Stejně jako jiné skriptovací jazyky, jakými jsou Perl nebo Python, je oblast použití Ruby poměrně široká. Asi nejčastější využití najde při psaní skriptů pro usnadnění každodenní práce. Díky své čisté a přehledné syntaxi a naprosté objektovosti se však hodí i k psaní velkých projektů, programování serverů, ba dokonce GUI aplikací.

1.3 Jak to vypadá?

Jelikož vás určitě zajímá, jak vypadá Ruby "v akci", pojďme si ukázat pár zdrojových kódů. Začněme už tradičním programem "Hello world":

```
puts "Ahoj svete!"
```

Uznávám, že tento příklad je tak jednoduchý, že nám toho o struktuře nic moc nepoví. Přejděme tedy ke složitějšímu příkladu:

```
def fib(n)
  a, b = 0, 1
  while b < n
    print b, " "
    a, b = b, a+b
  end
end

fib(100)
```

Zde jsme si definovali metodu `fib`, jež nám vypíše Fibonacciho řadu. Na konci pak tuto metodu zavoláme s parametrem 100, čímž říkáme, že chceme vypsat řadu všech čísel menších než 100. Kdybychom program spustili, získali bychom následující výstup:

```
1 1 2 3 5 8 13 21 34 55 89
```

Jelikož je ale Ruby plně objektový jazyk, pojďme si do třetice ukázat také nějaké objekty:

```
class Divka
  def initialize(jmeno, vek)
    @jmeno = jmeno
    @vek = vek
  end
  def to_s
    "Jmeno:\t#{@jmeno}\nVek:\t#{@vek}"
  end
  attr_reader :jmeno, :vek
end

mojeDevce = Divka.new('Tereza', 18)
```

```
puts mojeDevce.to_s
puts mojeDevce.jmeno
puts mojeDevce.vek
```

Spustíme-li, program, na výstupu dostaneme:

```
Jmeno:  Tereza
Vek:    18
Tereza
18
```

1.4 Trocha historie

Stvořitelem Ruby je jediný člověk - Yukihiro Matsumoto, známý také pod přezdívkou 'Matz'. Ten jako zastánce objektově orientovaného programování hledal v první polovině 90. let skriptovací jazyk, který by mu v tomto vyhovoval. Avšak Perl mu připadal v té době málo výkonný a Python zase nebyl až tak objektový, jak by chtěl. A tak se rozhodl, že vytvoří vlastní jazyk. Práce na něm započaly někdy v roce 1993, první verze byla uveřejněna v roce 1995. Matz svůj jazyk pojmenoval po rubínu - Ruby.

Dnes je Ruby asi nejšířeji používaná v zemi svého původu - v Japonsku. Výraznému rozšíření do světa dlouhou dobu bránila absence kvalitní dokumentace v anglickém jazyce. Dnes už je naštěstí anglických materiálů poměrně dost a tak nám nic nebrání pustit se do objevování jejích tajemství.

1.5 O tomto dokumentu

Tento dokument si neklade za cíl pokrýt všechny možnosti Ruby, ani neslouží jako referenční příručka. Jeho účelem je dát čtenáři potřebné základy, z nichž pak může vycházet při studiu dalších materiálů.

Tento tutoriál nepředpokládá žádné předchozí zkušenosti s programováním, ačkoli tyto jsou pochopitelně výhodou. Předpokládá se nicméně znalost konkrétního operačního systému na uživatelské úrovni a schopnost práce s příkazovou řádkou (shellem). Při psaní jsem se snažil o maximální srozumitelnost s velkým množstvím příkladů. Po jeho přečtení byste měli být schopni pracovat s prostředím Ruby, psát jednoduché skripty a chápat objektově orientovaný přístup.

Najdete-li při čtení tohoto dokumentu v textu faktické chyby nebo obtížně srozumitelné pasáže, dejte mi prosím vědět na e-mail *lord.blackened@seznam.cz* Jakýkoli ohlas či kritika jsou pochopitelně vítány také.

1.6 Použité materiály

K vytvoření tohoto tutoriálu jsem použil znalostí získaných z následujících pramenů:

- Ruby User's Guide (*http://www.rubyist.net/%7Eslagell/ruby/index.html*)
- Ruby 1.6 Built-in library Reference (*http://www.rubycentral.com/ref/index.html*)
- Programming Ruby: the Pragmatic Programmer's Guid (*http://www.rubycentral.com/book*)
- Ruby FAQ (*http://www.rubygarden.org/faq/dispatch.cgi?controller=main&action=index*)

2. INSTALACE

2.1 Systémy unixového typu

Pravděpodobně nejširšího uplatnění dosáhne Ruby v Linuxu a dalších systémech unixového typu. Pro mnoho distribucí existují připravené binární balíčky, jejichž instalace je asi nejsnazší.

V systému *Debian GNU/Linux* lze aktuální ruby nainstaloval přes program `apt-get`:

```
root@debian:~# apt-get install ruby
root@debian:~# apt-get install irb
```

`irb` je interaktivní prostředí pro Ruby a v následujících kapitolách je budeme hojně využívat, proto ho doporučuji sehnat a nainstalovat také.

V systému *Fedora Core* slouží k instalaci balíků nástroj `yum`:

```
root@fedora:~# yum install ruby
root@fedora:~# yum install irb
```

Ohlásí-li `yum` chybu, že balík nebyl nalezen, pravděpodobně nemáte v konfiguraci `yumu` nastavený žádný repozitář s balíky Ruby. Doporučuji vyhledat balík na stránkách *www.fedoratracker.org* a příslušný repozitář do souboru `/etc/yum.conf` přidat.

Nemáte-li možnost instalovat binární balík (třeba proto, že pro vaši distribuci neexistuje) a nebo nechcete, je zde možnost si Ruby zkompilovat ze zdrojových kódů.

Ze stránek *http://www.ruby-lang.org* si stáhněte nejnovější verzi (v mém případě 1.8.2). Pak se přesuňte do adresáře se staženým souborem a zadejte:

```
blackened@debian:~/download$ tar xvfz ruby-1.8.2.tar.gz
```

Tím komprimovaný balík rozbalíte. Připomínám, že jméno balíku se může lišit dle aktuální verze. Nyní se přesuňte do vytvořeného adresáře a spusťte konfigurační skript:

```
blackened@debian:~/download$ cd ruby-1.8.2
blackened@debian:~/download/ruby-1.8.2$ ./configure
```

Po úspěšném ukončení výpisu spusťte kompilaci:

```
blackened@debian:~/download/ruby-1.8.2$ make
```

Toto může chvíli trvat. Jakmile získáte opět prompt, zbývá už jen Ruby nainstalovat do systému. K tomu však potřebujete superuživatelská práva (práva uživatele root). Nemáte-li ve vašem systému přístup k účtu root, konzultujte instalaci se svým správcem systému. Jinak se příkazem `su` přepněte na uživatele `root` a přesuňte se zpět do adresáře se zdrojovými kódy. Pak už jen stačí:

```
root@debian:/home/blackened/download/ruby-1.8.2# make install
```

Nyní byste měli mít vše potřebné v systému nainstalované. Pro kontrolu zkuste spustit z příkazové řádky program `ruby`.

2.2 Microsoft Windows

Asi nejlepší je použití Ruby v kombinaci s programem *Cygwin*. Cygwin sám však přesahuje rámec tohoto tutoriálu a nebudeme se jím tedy dále zabývat. Namísto toho se podíváme na projekt *One-Click Ruby Installer*.

Ze stránek *http://rubyinstaller.rubyforge.org* stáhneme nejnovější stabilní verzi instalátoru (v mém případě soubor `ruby182-15.exe`). Soubor má okolo 15MB, takže jeho stažení může chvíli trvat. Po dokončení stažení program spustíme a necháme se provést instalací.

A co že všechno s tímto balíkem dostaneme?

- dokumentaci k Ruby, včetně vynikající knihy *Programming Ruby*
- RubyGems - správce balíků pro instalaci, update a odinstalaci rozšíření, knihoven a aplikací Ruby
- FreeRIDE - grafické vývojové prostředí
- Open SSL
- knihovny Tcl/Tk rozhraní, umožňující nám psát na Tk založené GUI aplikace
- knihovny FOX
- editor SciTe

Kompletní instalace zabírá okolo 42 MB. Kteroukoli z výše uvedených součástí můžete odmítnout instalovat a snížit tak nároky na diskový prostor, doporučuji však přivřít oči a nainstalovat raději všechno. Vyhnete se pak pozdějším problémům (např. program FreeRIDE potřebuje ke své činnosti rozhraní FOX). Po dokončení instalace pak můžeme kteroukoli z nainstalovaných součástí spustit z *Nabídky start*. Samotný program `ruby` spustíme z příkazového řádku:

```
C:\>ruby
puts "Ahoj, svete!"
^Z
Ahoj, svete!

C:\>
```

Symbol `^Z` reprezentuje stisk kláves [CTRL]+[Z]. Interaktivní interpretr pak spustíme:

```
C:\>irb
irb(main):001:0> puts "Miluji Ruby!"
Miluji Ruby!
=> nil
irb(main):002:0> exit

C:\>
```

3. SPUŠTĚNÍ

3.1 Interpretr `ruby`

Ke spouštění skriptů slouží program `ruby`. Spustíme-li jej bez jakýchkoli parametrů, dostaneme se do režimu zápisu skriptu. Po stisknutí (v unixových systémech, v DOSu to bývá `^Z`) `^D` ([CTRL]+[D]) se námi zapsaný program provede a `ruby` skončí:

```
blackened@debian:~$ ruby
3.times { puts "Ahoj, svete!" }
^D
Ahoj, svete!
Ahoj, svete!
Ahoj, svete!
blackened@debian:~$
```

Druhou možností je spustit `ruby` a program jí zadat jako parametr:

```
blackened@debian:~$ ruby -e 'puts "Ahoj, svete!"'
Ahoj, svete!
blackened@debian:~$
```

Toto je vhodné pro krátké skriptíky, které jsme schopni vměstnat do jednoho řádku, hůře se nám už bude psát složitější program. Nejčastěji používanou metodou je zapsat program do samostatného souboru a tento soubor pak předat k provedení interpretru. Mějme soubor `ahoj.rb`, obsahující následující:

```
puts "Ahoj, svete!"
```

K jeho spuštění použijeme příkazu:

```
blackened@debian:~$ ruby ahoj.rb
Ahoj, svete!
blackened@debian:~$
```

Poslední a asi nejpoužívanější možností je vytvořit z `ahoj.rb` spustitelný soubor. Upravme náš program:

```
#!/usr/bin/ruby

puts "Ahoj, svete!"
```

Řádek `#!/usr/bin/ruby` sděluje systému cestu k interpretru. Aby to fungovalo, musí být toto na úplném začátku prvního řádku souboru. Pak už stačí jen udělit souboru práva pro spuštění a spustit jej:

```
blackened@debian:~$ chmod +x ahoj.rb
blackened@debian:~$ ./ahoj.rb
Ahoj, svete!
blackened@debian:~$
```

V některých distribucích nebo v závislosti na volbě správce systému se může interpretr `ruby` nacházet i jinde (např. v adresáři `/usr/local/bin`). Lze použít i zápisu `#!/usr/bin/env ruby` - program `env` by se měl postarat o nalezení správné cesty k interpretru sám.

3.2 Interaktivní interpretr `irb`

Pro účely výuky budeme v dalším textu nejčastěji využívat interaktivního interpretru `irb`. Máte-li zkušenosti s Pythonem, Haskellem (a jeho interpretrem *Hugs98*) nebo Lua, pak již pravděpodobně interaktivní interpretry znáte a víte, jak je používat. Jejich výhodou je, že okamžitě vidíte, co se děje.

Interaktivní interpretr `irb` spustíte prostým:

```
blackened@debian:~$ irb
irb(main):001:0>
```

Na druhém řádku se již objeví jeho prompt. Krom jiného zde jsou dvě číselné hodnoty. První odpočítává řádky programu (počet již zadaných příkazů), druhá pak úroveň zanoření:

```
blackened@debian:~$ irb
irb(main):001:0> class Divka
irb(main):002:1>   def initialize(jmeno, vek)
irb(main):003:2>     @jmeno = jmeno
irb(main):004:2>     @vek = vek
irb(main):005:2>   end
irb(main):006:1>   attr_reader :jmeno, :vek
irb(main):007:1> end
=> nil
irb(main):008:0>
```

Ještě podotknu, že údaj za => označuje návratovou hodnotu, v tomto případě prázdnou hodnotu `nil`.

```
blackened@debian:~$ irb
irb(main):001:0> puts "Ahoj, svete!"
Ahoj, svete!
=> nil
irb(main):002:0>
```

Jak vidíte, výsledek naší činnosti se nám okamžitě zobrazí. Interaktivní interpretr ukončíte příkazem `exit`, nebo stiskem `^D` ([CTRL]+[D]):

```
irb(main):002:0> exit
blackened@debian:~$
```

4. ZAČÍNÁME

4.1 Komentáře

Jednou z velmi důležitých, avšak programátory často opomíjených součástí zdrojového kódu programu jsou komentáře. Komentář je část kódu, která se neprovádí (interpretr ji přeskočí, v případě kompilovaných jazyků je ignorována kompilátorem). A k čemu slouží?

Jak už ze samotného názvu vyplývá, komentáře slouží k usnadnění orientace při čtení zdrojového kódu programu. Je totiž známá věc, že po nějaké době má i programátor problémy vyznat se ve svém vlastním výtvoru. Vysvětlení je logické - když člověk vytváří program, má v hlavě spoustu věcí (k čemu slouží ta a ta metoda, jak pracuje, ...). Jak se program rozrůstá, programátor se koncentruje na určité části programu a ty ostatní odsouvá na pozadí svého povědomí. Když se pak k nim po nějaké době vrací, může zabrat poměrně dost času, než se v nich znovu zorientuje. Toto usnadňují právě komentáře.

Účelem komentářů je zpřehlednit zdrojový kód programu. Je proto třeba nakládat s nimi opatrně a najít správnou míru: příliš mnoho komentářů naopak čitelnost kódu zhoršuje. Správný komentář by měl být stručný a jasný a měl by dokumentovat co konkrétní část dělá, nikoli jak to dělá. Proč? Představte si, že programátor stojí před úkolem seřadit posloupnost celých čísel vzestupně. Existuje mnoho řadících algoritmů a ne všechny jsou stejně rychlé nebo náročné na paměť počítače. Časem programátor zjistí, že jím zvolený způsob řazení je příliš pomalý a rozhodne se jej tedy přepsat. Ovšem (a to se stává opravdu často) zapomene už přepsat komentář. Ten se tak stává pro případného čtenáře naopak matoucím a nejspíš napáchá víc škody, než užitku.

Ruby rozeznává dva druhy komentářů. Nejčastěji používaný je klasický jednořádkový komentář. Ten je zahájen znakem # a říká, že vše, co za tímto znakem následuje až do konce řádku je komentář a má být interpretrem ignorováno:

```
irb(main):020:0> # Toto je komentar.
irb(main):021:0* # Vse, co nasleduje za znakem '#' je interpretrem ignorovano:
irb(main):022:0* # puts "Ahoj, svete!"
```

Komentář pochopitelně nemusí začínat na začátku řádku a mohou mu bezprostředně předcházet příkazy:

```
irb(main):027:0> puts "Ahoj, svete!" # vypise pozdrav
Ahoj, svete!
=> nil
irb(main):028:0>
```

Druhým druhem komentářů je takzvaný blok. Ten začíná příkazem `=begin` a končí `=end`. Cokoli mezi nimi je bráno jako komentář:

```
irb(main):028:0> =begin
irb(main):029:0= Toto je blokovy komentar, presahujici libovolny pocet radku.
irb(main):030:0= Obzvlaste se hodi pro rozsahlejsi sdeleni, jakymi jsou
irb(main):031:0= kuprikladu informace o autorskych pravech nebo licence,
irb(main):032:0= pod kterou je kod distribuovan.
irb(main):033:0= =end
irb(main):034:0>
```

4.2 `irb` jako kalkulátor

Jak už jsem uvedl výše, pro účely výuky budeme používat interaktivní interpretr. Spusťte si tedy `irb`:

```
blackened@debian:~$ irb
irb(main):001:0>
```

Jednou z vlastností `irb`, podobně jako je tomu s interaktivními interpretry jiných jazyků, jakými jsou Python nebo Haskell, je možnost jeho využití jako příručního kalkulátoru:

```
irb(main):001:0> 27 + 15
=> 42
irb(main):002:0>
```

K dispozici máme všechny základní operace:

```
irb(main):002:0> 7 - 20
=> -13
irb(main):003:0> 3 * 6
=> 18
irb(main):004:0> 9 / 3
=> 3
irb(main):005:0>
```

Nutno podotknout, že dělení celých čísel nám dává celočíselný výsledek. K vypsání zbytku po celočíselném dělení slouží operátor `%`:

```
irb(main):005:0> 7 / 3
=> 2
irb(main):006:0> 7 % 3
=> 1
irb(main):007:0>
```

Čísla můžeme také umocňovat, a to pomocí operátoru `**`:

```
irb(main):007:0> 2**4
=> 16
irb(main):008:0>
```

Práce s celými čísly je možná hezká, řeknete si, ale celá čísla jsou v reálném životě poměrně vzácný úkaz. Jak to jen přinutím pracovat s reálnými čísly? Podívejte se na následující příklad:

```
irb(main):008:0> 7.0 / 3
=> 2.33333333333333
irb(main):009:0>
```

Zapsáním čísla 7 jako 7.0 explicitně říkáme, že chceme pracovat s reálnými čísly. Jednoduché, že?

Je pravda, že málokdo by si vzal na operaci 7 / 3 kalkulačku. V Ruby však lze velice snadno zapisovat i složitější konstrukce, na jednotlivé operace jsou pak uplatněna klasická matematická pravidla (násobení má přednost před sčítáním atp.):

```
irb(main):009:0> 2 + 2 * 2
=> 6
irb(main):010:0>
```

Při zápisu složitých konstrukcí nám také nic nebrání v použití závorek. To je nejen přehlednější, ale závorky mnohdy i usnadňují čitelnost zápisu:

```
irb(main):010:0> 1.7*(3.0+(12.0-7.0)/4.0)
=> 7.225
irb(main):011:0>
```

4.3 Proměnné

Proměnné jsou jedním ze stavebních kamenů imperativních programovacích jazyků a pravděpodobně je již znáte z algebry. Oproti jazykům, jakými jsou C++ nebo Pascal, není třeba proměnné deklarovat, ani explicitně uvádět její typ. Proměnná vzniká v okamžiku, kdy je jí přiřazena nějaká hodnota:

```
irb(main):011:0> PI = 3.14
=> 3.14
```

```
irb(main):012:0>
```

Takto jsme vytvořili proměnnou (resp. konstantu, viz dále) `PI`, která obsahuje reálnou hodnotu 3.14. Stejně jako v matematice ji nyní můžeme použít:

```
irb(main):012:0> PI * 5**2
=> 78.5
irb(main):013:0>
```

Je však třeba si dát pozor na to, aby proměnná skutečně existovala, použití neexistující proměnné (takové, jíž nebyla předtím přiřazena žádná hodnota) vede k chybě:

```
irb(main):013:0> 2 * PI * r
NameError: undefined local variable or method `r' for main:Object
        from (irb):13
irb(main):014:0>
```

Jak už jsem řekl, proměnné nejsou předem typovány. Jejich typ se mění s aktuální hodnotou:

```
irb(main):014:0> a = 42
=> 42
irb(main):015:0> a.class
=> Fixnum
irb(main):016:0> a = 2.75
=> 2.75
irb(main):017:0> a.class
=> Float
irb(main):018:0> a = 'Ahojky!'
=> "Ahojky!"
irb(main):019:0> a.class
=> String
irb(main):020:0>
```

Jak náš program poroste, budeme v něm používat čím dál více proměnných. Stejně jako komentáře i vhodný název proměnných přispívá k lepší čitelnosti kódu:

```
a = 2 * b * c
```

Tento zápis je sice srozumitelný, ale nijak nám neříká nic o operaci, kterou provádíme. Vhodně pojmenované proměnné dokumentují operaci samy:

```
obvodKruhu = 2 * PI * polomer
```

Přehlednější, že? Nabízí se otázka, jak zacházet s víceslovnými názvy proměnných. Běžně se používají dva druhy zápisu:

```
viceslovnyNazev
viceslovny_nazev
```

Je na vás, který způsob vám vyhovuje lépe. Je však dobré si osvojit jeden způsob a důsledně se ho držet, i to totiž přispívá k lepší srozumitelnosti.

4.4 Konstanty

Už jsem trochu nakousl, že naše `PI` je ve skutečnosti konstanta. Jaký je rozdíl mezi klasickou proměnnou a konstantou? Vysvětlení je nasnadě: konstanta nabývá jedné hodnoty při svém vytvoření a ta se již dále nemění, ba jakákoli změna její hodnoty je nežádoucí. Příkladem budiž naše `PI`. To má pevnou hodnotu 3.14, jakákoli jiná hodnota by v našich dalších výpočtech vedla k chybám.

Z pohledu jazyka Ruby spočívá rozdíl mezi konstantami a klasickými proměnnými v jejich názvu:

- název proměnné začíná znaky `a-z` nebo _ (podtržítko)

- název konstanty začíná znaky A-Z

Takto docílíme nejen výrazného optického rozdílu, ale i odlišného přístupu interpretru:

```
irb(main):020:0> a = 3.14  # promenna
=> 3.14
irb(main):021:0> a = 3
=> 3
irb(main):022:0> PI = 3.14  # konstanta
=> 3.14
irb(main):023:0> PI = 3
(irb):23: warning: already initialized constant PI
=> 3
irb(main):024:0>
```

Nová hodnota sice byla konstantě přiřazena, ale změnu provázelo varování, což nám později při ladění programu ulehčí hledání, kde že se stala chyba.

4.5 Lokální proměnné

To, co jsme dosud tiše používali, jsou ve skutečnosti lokální proměnné. To neznamená nic jiného, než že existují jen v rámci určité metody nebo třídy, jak ukazuje následující příklad:

```
irb(main):024:0> def pozdravuj
irb(main):025:1>   pozdrav = 'Nazdarek!'
irb(main):026:1>   puts pozdrav
irb(main):027:1> end
=> nil
irb(main):028:0> pozdrav = 'Ahoj.'
=> "Ahoj."
irb(main):029:0> puts pozdrav
Ahoj.
=> nil
irb(main):030:0> pozdravuj
Nazdarek!
=> nil
irb(main):031:0> puts pozdrav
Ahoj.
=> nil
irb(main):032:0>
```

Detaily definice metody se zatím nezabývejme, co je důležité je, že vidíme na dvou místech přiřazení různých hodnot do proměnné `pozdrav` (dle číslování `irb` na řádcích 25 a 28). Poté postupně vypisujeme hodnotu proměnné její hodnotu (řádky 29 až 31). Jak vidíme, proměnné `pozdrav` existují lokálně na dvou místech: jedna v rámci hlavního programu a druhá ve funkci `pozdravuj` a jsou na sobě nezávislé.

Pokud jste z výše uvedeného příkladu zmatení, nedělejte si starosti a klidně jej pro tuto chvíli pusťte z hlavy. Více pochopíte, až přejdeme k definicím metod a tříd.

Lokální proměnné začínají malým písmenem a-z nebo znakem _ (podtržítkem).

4.6 Globální proměnné

Globální proměnné jsou pozůstatkem z dob procedurálního programování. Oproti lokálním proměnným existuje v programu vždy jen jedna globální proměnná určitého jména. Globální proměnné začínají znakem $.

```
irb(main):032:0> def pozdravuj
irb(main):033:1>   $pozdrav = 'Nazdarek!'
irb(main):034:1>   puts $pozdrav
irb(main):035:1> end
```

```
=> nil
irb(main):036:0> $pozdrav = 'Ahoj.'
=> "Ahoj."
irb(main):037:0> puts $pozdrav
Ahoj.
=> nil
irb(main):038:0> pozdravuj
Nazdarek!
=> nil
irb(main):039:0> puts $pozdrav
Nazdarek!
=> nil
irb(main):040:0>
```

Mírně modifikovaná ukázka z předchozí kapitolky ukazuje chování globálních proměnných. Proměnné `$pozdrav` je nejprve přiřazena hodnota `'Ahoj.'` a následně pro kontrolu vypsána. Pak je zavolána metoda pozdravuj, v níž je hodnota proměnné změněna na `'Nazdarek!'`. Na řádku 39 je již vypsána změněná hodnota.

Použití globálních proměnných se možná může zdát snadné a logické (vždyť přece chci mít svou uloženou hodnotu přístupnou odevšad, ne?), nicméně je také velmi nebezpečné a proto se doporučuje s nimi šetřit, ba nejlépe nepoužívat je vůbec. Je až příliš snadné globální proměnnou nechtěně přepsat a zapříčinit tak chybné chování programu. Odhalit výskyt chyby v rozsáhlém zdrojovém kódu složité aplikace může zabrat hodiny ba dny, které bychom místo toho mohli věnovat tvůrčí činnosti. Jak uvidíme dále, předávání hodnot pomocí parametrů je nejen elegantnější, ale usnadňuje i lokalizaci případných chyb.

4.7 Více o přiřazení

Zatím jsme používali jen jednoduché intuitivní přiřazení:

```
irb(main):041:0> polomer = 5.4
=> 5.4
irb(main):042:0>
```

Jistě jste už pochopili, že na levé straně rovnítka je název proměnné, na pravé straně pak samotná hodnota. Krom prostých číselných hodnot je však možné přiřazovat i výsledky matematických operací:

```
irb(main):042:0> obsah = 3.14 * 5.4**2
=> 91.5624
irb(main):043:0>
```

Je také možné na levé straně použít jiné proměnné (vzpomínáte na `PI`?):

```
irb(main):043:0> obsah = PI * polomer**2
=> 91.5624
irb(main):044:0>
```

Můžeme definovat i několik proměnných naráz:

```
irb(main):044:0> a = b = c = 4
=> 4
irb(main):045:0> a
=> 4
irb(main):046:0> b
=> 4
irb(main):047:0> c
=> 4
irb(main):048:0>
```

Lze dokonce přiřadit na jednom řádku několika proměnným různé hodnoty:

```
irb(main):048:0> a, b = 5, 2.3
=> [5, 2.3]
irb(main):049:0> a
=> 5
irb(main):050:0> b
=> 2.3
irb(main):051:0>
```

Dost často se také stává, že na levé straně stojí shodná proměnná:

```
irb(main):051:0> pocet = 1
=> 1
irb(main):052:0> pocet = pocet + 1
=> 2
irb(main):053:0>
```

Matematici teď určitě kroutí nevěřícně hlavou. Musíte si však uvědomit, že rovnítko zde plní funkci přiřazení, nikoli porovnání (k tomu slouží operátor ==, ale o tom až později).

Jelikož změna hodnoty proměnné v závislosti na původní hodnotě je poměrně častým úkolem, nabízí Ruby zkrácení ve formě operátorů '+=', '-=', '*=', '/=' a '%=':

```
irb(main):053:0> a = 9
=> 9
irb(main):054:0> a += 1  # ekvivalentni k: a = a + 1
=> 10
irb(main):055:0> a -= 3  # ekvivalentni k: a = a - 3
=> 7
irb(main):056:0> a *= 4  # ekvivalentni k: a = a * 4
=> 28
irb(main):057:0> a /= 2  # ekvivalentni k: a = a / 2
=> 14
irb(main):058:0> a %= 9  # ekvivalentni k: a = a % 9
=> 5
irb(main):059:0>
```

V příští kapitole se podíváme hlouběji na datové typy a ukážeme si, jak se pracuje s řetězci, poli a hashi.

5. DATOVÉ TYPY

5.1 Dynamické datové typy

Na rozdíl od běžných nízkoúrovňových jazyků typu C/C++ má Ruby dynamické typování. Co to znamená? Už jsem uváděl, že proměnná není deklarována a nabývá typu až podle přiřazené hodnoty. Tomuto se říká dynamické typování a je společné i mnoha jiným skriptovacím jazykům (např. Pythonu). Pojďme se nyní podívat blíže na základní datové typy v Ruby.

5.2 Fixnum a Bignum

Fixnum a Bignum jsou označení pro celá čísla. Pokud máte již předchozí zkušenosti s programováním, nejspíš je znáte pod označením Integer a Long Integer. Jaký je mezi nimi rozdíl? Fixnum je určen pro menší čísla a zabírá méně místa v paměti. Chceme-li však velká čísla, pak přijde na řadu Bignum, který může nabývat teoreticky nekonečných hodnot (omezením je pochopitelně dostupná velikost vaší paměti RAM). Jak se s nimi pracuje? Jednoduše. Interpretr sám pozná, který typ má použít:

```
irb(main):001:0> a = 10
=> 10
irb(main):002:0> a.class
=> Fixnum
irb(main):003:0> a = 10_000_000_000
=> 10000000000
irb(main):004:0> a.class
=> Bignum
irb(main):005:0>
```

Jelikož čitelnost větších čísel je obtížná, umožňuje Ruby oddělovat řády pomocí znaku _ (podtržítko). Podtržítka jsou interpretrem ignorována (vidíme, že návratová hodnota je 10000000000), programátoři je však jistě ocení.

Druhou zajímavou věcí je metoda `class` (kterou jsem už tiše vpašoval do příkladu v minulé kapitole). O metodách si toho řekneme více, až přejdeme k objektově orientovanému přístupu; zatím nás zajímá jen výsledek a tím je právě typ proměnné.

5.3 Float

S typem Float jsme se už taky setkali a označuje nám reálná čísla (čísla s plovoucí desetinnou čárkou => angl. *floating point*). Stejně jako ve většině programovacích jazyků je místo u nás běžné čárky používána k oddělení desetinných míst tečka:

```
irb(main):005:0> a = 3.14159
=> 3.14159
irb(main):006:0> a.class
=> Float
irb(main):007:0>
```

5.4 String

String, neboli česky řetězec je už poněkud komplikovanější. Také už jsme jej už několikrát propašoval do předchozích příkladů. Řetězec je skupina po sobě jdoucích znaků. Běžně se řetězce vytváří uzavřením skupiny znaků buď mezi apostrofy (`'`), nebo do uvozovek (`"`):

```
irb(main):007:0> pozdrav = 'Ahoj!'
=> "Ahoj!"
irb(main):008:0> den = "Streda"
=> "Streda"
irb(main):009:0>
```

Na první pohled se zdá, že je zcela jedno, jestli použijeme apostrofů, nebo uvozovek. To však není pravda. Asi první, co vás napadne, je použít apostrofy tam, kde v našem řetězci chceme mít uvozovky, a naopak:

```
irb(main):009:0> pozdrav = '"No nazdar," pravil a zatvaril se kysele.'
=> "\"No nazdar,\" pravil a zatvaril se kysele."
irb(main):010:0> titul = "Blackened's Ruby Tutorial"
=> "Blackened's Ruby Tutorial"
irb(main):011:0>
```

Je zde ještě jeden podstatnější rozdíl: v řetězci uzavřeném v uvozovkách se berou v potaz speciální znaky uvozené znakem `\` (zpětné lomítko) a vyhodnocují se výrazy uzavřené v `#{}`. V řetězci uzavřeném v apostrofech se těmto znakům nepřipisuje žádný zvláštní význam a je s nimi zacházeno jako s jakýmikoliv jinými. Více nám ukáží následující příklady:

```
irb(main):011:0> puts "Ahoj\nsvete!"
Ahoj
svete!
=> nil
irb(main):012:0> puts 'Ahoj\nsvete!'
Ahoj\nsvete!
=> nil
irb(main):013:0>
```

Příkazem `puts` se budeme detailněji zabývat v některé z dalších kapitol, prozatím nám stačí vědět, že slouží k vypsání hodnot na obrazovku.

Z příkladu je jasně vidět odlišné chování - speciální znak '`\n`' je standardním znakem konce řádku a jako takový byl v prvním případě vyhodnocen. V případě druhém se s ním zacházelo jako s běžným řetězcem.

Možná vás napadlo, jak docílit třeba vypsání znaku zpětného lomítka v řetězci s uvozovkami. Použije-li se zpětné lomítko zdvojeně, je vypsáno jako standardní znak:

```
irb(main):013:0> puts "Toto je zpetne lomitko: \\"
Toto je zpetne lomitko: \
=> nil
irb(main):014:0>
```

Zpětné lomítko totiž stejně jako dovede znakům speciální význam dávat (např. '`\n`'), umí ho i brát:

```
puts "\"No nazdar,\" pravil a zatvaril se kysele."
"No nazdar," pravil a zatvaril se kysele.
=> nil
irb(main):015:0>
```

Když se podíváte o pár příkladů zpátky na návratovou hodnotu přiřazení do proměnné `pozdrav` zjistíte, že Ruby dělá vlastně při použití apostrofů totéž.

Druhou zajímavou vlastností je vyhodnocování výrazů uzavřených v `#{}`:

```
irb(main):015:0> puts "5 x 5 = #{5*5}"
5 x 5 = 25
=> nil
irb(main):016:0>
```

Lze pochopitelně použít i proměnných:

```
irb(main):016:0> jazyk = "Ruby"
=> "Ruby"
irb(main):017:0> puts "#{jazyk} je muj nejoblibenejsi skriptovaci jazyk!"
Ruby je muj nejoblibenejsi skriptovaci jazyk!
=> nil
```

```
irb(main):018:0>
```

Kdybychom se o totéž pokusili s apostrofy, moc bychom neuspěli:

```
irb(main):018:0> puts '#{jazyk} je muj nejoblibenejsi skriptovaci jazyk!'
#{jazyk} je muj nejoblibenejsi skriptovaci jazyk!
=> nil
irb(main):019:0>
```

Avšak jeden zápis přecijen vyhodnocuje jinak, a to když chceme vypsat právě apostrof:

```
irb(main):019:0> puts 'Blackened\'s Ruby Tutorial'
Blackened's Ruby Tutorial
=> nil
irb(main):020:0>
```

Řetězce se dají velmi snadno spojovat:

```
irb(main):020:0> 'Ahoj ' + 'svete!'
=> "Ahoj svete!"
irb(main):021:0>
```

...a to i v případě proměnných typu String:

```
irb(main):021:0> jmeno = 'Bilbo'
=> "Bilbo"
irb(main):022:0> prijmeni = 'Pytlik'
=> "Pytlik"
irb(main):023:0> celeJmeno = jmeno + ' ' + prijmeni
=> "Bilbo Pytlik"
irb(main):024:0>
```

Co je však ještě zajímavější, řetězce se dají i násobit:

```
irb(main):024:0> 'la'*3
=> "lalala"
irb(main):025:0>
```

Krom toho se dají s řetězci vyvádět psí kusy:

```
irb(main):025:0> 'Ahoj, svete!'.length  # vypise delku retezce
=> 12
irb(main):026:0> 'Ahoj, svete!'.upcase  # prevede vsechna pismena na velka
=> "AHOJ, SVETE!"
irb(main):027:0> 'Ahoj, svete!'.downcase  # prevede vsechna pismena na mala
=> "ahoj, svete!"
irb(main):028:0>  'Ahoj,  svete!'.swapcase   #  zameni  velka  pismena  za  mala  a
naopak
=> "aHOJ, SVETE!"
irb(main):029:0> 'Ahoj, svete!'.reverse  # vypise retezec pozpatku
=> "!etevs ,johA"
irb(main):030:0>
```

O metodách si budeme povídat důkladněji, až přejdeme k objektově orientovanému přístupu a třídám. Vidíte však už teď, že se s nimi dá užít pořádné legrace. :-)

Použijeme-li jednu z výše uvedených metod na nějaký řetězec, jeho původní hodnota se nezmění:

```
irb(main):030:0> pozdrav = 'Ahoj, svete!'
=> "Ahoj, svete!"
irb(main):031:0> pozdrav.upcase
=> "AHOJ, SVETE!"
```

```
irb(main):032:0> pozdrav
=> "Ahoj, svete!"
irb(main):033:0>
```

Dost často by se nám ale hodilo naši změnu do proměnné uložit. Běžným způsobem se to dá provést následovně:

```
irb(main):033:0> pozdrav = pozdrav.upcase
=> "AHOJ, SVETE!"
irb(main):034:0> pozdrav
=> "AHOJ, SVETE!"
irb(main):035:0>
```

Ruby však nabízí elegantnější řešení, a tím je použití metody s vykřičníkem na konci:

```
irb(main):017:0> pozdrav.downcase!
=> "ahoj, svete!"
irb(main):018:0> pozdrav
=> "ahoj, svete!"
irb(main):019:0>
```

Hezké, ne? Pochopitelně ne všechny metody mají definovaný ekvivalent s vykřičníkem, například v souvislosti s metodou `lenght` v tom ani mnoho užitku nevidím.

5.5 Array

Typ Array bývá česky překládán jako pole a v jazycích, jako je Python nebo Haskell, je označován jako list (seznam). Pole je zvláštní datový typ, který může oproti výše uvedeným nabývat více hodnot:

```
irb(main):019:0> cisla = [1, 2, 3]
=> [1, 2, 3]
irb(main):020:0>
```

Takto jsme vytvořili proměnnou `cisla` typu Array, obsahující tři prvky typu Fixnum. Důležitou vlastností polí v Ruby je, že mohou obsahovat i prvky různých typů:

```
irb(main):020:0> pelmel = [3, "Ahoj", -12.4]
=> [3, "Ahoj", -12.4]
irb(main):021:0>
```

Pole může dokonce obsahovat další pole:

```
irb(main):021:0> pelmel = [3, "Ahoj", -12.4, cisla]
=> [3, "Ahoj", -12.4, [1, 2, 3]]
irb(main):022:0>
```

Stejně jako tomu bylo u typu String, lze mezi sebou pole sčítat, nebo je násobit:

```
irb(main):022:0> pelmel + cisla
=> [3, "Ahoj", -12.4, [1, 2, 3], 1, 2, 3]
irb(main):023:0> cisla * 3
=> [1, 2, 3, 1, 2, 3, 1, 2, 3]
irb(main):024:0>
```

Takto můžeme vytvářet seznamy nejrůznějších prvků. Ale jak s nimi pak pracovat a k jednotlivým prvkům přistupovat? Pole je vlastně seznam prvků, indexovaných od nuly - obsah naší proměnné `cisla` tedy vypadá nějak takto:

```
         +---+---+---+
hodnota: | 1 | 2 | 3 |
         +---+---+---+
index:     0   1   2
```

Chceme-li vypsat 2. prvek pole `cisla` (tedy prvek s indexem 1), uděláme to takto:

```
irb(main):024:0> cisla[1]
=> 2
irb(main):025:0>
```

Pole lze indexovat i odzadu:

```
         +---+---+---+
hodnota: | 1 | 2 | 3 |
         +---+---+---+
index:     0   1   2
index:    -3  -2  -1

irb(main):025:0> cisla[-1]
=> 3
irb(main):026:0> cisla[-3]
=> 1
irb(main):027:0>
```

Pokusíme-li se zobrazit neexistující prvek pole, bude nám vrácena prázdná hodnota `nil`:

```
irb(main):027:0> cisla[10]
=> nil
irb(main):028:0>
```

Kromě zobrazování jednotlivých prvků má Ruby i další možnosti:

```
irb(main):028:0> cisla = [1, 2, 3, 4, 5, 6]
=> [1, 2, 3, 4, 5, 6]
irb(main):029:0> cisla[0,3]  # vypise 3 prvky, pocinaje indexem 0
=> [1, 2, 3]
irb(main):030:0> cisla[0..3]  # vypise prvky s indexem 0 az 3 vcetne
=> [1, 2, 3, 4]
irb(main):031:0> cisla[0...3]  # vypise prvky s indexem 0 az 2
=> [1, 2, 3]
irb(main):032:0> cisla[-3,2]  # vypise 2 prvky, pocinaje indexem -3
=> [4, 5]
irb(main):033:0>
```

Prvky lze podle indexů i přidávat, nebo nahrazovat:

```
irb(main):033:0> jmena = ['Jaromir', 'Lukas', 'Radek']
=> ["Jaromir", "Lukas", "Radek"]
irb(main):034:0> jmena[3] = 'Jiri'
=> "Jiri"
irb(main):035:0> jmena
=> ["Jaromir", "Lukas", "Radek", "Jiri"]
irb(main):036:0>
```

V případě, že přidáme prvek na index větší než je dosavadní rozsah, jsou "chybějící" prvky nahrazeny prázdnou hodnotou (`nil`):

```
irb(main):036:0> jmena[7] = 'Martin'
=> "Martin"
irb(main):037:0> jmena
=> ["Jaromir", "Lukas", "Radek", "Jiri", nil, nil, nil, "Martin"]
irb(main):038:0>
```

Stejně jako String, má i Array spoustu užitečných metod:

```
irb(main):038:0> jmena.length  # vrati pocet prvku pole
=> 8
irb(main):039:0> jmena.reverse  # vrati pole s obracenym poradim prvku
=> ["Martin", nil, nil, nil, "Jiri", "Radek", "Lukas", "Jaromir"]
irb(main):040:0> jmena.compact  # zkrati pole o prazdne hodnoty
=> ["Jaromir", "Lukas", "Radek", "Jiri", "Martin"]
irb(main):041:0>
```

Další zajímavou možností je převod pole na řetězec a naopak:

```
irb(main):041:0> retezec = jmena.join(';')
=> "Jaromir;Lukas;Radek;Jiri;;;;Martin"
irb(main):042:0> retezec.split(';')
=> ["Jaromir", "Lukas", "Radek", "Jiri", "", "", "", "Martin"]
irb(main):043:0>
```

V závorce obou metod uvádíme řetězec, který má být použit/rozpoznán jako oddělovač prvků. Převod z řetězce na pole však není 100% - bez ohledu na prapůvodní typ jsou všechny prvky převedeny na řetězec.

K vymazání prvku z pole slouží metoda `delete`:

```
irb(main):043:0> jmena.delete("Jaromir")
=> "Jaromir"
irb(main):044:0> jmena
=> ["Lukas", "Radek", "Jiri", nil, nil, nil, "Martin"]
irb(main):045:0>
```

Pomocí správných metod lze pole využít jako frontu s filozofií fifo (*first in, first out* - první dovnitř, první ven):

```
irb(main):045:0> buffer = []  # vytvorime prazdnou frontu
=> []
irb(main):046:0> buffer.push("Jaromir")  # pridame prvek na konec fronty
=> ["Jaromir"]
irb(main):047:0> buffer.push("Lukas")  # pridame prvek na konec fronty
=> ["Jaromir", "Lukas"]
irb(main):048:0> buffer.push("Radek")  # pridame prvek na konec fronty
=> ["Jaromir", "Lukas", "Radek"]
irb(main):049:0> buffer.shift  # odebereme prvek na zacatku fronty a vratime
jeho hodnotu
=> "Jaromir"
irb(main):050:0> buffer
=> ["Lukas", "Radek"]  # na prvni misto se dostal nasledujici prvek
irb(main):051:0>
```

Další užitečné metody lze nalézt v referenci jazyka Ruby, v části o třídě Array.

5.6 Hash

Hash nebo taky asociativní pole slouží také k ukládání více hodnot. Namísto číselných indexů však používá k označení prvků takzvaných klíčů. Hash je uzavřen ve složených závorkách:

```
irb(main):051:0> udaje = {"jmeno" => "Jaromir", "prijmeni" => "Hradilek"}
=> {"jmeno"=>"Jaromir", "prijmeni"=>"Hradilek"}
irb(main):052:0>
```

Hodnoty se zadávají vždy ve dvojici `klí => hodnota` a navzájem jsou oddělené čárkou. K jednotlivým hodnotám se pak přistupuje pomocí klíče:

```
irb(main):053:0> udaje["jmeno"]
=> "Jaromir"
irb(main):054:0> udaje["prijmeni"]
```

```
=> "Hradilek"
irb(main):055:0>
```

Nový údaj lze do hashe snadno přidat:

```
irb(main):055:0> udaje["pohlavi"] = "muz"
=> "muz"
irb(main):056:0> udaje
=> {"jmeno"=>"Jaromir", "prijmeni"=>"Hradilek", "pohlavi"=>"muz"}
irb(main):057:0>
```

I typ Hash má definovanou spoustu metod:

```
irb(main):057:0> udaje.length  # vrati pocet prvku
=> 3
irb(main):058:0> udaje.keys  # vrati pole vsech klicu
=> ["jmeno", "prijmeni", "pohlavi"]
irb(main):059:0> udaje.values  # vrati pole vsech hodnot
=> ["Jaromir", "Hradilek", "muz"]
irb(main):060:0> udaje.index("Jaromir")  # vrati klic k dane hodnote
=> "jmeno"
irb(main):061:0> udaje.invert  # zameni klice s hodnotami
=> {"Hradilek"=>"prijmeni", "Jaromir"=>"jmeno", "muz"=>"pohlavi"}
irb(main):062:0>
```

Dost často by se nám hodilo zjistit, zdali náš hash obsahuje určitý klíč:

```
irb(main):062:0> udaje.has_key?("prijmeni")
=> true
irb(main):063:0>
```

Návratovou hodnotou metody je buď `true` (pravda) v případě, že daný klíč existuje, nebo `false` (nepravda) v případě, že ne. Podobně lze zjišťovat i přítomnost hodnot:

```
irb(main):063:0> udaje.has_value?("Pavel")
=> false
irb(main):064:0>
```

Z hashe lze prvky i odebírat:

```
irb(main):064:0> udaje.delete("pohlavi")
=> "muz"
irb(main):065:0> udaje
=> {"jmeno"=>"Jaromir", "prijmeni"=>"Hradilek"}
irb(main):066:0>
```

Jako parametr je metodě `delete` předán klíč mazané položky, návratovou hodnotou je její hodnota. Chceme-li smazat všechny prvky, můžeme použít metodu `clear`:

```
irb(main):066:0> udaje.clear
=> {}
irb(main):067:0> udaje
=> {}
irb(main):068:0>
```

Úplný přehled metod najdete v referenci jazyka Ruby.

6. STANDARDNÍ VÝSTUP (A VSTUP)

6.1 O čem je řeč

V předchozích kapitolách jsme probrali základní datové typy a práci s nimi. Co však od programu obvykle očekáváme (a co jsme až doposud opomíjeli) je komunikace s uživatelem. Jelikož v dalších kapitolách budeme potřebovat vidět, co děláme, rozhodl jsem se na toto místo zařadit kapitolu o výstupu na obrazovku, a to konkrétně pomocí příkazů `puts`, `print` a `printf`. V závěru si také ukážeme, jak přijímat údaje od uživatele pomocí příkazu `gets`, resp. `readline`.

6.2 `puts`

Příkaz `puts` jsme dosud běžně používali:

```
irb(main):001:0> puts "Ahoj, svete!"
Ahoj, svete!
=> nil
irb(main):002:0>
```

Příkaz vypíše to, co je mu zadáno jako parametr a výstup ukončí novým řádkem. Jelikož příkaz `puts` je ve skutečnosti metoda, lze jej zapsat i pro metody běžnějším způsobem, tedy se závorkami:

```
irb(main):002:0> puts("Ahoj, svete!")
Ahoj, svete!
=> nil
irb(main):003:0>
```

Toto se ale příliš nepoužívá a ani já nebudu výjimkou. Vypsat lze hodnoty různých typů:

```
irb(main):003:0> puts "Ahoj"  # String
Ahoj
=> nil
irb(main):004:0> puts 256  # Fixnum
256
=> nil
irb(main):005:0> puts 320_678_792_445  # Bignum
320678792445
=> nil
irb(main):006:0> puts -7.25  # Float
-7.25
=> nil
irb(main):007:0>
```

Vypsat můžeme i více hodnot naráz:

```
irb(main):007:0> puts "Prvni", "Druha", "Treti"
Prvni
Druha
Treti
=> nil
irb(main):008:0>
```

Lze pochopitelně používat výrazy a proměnné:

```
irb(main):008:0> PI = 3.14
=> 3.14
irb(main):009:0> puts PI
3.14
```

```
=> nil
irb(main):010:0> puts PI * 3**2
28.26
=> nil
irb(main):011:0> puts "Ahoj" + " svete!"
Ahoj svete!
=> nil
irb(main):012:0>
```

6.3 `print`

Příkaz `print` je `puts` velmi podobný a ve skutečnosti se liší jen tím, že po vypsání zadaného údaje sám od sebe nepřejde na nový řádek:

```
irb(main):012:0> print "Ahoj svete!"
Ahoj svete!=> nil
irb(main):013:0> puts "Ahoj svete!"
Ahoj svete!
=> nil
irb(main):014:0>
```

Toho se dá využít, chceme-li vypsat více hodnot na jeden řádek:

```
irb(main):014:0> print "V jackpotu sportky je ", 9864000, " Kc\n"
V jackpotu sportky je 9864000 Kc
=> nil
irb(main):015:0>
```

K ukončení řádku jsme použili speciálního znaku `\n`. Jelikož jsou tyto znaky pro formátování výstupu důležité, přikládám tabulku těch nejběžněji používaných:

```
zapis: | vyznam:
-------+--------------------------------
\\     | zpetne lomitko \
\'     | apostrof '
\"     | uvozovky "
\b     | krok zpet
\n     | presun na novy radek
\r     | presun na zacatek radku
\t     | tabulator
```

Použití některých z nich vysvětluje následující příklad:

```
irb(main):015:0>  print  "jmeno:\t\t",  udaje["jmeno"],  "\nprijmeni:\t",  udaje
["prijmeni"], "\n"
jmeno:          Jaromir
prijmeni:       Hradilek
=> nil
irb(main):016:0>
```

6.4 printf

Příkaz `printf` funguje stejně jako jeho jmenovec v jazycích C nebo Perl a rozšiřuje možnosti formátování:

```
irb(main):016:0> printf "PI  = %0.3f\n", 3.14159
PI  = 3.142
=> nil
irb(main):017:0>
```

Co jsme to právě udělali? Kombinace `%0.3f` je zástupná značka, která říká, že na toto místo bude vypsáno reálné číslo o 3 desetinných místech. Zástupné značky jsou pak nahrazeny údaji za řetězcem, oddělenými

čárkami. Číslo 0 říká, kolikamístný údaj očekáváme. To se hodí v případě, že chceme vypsat zarovnanou tabulku hodnot.

```
irb(main):017:0> printf "%5.2f\n%5.2f\n%5.2f\n", 3.14, 9.8, 20.645
 3.14
 9.80
20.64
=> nil
irb(main):018:0>
```

Jak vidíte, čísla jsou vyrovnána do sloupce. Zápisem `%5.2f` totiž říkáme, že výpisem bude reálné číslo o 5 znacích, z toho 2 připadnou na desetinná místa. Za jeden znak se počítá desetinná tečka.

Zástupné symboly jsou pochopitelně definované i pro jiné typy:

```
zapis: | vyznam:
-------+--------------------------------
%f     | Float
%d     | Fixnum, Bignum
%s     | String

irb(main):019:0> printf "%5.2f\n%5d\n%5s\n", 3.14159, 1024, "Ahoj!"
 3.14
 1024
Ahoj!
=> nil
irb(main):020:0>
```

6.5 `gets` (`readline`)

Může se stát (a stane se), že budeme chtít, aby náš program přijal nějaké údaje od uživatele. Nyní si ukážeme, jak takové údaje přijmout za běhu programu (spuštění s parametry se budeme zabývat v některé z příštích kapitol).

K přijetí údajů slouží příkazy `gets` a `readline`. Jejich chování je shodné a záleží na vás, který se vám líbí víc. Oba přijmou řetězec, ukončený znakem nového řádku (stiskne-li uživatel [ENTER]). Tento řetězec můžeme uložit do proměnné a pak použít, jako jakýkoli jiný řetězec:

```
irb(main):020:0> jmeno = gets
Jaromir
=> "Jaromir\n"
irb(main):021:0> puts "Vase jmeno: #{jmeno}"
Vase jmeno: Jaromir
=> nil
irb(main):023:0>
```

6.5.1 Metoda chop *(*chop!*)*

Nevýhodou je, že se vstupní řetězec uloží i se znakem nového řádku (v Unixových systémech je to '\n', ve Windows '\r\n'). Naštěstí je pro řetězce definovaná metoda `chop`, která z řetězce odstraní poslední znak:

```
irb(main):023:0> jmeno.chop
=> "Jaromir"
irb(main):024:0> jmeno
=> "Jaromir\n"
irb(main):025:0>
```

Chceme-li, aby se změna aplikovala na proměnnou (takto je pozměněna jen návratová hodnota, proměnná jméno stále obsahuje znak '\n'), použijeme zápis metody s vykřičníkem:

```
irb(main):025:0> jmeno.chop!
```

```
=> "Jaromir"
irb(main):026:0> jmeno
=> "Jaromir"
irb(main):027:0>
```

chop odstraňuje jakýkoli jeden znak:

```
irb(main):027:0> jmeno.chop
=> "Jaromi"
irb(main):028:0>
```

V případě dvojice znaků '\r\n' však odstraní oba, neboť toto je standardní označení nového řádku v systémech MS Windows:

```
irb(main):028:0> jmeno = "Jaromir\r\n"
=> "Jaromir\r\n"
irb(main):029:0> jmeno.chop
=> "Jaromir"
irb(main):030:0>
```

Tak je zaručena přenositelnost našeho programu na jinou platformu bez nutnosti jakýchkoli úprav zdrojového kódu. Znaky '\r\n' však musí následovat přesně v tomto pořadí. Při jejich prohození bude odstraněn jen poslední z nich.

Aplikovat chop lze pochopitelně ihned při čtení ze vstupu:

```
irb(main):030:0> jmeno = gets.chop
Blackened
=> "Blackened"
irb(main):031:0>
```

6.5.2 Metoda `to_i`

Říkal jsem, že příkazy gets a readline přijímají hodnotu typu String. Ale co když potřebujeme pracovat s číselnými hodnotami? K převodu proměnné typu String na celočíselný typ (Fixnum/Bignum) slouží metoda to_i. Ta pracuje následujícím způsobem:

```
irb(main):031:0> "18".to_i
=> 18
irb(main):032:0> "18:40".to_i
=> 18
irb(main):033:0> "18 hodin 40 minut".to_i
=> 18
irb(main):034:0> "Je 18 hodin.".to_i
=> 0
irb(main):035:0> "-13.27".to_i
=> -13
irb(main):036:0>
```

Metoda to_i hledá na začátku řetězce celočíselnou hodnotu. Najde-li ji tam, vrátí příslušné celé číslo, přičemž jakékoli další znaky ignoruje. Nenajde-li na začátku řetězce celé číslo, vrátí hodnotu 0. Čtení číselné hodnoty by pak mohlo vypadat takto:

```
irb(main):036:0> mocnina = gets.to_i
4
=> 4
irb(main):037:0> puts 2**mocnina
16
=> nil
irb(main):038:0>
```

6.5.3 Metoda `to_f`

Stejně jako `to_i` pro celá, slouží metoda `to_f` k převodu řetězce na reálné číslo:

```
irb(main):038:0> "3.1415".to_f
=> 3.1415
irb(main):039:0> "2e-3".to_f  # znamena 2 * 10**-3
=> 0.002
irb(main):040:0> "37.6°C".to_f
=> 37.6
irb(main):041:0> "PI=3.14".to_f
=> 0.0
irb(main):042:0>
```

Načtení reálného čísla tedy provedeme takto:

```
irb(main):042:0> polomer = gets.to_f
12.7
=> 12.7
irb(main):043:0> print "prumer = ", polomer * 2, "\n"
prumer = 25.4
=> nil
irb(main):044:0>
```

7. ŘÍDÍCÍ STRUKTURY

7.1 Úvodem

Až dosud se naše "programy" omezovaly jen na práci s proměnnými a výpisem na obrazovku. Ačkoli jsou proměnné jedním z pilířů imperativních jazyků, určitě cítíte, že jen s nimi si nevystačíme. To co nám chybí jsou řídící struktury.

Řídící struktury (jak už název napovídá) slouží k řízení toku programu a jeho větvení na základě určitých podmínek. V této kapitole si vysvětlíme, co to podmínky jsou a jak se vytvářejí a ukážeme si, jak pracovat s cykly.

7.2 Logické výrazy

Představte si, že jste rodič dítěte ve školním věku, které má v devět hodin večerku. Před devátou má dítě dovoleno si hrát, dívat se na televizi, atp. Jakmile však nastane 9 hodin, musí do postele. Vaše chování je závislé na určitých okolnostech, v našem případě na čase. A tak je tomu i v programu.

Zjištění času je ve skutečnosti logický výraz: "je počet hodin roven nebo větší devíti?" a nabývá dvou stavů - pravdy, je-li podmínka splněna, a nepravdy, pokud není. Když to rozebereme ještě více, skládá se náš logický výraz ze srovnání dvou hodnot - aktuálního času a konstanty, označující večerku.

Naši podmínku bychom si mohli přepsat do pseudokódu takto:

```
je-li pocetHodin >= Vecerka pak
  je cas ulozit dite do postele
```

Zápis `pocetHodin >= Vecerka` je logický výraz. '>=' je relační operátor.

Relační operátory slouží ke srovnání dvou hodnot a vracejí buď hodnotu `true` v případě, že je logický výraz pravdivý, nebo `false` v případě, že pravdivý není. V jazyku Ruby rozeznáváme tyto základní relační operátory (existují i další, ale těmi se budeme zabývat později):

```
operator: | pouziti: | vyznam:
----------+----------+-----------------
==        | a == b   | pravda, je-li a rovno b
!=        | a != b   | pravda, neni-li a rovno b
>         | a > b    | pravda, je-li a vetsi nez b
<         | a < b    | pravda, je-li a mensi nez b
>=        | a >= b   | pravda, je-li a vetsi nebo rovno b
<=        | a <= b   | pravda, je-li a mensi nebo rovno b
```

Všimněte si, že pro zjištění rovnosti dvou hodnot se používá zdvojeného rovnítka ==. Jednoduché = slouží jako operátor přiřazení! Další častou chybou je přehození znaků: `=!` => =<. Tento zápis je neplatný a měl by vést k oznámení chyby. Mezi jednotlivé znaky také nesmíme vložit mezeru!

Mimo tyto se běžně setkáme s třemi dalšími logickými operátory:

```
operator: | alternativne: | vyznam:
----------+---------------+-------------
&&        | and           | logicke AND
||        | or            | logicke OR
!         | not           | logicke NOT
```

Tyto operátory rozšiřují možnosti stavby logických výrazů:

```
je-li pocetHodin >= Vecerka && dite != jeVPosteli pak
  ...
```

Jelikož každá relační operace vrací hodnotu `true` nebo `false`, srovnává logický operátor AND (`&&`) tyto hodnoty. Celý výraz je pak vyhodnocen podle následujících pravidel:

```
vyraz:          | vysledna hodnota:
----------------+------------------------
true && true    | true
true && false   | false
false && true   | false
false && false  | false
```

Pro logický operátor OR:

```
vyraz:          | vysledna hodnota:
----------------+------------------------
true || true    | true
true || false   | true
false || true   | true
false || false  | false
```

Logický operátor NOT slouží k negaci výrazu a obrací jeho význam:

```
vyraz:          | vysledna hodnota:
----------------+------------------------
! true          | false
! false         | true
```

Relační operátory (==, >, !=, atd.) mají před logickými operátory &&, || a ! přednost. Díky tomu by byla v našem pseudokódu nejprve vyhodnocena pravdivost tvrzení `pocetHodin >= Vecerka a dite != jeVPosteli` a pak by se teprv na výsledné hodnoty (`true`/`false`) aplikoval logický operátor `&&`. Při složitějších konstrukcích je však vhodnější výrazy uzavřít do závorek a tím dát pořadí vyhodnocování jasně najevo (hodí se zvláště v případě, kdy chceme pořadí jinak):

```
je-li ((pocetHodin >= Vecerka) && (dite != jeVPosteli)) pak
  ...
```

Ještě podotknu, že operátory `&&`, `||`, `!` mohou být alternativně zapsány jako `and`, `or` a `not`. Záleží na vás, co se vám líbí.

7.3 `if…elsif…else…end`

Dost již bylo teorie, pojďme se nyní konečně podívat, jak funguje větvení programu v praxi. Vezměme si původní znění našeho pseudokódu a přepišme si jej do jazyka Ruby:

```
irb(main):001:0> Vecerka = 21  # konstanta
=> 21
irb(main):002:0> pocetHodin = 22  # pozde, ale preci
=> 22
irb(main):003:0> if (pocetHodin >= Vecerka) then
irb(main):004:1*   puts "Musim jit ulozit dite do postylky..."
irb(main):005:1> end
Musim jit ulozit dite do postylky...
=> nil
irb(main):006:0>
```

Tato konstrukce je ekvivalentní našemu pseudokódu. Je-li splněna podmínka (logický výraz vrátí hodnotu `true`), jsou provedeny příkazy uvnitř těla podmínky. Závorky jsou nepovinné, nepovinný je i zápis `then`, následuje-li příkaz na samostatném řádku. Stejně tak správné by byly i následující zápisy:

```
if pocetHodin >= Vecerka then puts "Musim jit ulozit dite do postylky..." end

if pocetHodin >= Vecerka
```

```
  puts "Musim jit ulozit dite do postylky..."
end
```

Osobně doporučuji používat spíše druhého způsobu zápisu, v prvním případě může dojít k snadnému opomenutí `end`. Nicméně pro jednořádkové zápisy umožňuje Ruby následující zápis s podmínkou `if` na konci:

```
irb(main):006:0> puts "1 je mensi nez 2" if (1 < 2)
1 je mensi nez 2
=> nil
irb(main):007:0>
```

Levá část před příkazem `if` je vyhodnocena jen tehdy, je-li pravdivý (`true`) výraz vpravo.

Často bychom chtěli, aby se něco provedlo v případě, že podmínka splněna nebyla:

```
irb(main):007:0> pocetHodin = 19  # abychom videli vysledek
=> 19
irb(main):008:0> if pocetHodin >= Vecerka
irb(main):009:1>   puts "Musim jit ulozit dite do postylky..."
irb(main):010:1> else
irb(main):011:1*   puts "Co davaji v televizi?"
irb(main):012:1> end
Co davaji v televizi?
=> nil
irb(main):013:0>
```

`else` znamená "jinak" a blok příkazů za ním se provede kdykoli není splněna podmínka za `if`.

Co když chceme reagovat na více podmínek? Jednou z možností by bylo zanořit další podmínku do bloku za `else`:

```
irb(main):013:0> if pocetHodin >= Vecerka
irb(main):014:1>   puts "Musim jit ulozit dite do postylky..."
irb(main):015:1> else
irb(main):016:1*   if pocetHodin == 19
irb(main):017:2>     puts "Kde je ovladac? Zacinaji zpravy!"
irb(main):018:2>   else
irb(main):019:2*     puts "Co davaji v televizi?"
irb(main):020:2>   end
irb(main):021:1> end
Kde je ovladac? Zacinaji zpravy!
=> nil
irb(main):022:0>
```

Toto řešení však elegancí zrovna neoplývá a při víceru podmínek bychom se nejspíš utopili. Mnohem lepší je použití konstrukce `elsif`:

```
irb(main):022:0> if pocetHodin >= Vecerka
irb(main):023:1>   puts "Musim jit ulozit dite do postylky..."
irb(main):024:1> elsif pocetHodin == 19
irb(main):025:1>   puts "Kde je ovladac? Zacinaji zpravy!"
irb(main):026:1> else
irb(main):027:1*   puts "Co davaji v televizi?"
irb(main):028:1> end
Kde je ovladac? Zacinaji zpravy!
=> nil
irb(main):028:0>
```

Částí `elsif` může být v podmínce libovolný počet. `if` se však smí v bloku vyskytovat jen jednou na začátku, stejně tak `else` na konci. Celý blok musí uzavírat `end`.

7.4 unless...else...end

Konstrukce `unless` je v zásadě podobná `if`. Do češtiny by se dala přeložit jako "není-li" a je ekvivalentní negaci `if`:

```
irb(main):028:0> unless a == 2
irb(main):029:1>   puts "a se nerovna 2"
irb(main):030:1> end
a se 2 nerovna!
=> nil
irb(main):031:0> if !(a == 2)  # v praxi je prehlednejsi uziti 'if a != 2'
irb(main):032:1>   puts "a se nerovna 2"
irb(main):033:1> end
a se 2 nerovna
=> nil
irb(main):034:0>
```

Od `if` se ještě liší absencí ekvivalentu k `elsif`, čímž se omezuje na pouhou konstrukci:

```
unless výraz
  příkaz
else
  příkaz
end
```

Stejně jako u `if` existuje možnost jednořádkového zápisu:

```
irb(main):034:0> puts "a se nerovna 2" unless (a == 2)
a se nerovna 2
=> nil
irb(main):035:0>
```

Příkaz na levé straně se provede v případě, že je výraz vlevo nepravdivý (`false`).

7.5 case...when...else...end

V případě, že máme jednu proměnnou a chceme ji testovat na více hodnot, bylo by použití struktury `if` neobratné. Struktura `case` nám umožňuje testovat více hodnot naráz, včetně rozsahů:

```
rb(main):035:0> pocetHodin = 16
=> 16
irb(main):036:0> case pocetHodin
irb(main):037:1> when 5 .. 7
irb(main):038:1>   puts "Dobre rano!"
irb(main):039:1> when 8 .. 11
irb(main):040:1>   puts "Hezke dopoledne."
irb(main):041:1> when 12
irb(main):042:1>   puts "Dobre poledne!"
irb(main):043:1> when 13 .. 18
irb(main):044:1>   puts "Dobre odpoledne."
irb(main):045:1> when 19 .. 21
irb(main):046:1>   puts "Dobry vecer"
irb(main):047:1> when 24, 0
irb(main):048:1>   puts "Pulnoc!"
irb(main):049:1> else
irb(main):050:1*   puts "Dobrou noc..."
irb(main):051:1> end
Dobre odpoledne.
=> nil
```

```
irb(main):052:0>
```

Jak vidíte, můžeme zadávat rozsahy hodnot pomocí zápisu `od .. do` (včetně), zadávat jednu nebo více hodnot oddělených čárkou, i použít části `else` pro případ, že nevyhovuje žádná z výše uvedených podmínek. Celá konstrukce musí být ukončena slovem `end`.

7.6 Cyklus `while`

Chceme-li, aby se nějaká část programu opakovala, použijeme tzv. cyklus. Ruby nabízí hned několik druhů cyklů, jedním z nich je i cyklus `while`, známý z jiných programovacích jazyků (namátkově C/C++, Pascal, Java). Syntaxe příkazu `while`:

```
while vyraz do
  prikaz(y)
end
```

Dokud je výraz pravdivý (vrací hodnotu `true`), jsou prováděny příkazy v těle cyklu. Psaní `do` je nepovinné, stejně dobře funguje i zápis:

```
while vyraz
  prikaz(y)
end
```

Následující příklad sice moc užitečný není (a jsou vhodnější druhy cyklů pro tento úkon), nicméně demonstruje funkci cyklu:

```
irb(main):052:0> a = 1
=> 1
irb(main):053:0> while a <= 10
irb(main):054:1>   puts a
irb(main):055:1>   a += 1
irb(main):056:1> end
1
2
3
4
5
6
7
8
9
10
=> nil
irb(main):057:0>
```

Cyklus `while` má podobně jako `if` i možnost jednořádkového zápisu:

```
irb(main):057:0> a = 1
=> 1
irb(main):058:0> a += 1 while (a <= 10)
=> nil
irb(main):059:0>
```

Příkaz vlevo od `while` se provádí do té doby, dokud je výraz vpravo pravdivý (`true`).

7.7 Cyklus `until`

Cyklus `until` je ekvivalentní zápisu:

```
while !(vyraz) do
```

```
  ...
end
```

Dokud je výraz nepravdivý (`false`), jsou prováděny příkazy v těle cyklu. Naše napočítání do desíti by se tak dalo zapsat:

```
irb(main):059:0> a = 1
=> 1
irb(main):060:0> until (a > 10)
irb(main):061:1>   puts a
irb(main):062:1>   a += 1
irb(main):063:1> end
1
2
3
4
5
6
7
8
9
10
=> nil
irb(main):064:0>
```

I cyklus `until` má jednořádkový zápis:

```
irb(main):064:0> a = 1
=> 1
irb(main):065:0> a += 1 until (a > 10)
=> nil
irb(main):066:0>
```

7.8 Cyklus `for`

Ačkoli cyklus `for` se také ve výše zmíněných jazycích vyskytuje, jeho použití se v Ruby (a třeba i v Pythonu) mírně liší. Cyklus `for` totiž slouží k procházení položek seznamu:

```
irb(main):066:0> for i in [1, "Ahoj", 3.14159] do
irb(main):067:1*   puts i
irb(main):068:1> end
1
Ahoj
3.14159
=> [1, "Ahoj", 3.14159]
irb(main):069:0>
```

Položka `do` je opět nepovinná. Počítání do desíti by s cyklem `for` vypadalo takto:

```
irb(main):069:0> for i in (1..10)
irb(main):070:1>   puts i
irb(main):071:1> end
1
2
3
4
5
6
7
8
9
```

```
10
=> 1..10
irb(main):072:0>
```

Jen pro srovnání, stejný příklad by v jazyku C vypadal takto:

```
#include <stdio.h>

int main() {
  int i;
  for (i = 1; i <= 10; i++) {
    printf("%d\n", i);
  }
  return 0;
}
```

7.9 Metoda `times`

Pro celočíselné typy je definovaná metoda `times` (česky "krát"):

```
irb(main):072:0> 3.times {
irb(main):073:1*   puts "Ahoj"
irb(main):074:1> }
Ahoj
Ahoj
Ahoj
=> 3
irb(main):075:0>
```

Jak vidíte, `3.times` říká, že se mají příkazy mezi znaky `{}` provést 3x. Naše známé počítání do desíti by pak vypadalo takto:

```
irb(main):075:0> 10.times {
irb(main):076:1*   |i|
irb(main):077:1*   puts i + 1
irb(main):078:1> }
1
2
3
4
5
6
7
8
9
10
=> 10
irb(main):079:0>
```

Cyklus si někde uvnitř pro sebe počítá od 0 do 9. Zápis |i| říká, že se má aktuální stav zapsat do proměnné `i`. K té pak přičteme 1 a vypíšeme.

7.10 Metoda each a jí podobné

K procházení položek polí jsme dosud používali cyklu `for`. Pro jednotlivé datové typy (pole, hashe, ba dokonce řetězce) existuje metoda `each` a od ní odvozené metody. Každá se chová trošku jinak v závislosti na datovém typu. Jejich zvládnutí vám ovšem usnadní práci a elegantně dosáhnete toho, co byste museli s použitím for složitě obcházet.

7.10.1 Array

Pro datový typ Array (pole) je metoda each intuitivní a funguje stejně, jako cyklus for:

```
irb(main):079:0> jmena = ["Jaromir", "Lukas", "Radek", "Jiri"]
=> ["Jaromir", "Lukas", "Radek", "Jiri"]
irb(main):080:0> jmena.each {
irb(main):081:1*   | jmeno |
irb(main):082:1*   puts "Jmeno: " + jmeno
irb(main):083:1> }
Jmeno: Jaromir
Jmeno: Lukas
Jmeno: Radek
Jmeno: Jiri
=> ["Jaromir", "Lukas", "Radek", "Jiri"]
irb(main):084:0>
```

Na prvním řádku jsme si vytvořili seznam jmen, obsahující čtyři položky. Ten pak procházíme pomocí metody each, aktuální položka je vždy načtena do proměnné jméno a pak vypsána na obrazovku.

Kromě metody each můžeme ještě pole procházet po indexech:

```
irb(main):084:0> jmena.each_index {
irb(main):087:1*   | i |
irb(main):088:1*   print i, " ", jmena[i], "\n"
irb(main):089:1> }
0 Jaromir
1 Lukas
2 Radek
3 Jiri
=> ["Jaromir", "Lukas", "Radek", "Jiri"]
irb(main):090:0>
```

7.10.2 Hash

Podobně jako tomu bylo v případě polí, i u typu Hash dochází k procházení po jednotlivých položkách, v případě hashů však vrací metoda each dvojici hodnot - klíč a jemu příslušnou hodnotu:

```
irb(main):090:0> udaje = {"jmeno" => "Jaromir", "prijmeni" => "Hradilek"}
=> {"jmeno"=>"Jaromir", "prijmeni"=>"Hradilek"}
irb(main):091:0> udaje.each {
irb(main):092:1*   | klic, hodnota |
irb(main):093:1*   puts "#{klic} => #{hodnota}"
irb(main):094:1> }
jmeno => Jaromir
prijmeni => Hradilek
=> {"jmeno"=>"Jaromir", "prijmeni"=>"Hradilek"}
irb(main):095:0>
```

Ne vždy nás však zajímají obě hodnoty. Ruby myslí i na toto, a proto definuje dvě další metody - each_key pro procházení podle klíčů a each_value pro procházení po hodnotách:

```
irb(main):095:0> udaje.each_key { | klic | puts klic }
jmeno
prijmeni
=> {"jmeno"=>"Jaromir", "prijmeni"=>"Hradilek"}
irb(main):096:0> udaje.each_value { | hodnota | puts hodnota }
Jaromir
Hradilek
=> {"jmeno"=>"Jaromir", "prijmeni"=>"Hradilek"}
irb(main):097:0>
```

7.10.3 String

Jelikož řetězec je ve skutečnosti skupina znaků, lze procházet i jej. Metoda `each` (nebo její synonymum `each_line`) bez parametrů slouží k procházení řetězců po řádcích:

```
irb(main):097:0> retezec.each { | radek | radek }
=> "Prvni\nDruhy\nTreti\nCtvrty"
irb(main):098:0> retezec.each { | radek | print "-- #{radek}" }
-- Prvni
-- Druhy
-- Treti
-- Ctvrty=> "Prvni\nDruhy\nTreti\nCtvrty"
irb(main):099:0>
```

Z ukázky je patrné, že řetězec je procházen po řádcích, přičemž znak '\n' je v řetězci `radek` zachován.

Metodě `each` lze parametricky vnutit znak nebo řetězec znaků, podle nějž má řetězec procházet. Následující příklad prochází řetězec podle vět, k rozpoznání konce věty slouží kombinace znaků `'. '`:

```
irb(main):099:0> retezec = "Dobry den, slecno Marie. Doufam, ze se vam u nas
libi."
=> "Dobry den, slecno Marie. Doufam, ze se vam u nas libi."
irb(main):100:0> retezec.each('. ') { | veta | puts veta }
Dobry den, slecno Marie.
Doufam, ze se vam u nas libi.
=> "Dobry den, slecno Marie. Doufam, ze se vam u nas libi."
irb(main):101:0>
```

Kromě toho je pro řetězce definována ještě metoda `each_byte`. Ta slouží k procházení řetězce po jednotlivých znacích:

```
irb(main):101:0> retezec = "Ahoj svete!"
=> "Ahoj svete!"
irb(main):102:0> retezec.each_byte { | znak | print znak, " " }
65 104 111 106 32 115 118 101 116 101 33 => "Ahoj svete!"
irb(main):103:0>
```

Co znamenají ta čísla? Matoda `each_byte` vrací číselnou hodnotu daného znaku dle tabulky ASCII - znak 'A' má například hodnotu 65, znak 'a' 97. Chceme-li vypsat konkrétní znak, musíme jej nejprve na převést. K převodu celočíselné hodnoty na příslušný znak slouží metoda `chr`:

```
irb(main):103:0> retezec.each_byte { | znak | print znak.chr, " " }
A h o j   s v e t e ! => "Ahoj svete!"
irb(main):104:0>
```

8. REGULÁRNÍ VÝRAZY

8.1 Co to je?

Jste-li uživatelem Linuxu nebo jiného systému unixového typu a nebojíte se příkazové řádky, pravděpodobně jste se už s regulárními výrazy setkali. Využívají jich streamové editory `ed` a `sed`, vyhledávací programy `grep` a `egrep` a spousta editorů (`vi`, `vim`) a dalších programů (`more`, `less`, `awk`). Regulární výraz je jakýsi vzor, podle nějž se vyhledává řetězec v textu a provádí se nějaká operace, obvykle náhrada za jiný řetězec, nebo extrakce určitých údajů.

Regulární výrazy jsou samy o sobě látkou dosti obsáhlou a jejich výuka není účelem tohoto dokumentu - od toho jsou zde lepší zdroje, například vynikající seriál Pavla Satrapy - Regulární výrazy [*http://www.kit.vslib.cz/~satrapa/docs/regvyr/*] (vyšlo i na portálu *www.root.cz*). Pokud jste se s nimi tedy dosud nesetkali, doporučuji prostudovat výše zmíněné materiály. Ačkoli jejich zvládnutí není pro další pokračování v jazyku Ruby nezbytné, jejich ignorací se připravíte o velmi silný nástroj.

8.2 Regulární výrazy v Ruby

V Ruby se regulární výrazy uzavírají mezi dopředná lomítka `/`. Tedy `/ruby/` je regulární výraz. Jelikož je vyhledávání implicitně case-sensitive (rozlišuje malá a velká písmena) a my chceme najít jak výskyt "Ruby", tak i "ruby", můžeme napsat:

```
/(R|r)uby/
```

Jednoduché závorky slouží k seskupování znaků. Znak `|` slouží jako logické OR. Našemu vzoru tedy vyhovuje slovo "Ruby" i "ruby" s malým i velkým písmenem. Kromě toho existují speciální symboly:

```
zapis: | vyznam:
-------+--------------------------------
[]     | definuje rozsah: [a-z] vyhovuje jakekoli
       | pismeno male (anglicke) abecedy
\w     | pismeno nebo cislo; ekvivalentni: [a-zA-Z0-9]
\W     | cokoli jineho nez pismeno nebo cislo
\s     | bily znak (tzn. mezera, tabulator, novy radek);
       | ekvivalentni: [ \t\n\r\f]
\S     | cokoli jineho nez bily znak
\d     | cislo; ekvivalentni [0-9]
\D     | cokoli jineho nez cislo
.      | jakykoli jeden znak
*      | zadne nebo libovolny pocet opakovani predchoziho
+      | jedno nebo vice opakovani predchoziho
{m,n}  | nejmene m, nejvice n opakovani predchoziho
?      | nejvice jedno opakovani predchoziho;
       | ekvivalentni: {0,1}
|      | bud vyhovuje predchozi, nebo nasledujici
()     | seskupeni
```

8.3 Operátor =~

K zjištění, zda se v řetězci vyskytuje kombinace vyhovující vzoru, existuje v Ruby operátor =~:

```
retezec =~ /regularni vyraz/
```

Ten vrací buď index, na kterém začíná první nalezený řetězec, nebo `nil` v případě, že nebyla nalezena žádná vyhovující kombinace:

```
irb(main):001:0> "Myslim, ze Ruby je vyborny jazyk." =~ /ruby/  # case-sensitive
=> nil
```

```
irb(main):002:0> "Myslim, ze Ruby je vyborny jazyk." =~ /Ruby/
=> 11
irb(main):003:0>
```

Díky tomu, že jako `false` vyhodnocuje Ruby pouze `nil` a `false` a cokoli jiného je tedy vyhodnoceno jako `true`, lze použít operátor =~ i k větvení programu:

```
irb(main):003:0> retezec = "Myslim, ze Ruby je vyborny jazyk."
=> "Myslim, ze Ruby vyborny jazyk."
irb(main):004:0> if retezec =~ /(R|r)uby/
irb(main):005:1>   puts "Diky, ze se zminujete o Ruby. :-)"
irb(main):006:1> else
irb(main):007:1*   puts ":'("
irb(main):008:1> end
Diky, ze se zminujete o Ruby. :-)
=> nil
irb(main):009:0>
```

Stejně tak můžeme použít regulárních výrazů v cyklu:

```
irb(main):009:0> retezec = gets.chop
Ruby je fajn.
=> "Ruby je fajn."
irb(main):010:0> until retezec =~ /konec/
irb(main):011:1>   puts "Rekl jsi: " + retezec
irb(main):012:1>   retezec = gets.chop
irb(main):013:1> end
Rekl jsi: Ruby je fajn.
Ruby je skvela!
Rekl jsi: Ruby je skvela!
konec
=> nil
irb(main):014:0>
```

A do třetice, regulárních výrazů lze bez obav použít i ve struktuře `case`:

```
irb(main):014:0> retezec = "Myslim, ze Python je taky zajimavy jazyk."
=> "Myslim, ze Python je taky zajimavy jazyk."
irb(main):015:0> case retezec
irb(main):016:1> when /(R|r)uby/
irb(main):017:1>   puts "Ruby je skvela!"
irb(main):018:1> when /(P|p)ython/
irb(main):019:1>   puts "Python je fajn, ale Ruby je lepsi. ;-)"
irb(main):020:1> when /(P|p)erl/
irb(main):021:1>   puts "No comment. :-P"
irb(main):022:1> else
irb(main):023:1*   puts "Nemam poneti, o cem je rec. :-O"
irb(main):024:1> end
Python je fajn, ale Ruby je lepsi. ;-)
=> nil
irb(main):025:0>
```

8.4 Metoda sub (sub!)

Jedním z nejčastějších použití regulárních výrazů je k záměně nějakého textu za jiný. Pro typ String je proto definována metoda `sub`:

```
irb(main):025:0> retezec = "Jaromir Hradilek: Ruby - tutorial"
=> "Jaromir Hradilek: Ruby - tutorial"
irb(main):026:0> retezec.sub(/\w+ \w+:/, "Blackened:")
=> "Blackened: Ruby - tutorial"
```

```
irb(main):027:0>
```

Chceme-li změnu okamžitě uložit do proměnné, na niž byla metoda aplikována, použijeme ekvivalent s vykřičníkem:

```
irb(main):028:0> retezec.sub!(/\w+ \w+:/, "Blackened:")
=> "Blackened: Ruby - tutorial"
irb(main):029:0> retezec
=> "Blackened: Ruby - tutorial"
irb(main):030:0>
```

8.5 Metoda gsub (gsub!)

Výše zmíněná metoda má jen jedu "nevýhodu" - aplikuje se pouze na první výskyt vyhovujícího řetězce:

```
irb(main):030:0> "Jaromir Jaromir Jaromir".sub(/Jaromir/, "Blackened")
=> "Blackened Jaromir Jaromir"
irb(main):031:0>
```

To se někdy hodí, jindy ne. I na to Ruby myslí, a proto existuje metoda `gsub`, která nahradí každý vyhovující řetězec:

```
irb(main):031:0> "Jaromir Jaromir Jaromir".gsub(/Jaromir/, "Blackened")
=> "Blackened Blackened Blackened"
irb(main):032:0>
```

Pochopitelně existuje ekvivalent s vykřičníkem:

```
irb(main):032:0> retezec = "Jaromir Jaromir Jaromir"
=> "Jaromir Jaromir Jaromir"
irb(main):033:0> retezec.gsub!(/Jaromir/, "Blackened")
=> "Blackened Blackened Blackened"
irb(main):034:0> retezec
=> "Blackened Blackened Blackened"
irb(main):035:0>
```

8.6 Speciální proměnná $n

Říkal jsem, že jedním z nejčastějších použití regulárních výrazů je náhrada jednoho řetězce za druhý. Druhým nejčastějším je pak extrakce údajů. Představme si, že máme konfigurační soubor v následujícím formátu:

```
title=Ruby tutorial
author=Blackened
year=2005
```

Čtením dat ze souborů se budeme zabývat později, prozatím dělejme, že máme údaje načtené do proměnné typu String:

```
irb(main):035:0> udaje = "title=Ruby tutorial\nauthor=Blackened\nyear=2005"
=> "title=Ruby tutorial\nauthor=Blackened\nyear=2005"
irb(main):036:0>
```

A nyní chceme jednotlivé údaje extrahovat a nějakým způsobem zpracovat. Následující program je ukázkou, jak toho docílit. Zároveň je opakováním některých výše probraných věcí:

```
irb(main):036:0> udaje.each_line {
irb(main):037:1*   |polozka|
irb(main):038:1*   case polozka
irb(main):039:2>   when /title=(.*)/
irb(main):040:2>     puts "Nazev: " + $1
```

```
irb(main):041:2>   when /author=(.*)/
irb(main):042:2>     puts "Autor: " + $1
irb(main):043:2>   when /year=(.*)/
irb(main):044:2>     puts "Rok: " + $1
irb(main):045:2>   end
irb(main):046:1> }
Nazev: Ruby tutorial
Autor: Blackened
Rok: 2005
=> "title=Ruby tutorial\nauthor=Blackened\nyear=2005"
irb(main):047:0>
```

Až na části s `$1` by vám mělo být vše jasné. Pokud ne, doporučuji vrátit se zpět a znovu si pročíst kapitolu o řídících strukturách. V opačném případě si pojďme vysvětlit, co jsem to právě udělal. :-)

Pomocí metody `each_line` procházíme po jednotlivých řádcích řetězec `udaje`. Každý řádek je pomocí struktury `case` testován na výskyt jednotlivých vzorů. Závorka má speciální účinek, a to ten, že řetězec vyhovující vzoru uvnitř ní je uložen do speciální proměnné `$n`, kde n je číslo. Takto můžeme pochopitelně uložit i více proměnných.

9. METODY

9.1 Metody, alias funkce

Jak se bude kód vašeho programu rozvíjet, přijdete na to, že jsou věci, které se až příliš často opakují - například vytváříme-li matematickou aplikaci, věci jako výpočet faktoriálu či obvod kruhu využijeme poměrně často. Opisovat stále dokola to samé není zrovna efektivní a úprava takového kódu je přinejmenším pracná. Proto je více než vhodné vytvořit samostatnou funkci. V objektově orientovaných jazycích se funkcím říká metody.

Jednoduchá funkce pro výpočet odmocniny by vypadala takto:

```
irb(main):001:0> def odmocnina(x, n)
irb(main):002:1>   return x**(1/n.to_f)
irb(main):003:1> end
irb(main):004:0>
```

Definici metody uvozuje klíčové slovo `def`, za nímž následuje název metody. V závorce jsou udány parametry, které metoda přijímá. V těle metody je pak samotný kód. Za slovem `return` následuje návratová hodnota, samotná definice je pak ukončena slovem `end`. Naše funkce je teď připravena k použití:

```
irb(main):004:0> odmocnina(4, 2)
=> 2.0
irb(main):005:0>
```

Metody se volají svým jménem, v závorce se jsou jim předány hodnoty, jež jsou pak přiřazeny lokálním proměnným metody - v našem případě je x = 4, n = 2. Jelikož metoda vrací hodnotu (v tomto případě druhou odmocninu ze 4), lze ji postavit i na pravou stranu přiřazení:

```
irb(main):005:0> vysledek = odmocnina(25, 2)
=> 5.0
irb(main):006:0> vysledek
=> 5.0
irb(main):007:0>
```

Takto definované metodě musíme předat přesný počet parametrů (tedy dva). Co když ale druhou odmocninu používáme častěji než jakoukoli jinou a nechce se nám pořád psát 2? Ruby s tímto počítá a umožňuje zadat už v definici implicitní hodnotu. Naše upravená definice odmocniny by pak vypadala takto:

```
irb(main):007:0> def odmocnina(x, n=2)
irb(main):008:1>   return x**(1/n.to_f)
irb(main):009:1> end
=> nil
irb(main):010:0> odmocnina(4)
=> 2.0
irb(main):011:0> odmocnina(2187, 7)
=> 3.0
irb(main):012:0>
```

Parametry s implicitními hodnotami musí být vždy až co nejvíc vpravo a nelze je prokládat. Následující definice je tedy nesmyslná a chybná:

```
def pitomaMetoda(x=4, y, z=7)
  ...
```

9.2 Rekurze

Rekurze, neboli volání sebe sama, je silným nástrojem funkcionálních jazyků. V Ruby je rekurze samozřejmě možná:

```
irb(main):012:0> def fact(n)
irb(main):013:1>   if n == 0
irb(main):014:2>     return 1
irb(main):015:2>   else
irb(main):016:2*     return n * fact(n - 1)
irb(main):017:2>   end
irb(main):018:1> end
=> nil
irb(main):019:0>
```

Tato metoda vypočítá faktoriál (n!) daného čísla (pro připomenutí, 4! = 4 * 3 * 2 * 1 = 24) tak, že není-li n nulové, vynásobí n výsledkem po volání (n - 1)! a tak dále (pro lepší pochopení doporučuji si to rozepsat). To, že to skutečně funguje, můžeme snadno ověřit:

```
irb(main):019:0> fact(4)
=> 24
irb(main):020:0> fact(6)
=> 720
irb(main):021:0>
```

10. TŘÍDY

10.1 Objektově orientovaný přístup (OOP)

Než přejdeme k samotným třídám, bylo by víc než vhodné se na tomto místě zmínit o objektově orientovaném programování. Ačkoli není účelem tohoto tutoriálu naučit perfektně OOP, vzhledem k povaze jazyka se mu nelze vyhnout. Pokud již máte s psaním programů nějaké zkušenosti, je pravděpodobné, že jste se s tímto termínem už setkali a rozumíte, co znamená. Pokud ano, můžete následující text přeskočit a přejít rovnou ke kapitole *10.2 Definice třídy*.

Jak se s léty vyvíjela výpočetní technika, vyvíjely se i programovací jazyky, a to směrem k lepší srozumitelnosti. Konstruktéři strojů se oddělili od programátorů, děrné štítky a strojový kód nahradily srozumitelnější jazyky, jako Assembler, COBOL nebo C. Měnil se i přístup k psaní programů.

Dlouhou dobu byl zaběhnutý model procedurálního programování (pokud jste se třeba na škole setkali s Pascalem, víte, o čem mluvím), kde program sestává z dat, uložených v proměnných, a procedur, které s nimi pracují. Přístup dělit jednotlivé úkony do samostatných procedur je jistě efektivní a umožnil rozvoj složitých aplikací (například linuxové jádro je celé napsáno v C). S programy současného rozsahu se však ukazují i jeho nevýhody, a to především obtížnost jejich rozšíření či náchylnost k chybám.

Objektově orientovaný přístup se snaží ještě víc přiblížit programování lidskému myšlení a vnímání reálného světa. Když se rozhlédnete okolo sebe, nejspíš neuvažujete o věcech okolo jako o sadě proměnných a procedur, ale jako o objektech. OOP se snaží přiblížit psaní programů tomuto modelu:

```
class Pes
  def initialize(jmeno, rasa)
    @jmeno = jmeno
    @rasa = rasa
  end
  def stekni
    puts "Haf, haf!"
  end
  def to_s
    return "#{@jmeno} je #{@rasa}."
  end
end
```

Jednotlivými detaily se budeme podrobně zabývat v dalších kapitolách, ale už teď vidíme, že náš objekt Pes si udržuje údaje o svém jméně a rase a umí štěkat.

Tento přístup umožňuje nejen o problémech smýšlet a pracovat s nimi jako s reálnými věcmi, ale i skládat objekty z objektů nebo dědit vlastnosti tříd. Díky přístupovým metodám dokáže ochránit data před přepsáním a skrýt vnitřní implementace. Pokud nerozumíte, o čem to tady mluvím, pusťte to z hlavy. Vše si podrobně vysvětlíme v následujících kapitolách.

10.2 Definice třídy

Stejně jako je Aris pes a já jsem člověk, tak i náš objekt je instancí nějaké třídy. Třída se skládá z proměnných a metod. Definici třídy uvádíme slovem `class`, za něž stavíme jméno třídy (konvence je taková, že jména třídy se označují velkým písmenem). Následují definice proměnných a metod, definici uzavřeme slůvkem `end`:

```
irb(main):001:0> class Pes
irb(main):002:1>   def stekni
irb(main):003:2>     puts "Haf, haf!"
irb(main):004:2>   end
irb(main):005:1> end
=> nil
irb(main):006:0>
```

Takto jsme vytvořili třídu Pes, která má právě jednu metodu. Se samotnou třídou si mnoho legrace neužijeme, proto vytvoříme objekt (instanci) této třídy, tedy konkrétního psa:

```
irb(main):006:0> aris = Pes.new
=> #<Pes:0xb7f16724>
irb(main):007:0>
```

A cože náš Aris dovede?

```
irb(main):007:0> aris.stekni
Haf, haf!
=> nil
irb(main):008:0>
```

Že mu to ale moc hezky štěká. ;-)

10.3 Proměnné instance, inicializace

Uštěkaný objekt je možná velmi pěkná věc, ale zde taky jeho užitečnost končí. Chceme-li dělat něco pořádného, budeme muset ukládat i nějaké hodnoty.

K udržování hodnot v rámci instance slouží tzv. *proměnné instance* a jsou to lokální proměnné, společné všem jejím metodám. Od běžných proměnných se liší tím, že začínají znakem @ (zavináč). Trošku si našeho psa upravíme:

```
irb(main):009:0> class Pes
irb(main):010:1>   def initialize(jmeno, rasa)
irb(main):011:2>     @jmeno = jmeno
irb(main):012:2>     @rasa = rasa
irb(main):013:2>   end
irb(main):014:1>   def stekni
irb(main):015:2>     puts "Haf, haf!"
irb(main):016:2>   end
irb(main):017:1>   def to_s
irb(main):018:2>     return "#{@jmeno} je #{@rasa}."
irb(main):019:2>   end
irb(main):020:1> end
=> nil
irb(main):021:0>
```

Náš pes má nyní dvě proměnné instance: `@jmeno` a `@rasa`. Krom toho přibyly dvě metody: `initialize` a `to_s`. Jelikož jsou obě důležité a ukazují nové věci, pojďme se na ně podívat detailněji.

Metoda `initialize` je standardní název takzvané inicializační metody a slouží k předání hodnot objektu už při jeho vytvoření. Na rozdíl od jiných metod se nevolá přímo, ale při vytváření metodou `new`:

```
irb(main):021:0> mujPes = Pes.new("Aris", "Border Terier")
=> #<Pes:0xb7f21200 @rasa="Border Terier", @jmeno="Aris">
irb(main):022:0>
```

Z návratové hodnoty se dá vyčíst, že námi zadané parametry byly skutečně přiřazeny konkrétním proměnným instance.

Metoda `to_s` by vám měla být povědomá - ano, používali jsme ji k převodu jiných datových typů na typ String. Jelikož je Ruby čistě objektový jazyk, i proměnné jsou vlastně instancemi nějaké třídy. Naše třídy nejsou nic jiného než složitější datové struktury a jako takové je vhodné (máme-li zapotřebí vypsat hodnotu vnitřních proměnných) definovat standardní metodu `to_s`. Ta přímo nevypisuje nic na obrazovku, jen vrací řetězec (příkaz `return`):

```
irb(main):022:0> mujPes.to_s
```

```
=> "Aris je Border Terier."
irb(main):023:0>
```

10.4 Přístupové metody

Možná vás napadlo, že by se dal pomocí tečkové notace vypsat obsah proměnné instance:

```
irb(main):023:0> mujPes.jmeno
NoMethodError: undefined method `jmeno' for #<Pes:0xb7f21200 @rasa="Border
Terier", @jmeno="Aris">
        from (irb):58
        from :0
irb(main):024:0>
```

Nebo snad dokonce:

```
irb(main):024:0> mujPes.@jmeno
SyntaxError: compile error
(irb):59: syntax error
        from (irb):59
        from :0
irb(main):025:0>
```

Oba pokusy končí chybovou hláškou. Proboha, ale k čemu jsou mi proměnné, které nemohu měnit ba dokonce je ani zobrazit? I toto je součást objektového přístupu. K práci s proměnnými slouží konkrétní metody. To nejen omezuje výskyt chyby, ale zároveň skrývá vnitřní postupy před světem. Používáte-li nějakou třídu, už vás nezajímá, jak to uvnitř funguje. Co potřebujete vědět je, jak se pracuje s jednotlivými metodami. Tento přístup také umožňuje měnit vnitřní implementace bez toho, aby se to jakkoli dotklo kódu, který s třídou pracuje.

Náš pes, obohacený o možnost čtení hodnot proměnných instance by vypadal takto (pro zjednodušení tentokrát vynecháme metody `stekni` a `to_s`):

```
irb(main):025:0> class Pes
irb(main):026:1>   def initialize(jmeno, rasa)
irb(main):027:2>     @jmeno = jmeno
irb(main):028:2>     @rasa = rasa
irb(main):029:2>   end
irb(main):030:1>   def jmeno
irb(main):031:2>     return @jmeno
irb(main):032:2>   end
irb(main):033:1>   def rasa
irb(main):034:2>     return @rasa
irb(main):035:2>   end
irb(main):036:1> end
=> nil
irb(main):037:0>
```

Přístupové metody jsem pojmenoval stejně, jako proměnné instance. Toto umožňuje intuitivní přístup k proměnným, avšak někteří možná upřednostňují pojmenování typu `vypisJmeno` a `nastavJmeno`. Výběr je na vás. Každopádně nyní už obsah proměnných vidíme:

```
irb(main):037:0> mujPes = Pes.new("Aida", "Kokrspanel")
=> #<Pes:0xb7ef9764 @rasa="Kokrspanel", @jmeno="Aida">
irb(main):038:0> mujPes.jmeno
=> "Aida"
irb(main):039:0> mujPes.rasa
=> "Kokrspanel"
irb(main):040:0>
```

Zapisovat však stále nemůžeme. Dejme tomu, že náš pes je ještě štěně a na konkrétním jméně jsme se zatím neshodli. Rasa je od narození stejná a tedy možnost ji měnit by vedla k zbytečným zmatkům:

```
irb(main):040:0> class Pes
irb(main):041:1>   def initialize(jmeno, rasa)
irb(main):042:2>     @jmeno = jmeno
irb(main):043:2>     @rasa = rasa
irb(main):044:2>   end
irb(main):045:1>   def jmeno
irb(main):046:2>     return @jmeno
irb(main):047:2>   end
irb(main):048:1>   def rasa
irb(main):049:2>     return @rasa
irb(main):050:2>   end
irb(main):051:1>   def jmeno=(noveJmeno)
irb(main):052:2>     @jmeno = noveJmeno
irb(main):053:2>   end
irb(main):054:1> end
=> nil
irb(main):055:0>
```

Zápis metody `jmeno=(noveJmeno)` nám umožní pracovat s ní, jako by se jednalo přímo o proměnnou:

```
irb(main):055:0> mujPes = Pes.new("Aida", "Kokrspanel")
=> #<Pes:0xb7f1480c @rasa="Kokrspanel", @jmeno="Aida">
irb(main):056:0> mujPes.jmeno  # puvodni jmeno
=> "Aida"
irb(main):057:0> mujPes.jmeno = "Betty"
=> "Betty"
irb(main):058:0> mujPes.jmeno  # nove jmeno
=> "Betty"
irb(main):059:0>
```

Ochrana našich proměnných je dokonalá. Můžeme rozhodovat, které umožníme číst a do kterých zapisovat a které skryjeme před zraky uživatele naší třídy úplně. To má své nesporné výhody, nicméně kvůli relativně triviální operaci se definice naší třídy natáhla na pěkných pár řádků. Naštěstí programátoři jsou od přírody líní lidé a tak Ruby nabízí elegantní zkratku.

```
irb(main):059:0> class Pes
irb(main):060:1>   def initialize(jmeno, rasa)
irb(main):061:2>     @jmeno = jmeno
irb(main):062:2>     @rasa = rasa
irb(main):063:2>   end
irb(main):064:1>   attr_reader :jmeno, :rasa
irb(main):065:1>   attr_writer :jmeno
irb(main):066:1> end
=> nil
irb(main):067:0>
```

Tento příklad je naprostým ekvivalentem příkladu předchozího. Příkaz `attr_reader` vytvoří metody pro čtení a `attr_writer` pro zápis k proměnným, které mu předáme jako parametry. Všimněte si, že názvy začínají dvojtečkou, nikoli zavináčem, a jsou odděleny čárkami. Že to skutečně funguje se přesvědčíme snadno:

```
irb(main):067:0> mujPes = Pes.new("Aris", "jezevcik")
=> #<Pes:0xb7f54a50 @rasa="jezevcik", @jmeno="Aris">
irb(main):068:0> mujPes.rasa
=> "jezevcik"
irb(main):069:0> mujPes.jmeno = "Arin"
=> "Arin"
irb(main):070:0> mujPes.jmeno
=> "Arin"
irb(main):071:0>
```

10.5 Dědičnost

Dědičnost je způsob, jak odvodit novou třídu z třídy již existující. Nová třída "podědí" metody a proměnné původní třídy a umožní jejich rozšíření. Mějme tedy třídu Pes (pokud jste od minulé kapitoly nevypnuli `irb` nebo s naším psem samostatně neexperimentovali, měli byste ji už mít definovanou):

```
irb(main):059:0> class Pes
irb(main):060:1>   def initialize(jmeno, rasa)
irb(main):061:2>     @jmeno = jmeno
irb(main):062:2>     @rasa = rasa
irb(main):063:2>   end
irb(main):064:1>   attr_reader :jmeno, :rasa
irb(main):065:1>   attr_writer :jmeno
irb(main):066:1> end
=> nil
irb(main):067:0>
```

Nyní vytvoříme třídu `Vorisek`, která bude od třídy Pes odvozena. Abychom ale neměli různě pojmenované stejné třídy, naučíme našeho voříška štěkat:

```
irb(main):067:0> class Vorisek<Pes
irb(main):068:1>   def stekej
irb(main):069:2>     puts "Haf, haf!"
irb(main):070:2>   end
irb(main):071:1> end
=> nil
irb(main):072:0>
```

Zápis `Vorisek<Pes` reprezentuje dědičnost, tedy že `Vorisek` přejímá vlastnosti třídy `Pes`. Následuje definice nové metody. Že náš voříšek skutečně dovede to, co jakýkoli pes, si můžeme snadno ověřit:

```
irb(main):072:0> mujPes = Vorisek.new("Asta", "buhvi co")
=> #<Vorisek:0xb7f232cc @rasa="buhvi co", @jmeno="Asta">
irb(main):073:0> mujPes.jmeno
=> "Asta"
irb(main):074:0> mujPes.rasa
=> "buhvi co"
irb(main):075:0> mujPes.stekej
Haf, haf!
=> nil
irb(main):076:0>
```

10.6 Redefinice metod

Možná jste si všimli, že náš voříšek podědil i v jeho případě nesmyslný údaj "rasa". Abychom se tohoto údaje zbavili, chtělo by to předefinovat metodu `initialize`. To provedeme tak, že ji jednoduše nadefinujeme znovu a původní metoda bude nahrazena novou:

```
irb(main):076:0> class Vorisek<Pes
irb(main):077:1>   def initialize(jmeno)
irb(main):078:2>     @jmeno = jmeno
irb(main):079:2>   end
irb(main):080:1>   def stekej
irb(main):081:2>     puts "Haf, haf!"
irb(main):082:2>   end
irb(main):083:1> end
=> nil
irb(main):084:0>
```

V případě, že bychom chtěli našeho psa naopak rozšířit o další údaje, bychom mohli postupovat stejně. Opisovat obsah celé metody `initialize` je však při větším rozsahu nepohodlné a vyžaduje znalost její

implementace v původní třídě. Naštěstí lze zdědit i obsah přepisované metody:

```
irb(main):084:0> class Pes2<Pes
irb(main):085:1>   def initialize(jmeno, rasa, popis, vek)
irb(main):086:2>     super(jmeno, rasa)
irb(main):087:2>     @popis = popis
irb(main):088:2>     @vek = vek
irb(main):089:2>   end
irb(main):090:1>   attr_reader :popis, :vek
irb(main):091:1>   attr_writer :popis, :vek
irb(main):092:1> end
=> nil
irb(main):093:0>  mujPes  =  Pes2.new("Aida",  "kokrspanel",  "Roztomila  mala
potvurka.", 14)
=>  #<Pes2:0xb7f32844  @rasa="kokrspanel",  @popis="Roztomila  mala  potvurka.",
@jmeno="Aida", @vek=14>
irb(main):094:0> mujPes.popis
=> "Roztomila mala potvurka."
irb(main):095:0>
```

10.7 Řízení přístupu

Už víme, že proměnné instance jsou považovány za soukromé a chceme-li k nim přistoupit, musíme definovat patřičné přístupové metody. To nám umožní nejen měnit způsob, jak je s proměnnými vnitřně nakládáno, bez toho, aby se jakkoli měnila práce s metodou, ale i volit, ke kterým proměnným vůbec přístup umožníme.

Složitější třídy obsahují spoustu metod, které slouží k práci s daty a jsou volány jinými metodami, ale zároveň se neočekává, že se budou volat "zvenčí". Ruby rozlišuje tři druhy metod třídy:

- **veřejné metody** (`public`) - Ty, které lze volat odkudkoli z programu. Neřekneme-li jinak, jsou všechny metody třídy veřejné.
- **chráněné metody** (`protected`) - Mohou s nimi pracovat jen objekty z dané třídy. To znamená, že máme-li dva objekty třídy Pes, mohou ke svým metodám navzájem přistupovat. Pro objekty jiných tříd ani zvenčí nejsou dostupné.
- **soukromé metody** (`private`) - Mohou s nimi pracovat jen metody dané třídy v rámci jedné instance. Nejsou dostupné ani pro jiné objekty stejné třídy.

Které metody jsou veřejné, chráněné nebo soukromé můžeme udat explicitně pomocí klíčových slov `public`, `protected` nebo `private`:

```
irb(main):095:0> class Kruh
irb(main):096:1>     def initialize(polomer)  # implicitne verejna
irb(main):097:2>       @polomer = polomer
irb(main):098:2>       @PI = 3.14159
irb(main):099:2>     end
irb(main):100:1>
irb(main):101:1*   private          # nasledujici metody budou soukrome
irb(main):102:1>
irb(main):103:1*     def obvod
irb(main):104:2>       return 2 * @PI * @polomer
irb(main):105:2>     end
irb(main):106:1>     def obsah
irb(main):107:2>       return @PI * @polomer**2
irb(main):108:2>     end
irb(main):109:1>
irb(main):110:1*   public           # nasledujici metody budou verejne
irb(main):111:1>
irb(main):112:1*     def parametry
irb(main):113:2>         return "polomer:\t" + @polomer.to_s + "\nobvod:\t\t" +
obvod.to_s + "\nobsah:\t\t" + obsah.to_s + "\n"
```

```
irb(main):114:2>     end
irb(main):115:1> end
=> nil
irb(main):116:0>
```

Třída `Kruh` obsahuje krom inicializace jen jedinou veřejnou metodu parametry, která vrátí řetězec s informacemi o kruhu:

```
irb(main):116:0> k = Kruh.new(5)
=> #<Kruh:0xb7eb0aa0 @PI=3.14159, @polomer=5>
irb(main):117:0> puts k.parametry
polomer:        5
obvod:          31.4159
obsah:          78.53975
=> nil
irb(main):118:0>
```

Druhou možností je definovat jednotlivé metody a pak teprve rozlišit, které jsou veřejné, chráněné a soukromé:

```
irb(main):118:0> class Kruh
irb(main):119:1>   def initialize(polomer)
irb(main):120:2>     @polomer = polomer
irb(main):121:2>     @PI = 3.14159
irb(main):122:2>   end
irb(main):123:1>   def obvod
irb(main):124:2>     return 2 * @PI * @polomer
irb(main):125:2>   end
irb(main):126:1>   def obsah
irb(main):127:2>     return @PI * @polomer**2
irb(main):128:2>   end
irb(main):129:1>   def parametry
irb(main):130:2>       return "polomer:\t" + @polomer.to_s + "\nobvod:\t\t" +
obvod.to_s + "\nobsah:\t\t" + obsah.to_s + "\n"
irb(main):131:2>   end
irb(main):132:1>   public :initialize, :parametry
irb(main):133:1>   private :obvod, :obsah
irb(main):134:1> end
=> Kruh
irb(main):135:0>
```

Toto je o něco kratší a možná i přehlednější.

```
irb(main):135:0> k = Kruh.new(5)
=> #<Kruh:0xb7eb8084 @PI=3.14159, @polomer=5>
irb(main):136:0> puts k.parametry
polomer:        5
obvod:          31.4159
obsah:          78.53975
=> nil
irb(main):137:0>
```

Při pokusu zavolat soukromou metodu dostaneme chybové hlášení:

```
irb(main):137:0> k.obvod
NoMethodError: private method `obvod' called for #<Kruh:0xb7eb8084 @PI=3.14159,
@polomer=5>
        from (irb):138
        from :0
irb(main):138:0>
```

10.8 Proměnné třídy

Na závěr naší kapitoly o třídách ještě malý bonbónek. Ruby totiž kromě proměnných instance umožňuje i definici proměnných třídy:

```
irb(main):138:0> class Pes
irb(main):139:1>   @@pocetPsu = 0  # promenna tridy Pes
irb(main):140:1>
irb(main):141:1*   def initialize(jmeno, rasa)
irb(main):142:2>     @jmeno = jmeno  # promenna instance
irb(main):143:2>     @rasa = rasa    # promenna instance
irb(main):144:2>     @@pocetPsu += 1 # pripocitame noveho psa
irb(main):145:2>   end
irb(main):146:1>   def to_s
irb(main):147:2>      return "jmeno: " + @jmeno + "\nrasa:  " + @rasa + "\npsu:
" + @@pocetPsu.to_s + "\n"
irb(main):148:2>   end
irb(main):149:1> end
=> nil
irb(main):150:0> pes1 = Pes.new("Aris", "border terier")
=> #<Pes:0xb7ee2adc @jmeno="Aris", @rasa="border terier">
irb(main):151:0> puts pes1.to_s
jmeno: Aris
rasa:  border terier
psu:   1
=> nil
irb(main):152:0> pes2 = Pes.new("Arin", "jezevcik")
=> #<Pes:0xb7ec9d5c @jmeno="Arin", @rasa="jezevcik">
irb(main):153:0> puts pes2.to_s
jmeno: Arin
rasa:  jezevcik
psu:   2
=> nil
irb(main):154:0> puts pes1.to_s
jmeno: Aris
rasa:  border terier
psu:   2
=> nil
irb(main):155:0>
```

11. PRÁCE SE SOUBORY

11.1 Nutné základy

Píšeme-li program nebo skript, často pracujeme s externími textovými soubory. Práce s nimi je v Ruby velmi jednoduchá:

```
File.open(soubor, mod)
```

Tento příkaz vrací objekt typu File. Jako parametry mu zadáme název souboru a mód:

```
f = File.open("pokus.txt", "r")
```

Mód udává, co s naším souborem chceme dělat (číst, zapisovat, nebo obojí):

r …Otevře soubor pro čtení, ukazatel je na začátku souboru.
r+ ...Otevře soubor pro čtení i zápis. Ukazatel je na začátku souboru.
w ...Otevře soubor pro zápis. Pokud soubor existuje, je jeho obsah vymazán, pokud neexistuje, je vytvořen prázdný soubor. Ukazatel je na začátku souboru.
w+ ...Otevře soubor pro čtení i zápis. Pokud soubor existuje, je jeho obsah vymazán, pokud neexistuje, je vytvořen prázdný soubor. Ukazatel je na začátku souboru.
a ...Otevře soubor pro zápis na konec souboru; pokud soubor neexistuje, je vytvořen. Ukazatel je na konci souboru.
a+ ...Otevře soubor pro čtení a zápis na konec souboru; pokud soubor neexistuje, je vytvořen. Ukazatel pro čtení je na začátku souboru, ukazatel pro zápis je vždy na jeho konci.

Jakmile práci se souborem ukončíme, je vhodné jej uzavřít a uvolnit tak místo v paměti počítače:

```
f.close
```

V dalších podkapitolách si postupně probereme čtení, zápis i přidávání na konec souboru.

11.2 Čtení ze souboru

11.2.1 Metoda `gets`

Vytvořte si textový soubor `pokus.txt` s následujícím obsahem:

```
Toto je 1. radek...
A toto druhy!
...i treti by se nasel. ;-)
```

Přesuňte se do adresáře s ním a spusťte `irb`, tak nebudete muset psát absolutní cesty. Pro čtení jednotlivých řádků ze souboru pak můžeme použít nám už známou metodu `gets`:

```
irb(main):001:0> f = File.open("pokus.txt", "r")
=> #<File:pokus.txt>
irb(main):002:0> f.gets
=> "Toto je 1. radek...\n"
irb(main):003:0> f.gets
=> "A toto druhy!\n"
irb(main):004:0> f.gets
=> "...i treti by se nasel. ;-)\n"
irb(main):005:0> f.gets
=> nil
irb(main):006:0> f.close
=> nil
irb(main):007:0>
```

V případě, že se dostaneme na konec souboru, vrací `gets` prázdnou hodnotu nil. Jak už jsme si říkali, je `nil` považováno za `false`. Toho se dá využít; program na výpis obsahu programu by pak mohl vypadat třeba takto:

```
irb(main):007:0> f = File.open("pokus.txt", "r")
=> #<File:pokus.txt>
irb(main):008:0> while radek = f.gets
irb(main):009:1>   print radek
irb(main):010:1> end
Toto je 1. radek...
A toto druhy!
...i treti by se nasel. ;-)
=> nil
irb(main):011:0> f.close
=> nil
irb(main):012:0>
```

11.2.2 Metoda `getc`

V případě, že nechcete číst po celých řádcích, ale po jednotlivých znacích, můžete použít metodu `getc`. Ta vrací celé číslo, reprezentující ASCII hodnotu přečteného znaku. Pokud chceme zobrazovat znaky, musíme celočíselnou hodnotu převést na znak pomocí metody `chr`:

```
irb(main):012:0> f = File.open("pokus.txt", "r")
=> #<File:pokus.txt>
irb(main):013:0> while znak = f.getc
irb(main):014:1>   print znak.chr, " "
irb(main):015:1> end
T o t o   j e   1 .   r a d e k . . .
 A   t o t o   d r u h y !
 . . . i   t r e t i   b y   s e   n a s e l .   ; - )
 => nil
irb(main):016:0> f.close
=> nil
irb(main):017:0>
```

Upozorňuji, že mezery mezi znaky jsem zavedl záměrně pro zřetelnost. Stejně jako `gets` i `getc` vrací `nil` v případě, že se dostal na konec souboru.

11.2.3 Metoda `read`

Metoda `read` se neomezuje ani na čtení jednoho řádku nebo znaku, ale umožňuje nám uvést, kolik znaků chceme načíst:

```
irb(main):017:0> f = File.open("pokus.txt", "r")
=> #<File:pokus.txt>
irb(main):018:0> f.read(6)
=> "Toto j"
irb(main):019:0> f.read(100)
=> "e 1. radek...\nA toto druhy!\n...i treti by se nasel. ;-)\n"
irb(main):020:0> f.close
=> nil
irb(main):021:0>
```

11.2.4 Metoda `readlines`

Metoda `readlines` vrací pole řádků:

```
irb(main):021:0> f = File.open("pokus.txt", "r")
=> #<File:pokus.txt>
```

```
irb(main):022:0> pole = f.readlines
=> ["Toto je 1. radek...\n", "A toto druhy!\n", "...i treti by se nasel. ;-)\n"]
irb(main):023:0> f.close
=> nil
irb(main):024:0>
```

Alternativně lze také zadat řetězec, který má sloužit jako oddělovač namísto '\n':

```
irb(main):024:0> f = File.open("pokus.txt", "r")
=> #<File:pokus.txt>
irb(main):025:0> pole = f.readlines(" ")
=> ["Toto ", "je ", "1. ", "radek...\nA ", "toto ", "druhy!\n...i ", "treti ",
"by ", "se ", "nasel. ", ";-)\n"]
irb(main):026:0> f.close
=> nil
irb(main):027:0>
```

11.2.5 Metoda `each` *(*`each_line`*)*

Procházet soubor pomocí cyklu `while` sice jde, Ruby však má pro tyto účely zabudované metody. S metodou `each` jsme se setkali už v kapitole *7.10* a proto by pro vás následující text neměl být novinkou.

Metoda `each` (nebo její synonymum `each_line`) prochází soubor po jednotlivých řádcích:

```
irb(main):027:0> f.each { |radek| print radek }
Toto je 1. radek...
A toto druhy!
...i treti by se nasel. ;-)
=> #<File:pokus.txt>
irb(main):028:0> f.close
=> nil
irb(main):029:0>
```

Tento způsob je o něco elegantnější než jeho ekvivalent s cyklem `while` a je také mnohem více ve stylu Ruby. :-) Podobně jako u `readlines` i zde lze definovat oddělovač.

11.2.6 Metoda `each_byte`

Opět ekvivalentní procházení souboru po znacích. Probírána byla už v kapitole *7.10.3*, omezím se tedy jen na příklad:

```
irb(main):029:0> f = File.open("pokus.txt", "r")
=> #<File:pokus.txt>
irb(main):030:0> f.each_byte { |znak| print znak.chr, " " }
T o t o   j e   1 .   r a d e k . . .
 A   t o t o   d r u h y !
 . . . i   t r e t i   b y   s e   n a s e l .   ; - )
 => #<File:pokus.txt>
irb(main):031:0> f.close
=> nil
irb(main):032:0>
```

11.2.7 Metoda `lineno`

Pomocí `lineno` lze zjistit informaci o čísle řádku, který čteme:

```
irb(main):032:0> f = File.open("pokus.txt", "r")
=> #<File:pokus.txt>
irb(main):033:0> f.each { |retezec| print "#{f.lineno} -- #{retezec}" }
1 -- Toto je 1. radek...
2 -- A toto druhy!
```

```
3 -- ...i treti by se nasel. ;-)
=> #<File:pokus.txt>
irb(main):034:0> f.close
=> nil
irb(main):035:0>
```

11.3 Zápis do souboru

Otevřeme-li soubor s parametrem `"w"`, ať už je v něm cokoli, je jeho obsah nemilosrdně vymazán a soubor připraven k naplnění našimi údaji. Pokud soubor našeho jména neexistuje, je vytvořen. K samotnému zápisu do souboru pak můžeme využívat všech našich tří známých metod - `puts`, `print` a `printf`:

```
irb(main):035:0> f = File.open("zapis.txt", "w")
=> #<File:zapis.txt>
irb(main):036:0> f.puts "Zapsano prikazem puts"
=> nil
irb(main):037:0> f.print "Zapsano prikazem print\n"
=> nil
irb(main):038:0> f.printf "PI = %4.2f\n", 3.141592
=> nil
irb(main):039:0> f.close
=> nil
irb(main):040:0>
```

Ve svém adresáři byste měli nyní najít soubor `zapis.txt` s následujícím obsahem:

```
Zapsano prikazem puts
Zapsano prikazem print
PI = 3.14
```

11.4 Zápis na konec souboru

Píšeme-li třeba logovací aplikaci, to, že se při zápisu soubor smaže se nám vůbec nehodí. Zapisovat na konec souboru nám umožňuje otevření pro zápis s parametrem `"a"`:

```
irb(main):040:0> f = File.open("zapis.txt", "a")
=> #<File:zapis.txt>
irb(main):041:0> f.puts "Sbohem a diky za vsechny ryby!"
=> nil
irb(main):042:0> f.close
=> nil
irb(main):043:0>
```

Když otevřete soubor `zapis.txt` teď, měli byste v něm najít:

```
Zapsano prikazem puts
Zapsano prikazem print
PI = 3.14
Sbohem a diky za vsechny ryby!
```

12. OŠETŘENÍ VÝJIMEK

12.1 Výjimka?

Zvláště při práci s uživatelskými daty nastávají situace, kdy hrozí při nesprávné kombinaci hodnot zhroucení programu v důsledku vzniklé chyby. Představte si program, který přijímá od uživatele dvě číselné hodnoty a vypisuje jejich podíl. Dojde-li však v programu na dělení nulou, vypíše se chybová hláška a program je předčasně ukončen:

```
blackened@debian:~$ ruby
puts "Deleni nulou: "
puts 10 / 0
puts "Kvuli chybe se uz tato cast programu neprovede. :'("
^D
Deleni nulou:
-:2:in `/': divided by 0 (ZeroDivisionError)
        from -:2
blackened@debian:~$
```

Příkazy za místem vzniku chyby nebudou provedeny. Podobný problém může nastat i při pokusu otevřít pro čtení neexistující soubor:

```
blackened@debian:~/programy/ruby$ ls
pokus.txt  zapis.txt

blackened@debian:~/programy/ruby$ ruby
f = File.open("neexistuji.txt", "r")
puts "Tento text se v dusledku chyby nevypise"
f.close  # nedojde ani k uzavreni souboru
^D
-:1:in `initialize': No such file or directory - neexistuji.txt (Errno::ENOENT)
        from -:1:in `open'
        from -:1

blackened@debian:~/programy/ruby$
```

Toto chování je nejen nepříjemné pro uživatele, ale dělá i špatnou vizitku programátorovi. Ruby proto nabízí nástroje k ošetření výjimek a každý slušný program by je měl využívat.

12.2 `begin...rescue...ensure...end`

Pokud víme, že v nějaké části kódu hrozí výjimka, můžeme tento "nebezpečný" blok označit a v případě výskytu výjimky na něj zareagovat. K tomu v Ruby slouží konstrukce `begin...rescue...end`. Vyskytne-li se výjimka v bloku `begin`, provede se blok `rescue`. Tak můžeme sami rozhodnout, jak reagovat - zda vypsat varovnou hlášku, ukončit program nebo událost zapsat do logu atd.:

```
irb(main):001:0> delenec = 10
=> 10
irb(main):002:0> delitel = 0
=> 0
irb(main):003:0> begin
irb(main):004:1*   puts delenec / delitel  # nebezpecna cast
irb(main):005:1> rescue
irb(main):006:1>   puts "Nulou nelze delit!"
irb(main):007:1> end
Nulou nelze delit!
=> nil
irb(main):008:0>
```

Krom `begin` a `rescue` nabízí Ruby ještě blok `ensure` - ten je proveden vždy, ať už výjimka nastala či ne:

```
irb(main):008:0> begin
irb(main):009:1*   # zde se provadi nebezpecny kod
irb(main):010:1*   f = File.open("ruby.log", "w")
irb(main):011:1>   f.print = "Podil: ", 10/0, "\n"
irb(main):012:1> rescue
irb(main):013:1>   # tento blok se provede v pripade vyskytu vyjimky
irb(main):014:1*   puts "CHYBA!"
irb(main):015:1> ensure
irb(main):016:1*   # tento blok se provede vzdy
irb(main):017:1*   f.close
irb(main):018:1> end
CHYBA!
=> nil
irb(main):019:0>
```

12.3 Zvláštní proměnná `$!`

Pokud si nejste jisti, jaká že to přesně nastala výjimka, a nebo chcete nechat formulaci na interpretru, pak se vám bude hodit proměnná $! - v ní je totiž uložena poslední výjimka, která nastala:

```
irb(main):019:0> begin
irb(main):020:1*   puts 10/0
irb(main):021:1> rescue
irb(main):022:1>   puts "CHYBA: " + $!
irb(main):023:1> end
CHYBA: divided by 0
=> nil
irb(main):024:0>
```

12.4 Vyvolání výjimek: `raise`

Výjimku je možné i uměle vytvořit - k tomu slouží příkaz `raise`:

```
irb(main):024:0> begin
irb(main):025:1*   raise "Chyba! Vase pozornost klesla pod unosnou mez!!!"
irb(main):026:1> rescue
irb(main):027:1>   puts $!
irb(main):028:1> end
Chyba! Vase pozornost klesla pod unosnou mez!!!
=> nil
irb(main):029:0>
```

13. KONTAKT S OPERAČNÍM SYSTÉMEM

13.1 Spuštění s parametry

Většina programů a skriptů pro systémy unixového typu (Linux, *BSD, AIX, Solaris, ...) se drží zásady, že je lze spouštět bez nutnosti další interakce s uživatelem a to předáním potřebných hodnot už při jejich spuštění. Díky tomu je možné práci automatizovat pomocí skriptů nebo psát různé frontendy a GUI nástavby pro konzolové aplikace. Tento přístup je dobré zachovat a proto si nyní ukážeme, jak parametry přijmout.

13.1.1 Zvláštní proměnná `$*`

Všechny parametry zadané při spouštění našeho skriptu jsou uloženy ve speciální proměnné `$*`. Ta je typu Array a proto se s ní také dobře pracuje. Vytvořte soubor `parametry.rb` s následujícím obsahem:

```
parametry = $*
parametry.each { |p| print p, " : " }
puts
```

Tento program vypíše všechny jemu zadané parametry, oddělené dvojtečkou:

```
blackened@debian:~/programy/ruby$ ruby parametry.rb prvni druhy treti 'Ahoj, svete!'
prvni : druhy : treti : Ahoj, svete! :
blackened@debian:~/programy/ruby$
```

Práce s proměnnou `$*` je ekvivalentní práci s běžnou proměnnou typu Array.

13.1.2 Zvláštní proměnná `$0`

Proměnná `$0` je typu String a uchovává název našeho skriptu. Mějme skript `ahoj.rb` s následujícím obsahem:

```
puts "Ahoj, zdravi te tvuj " + $0
```

Když ho spustíme, dostaneme:

```
blackened@debian:~/programy/ruby$ ruby ahoj.rb
Ahoj, zdravi te tvuj ahoj.rb
blackened@debian:~/programy/ruby$
```

13.2 Příkazy systému

Upozornění: Níže uvedené příkazy (`pwd`, `date`, `whoami` atp.) jsou platné pro systémy unixového typu a pod systémy společnosti Microsoft nebudou pravděpodobně fungovat. Pro pochopení problematiky by nicméně mělo stačit následující text jen přečíst.

Může se stát, že budete potřebovat zadávat příkazy přímo operačnímu systému. V jazyku C k tomu slouží příkaz `system();`, v Ruby je za příkaz systému považováno cokoli uzavřené mezi obrácené apostrofy (`):

```
irb(main):001:0> `pwd`  # vrati cestu k pracovnimu adresari
=> "/home/blackened/programy/ruby\n"
irb(main):002:0> `date` # vrati datum a cas
=> "\303\232t srp  2 17:32:33 CEST 2005\n"
irb(main):003:0>
```

Díky tomu, že odezva systému je předána jako návratová hodnota, je možné ji přiřadit do proměnné:

```
irb(main):003:0> jmeno = `whoami`  # vrati vas login
```

```
=> "blackened\n"
irb(main):004:0>
```

Unixový příkaz `whoami` vrátí login uživatele. V našem programu je tento předán jako řetězec proměnné `jmeno`. Následující program zjistí login uživatele a pozdraví ho:

```
irb(main):004:0> jmeno = `whoami`.chop # zbavime se "\n"
=> "blackened"
irb(main):005:0> puts "Dobry den, #{jmeno}!"
Dobry den, blackened!
=> nil
irb(main):006:0>
```

Přejímat údaje ze systému je moc užitečná věc. V případě, že bychom chtěli hodnotu z naší proměnné i předat, můžeme použít zápisu `#{}`:

```
irb(main):006:0> prikaz = "date"
=> "date"
irb(main):007:0> datum = `#{prikaz}`
=> "\303\232t srp  2 17:59:44 CEST 2005\n"
irb(main):008:0>
```

Toto se může hodit třeba v případě dávkového zpracování souborů, např. v kombinaci s grafickým balíkem ImageMagick.

K práci se systémem lze použít spoustu metod třídy File (`chmod`, `chown`, `umask`) - ty jsou však příliš závislé na operačním systému unixového typu a popis jejich použití najdete v referencích jazyka.

14. ZÁVĚR

14.1 Co se nevešlo...

Tento tutoriál si od začátku kladl dvě podmínky - že bude stručný a srozumitelný, jak má správný tutoriál (průvodce) být. To první se mi jaksi vymklo z rukou a očekávaný rozsah byl nakonec dvojnásobně přesažen. Srozumitelnost musíte posoudit až vy, snažil jsem se však podat text formou spousty příkladů s jejich vysvětlením.

Díky této koncepci se sem spousta věcí nevešla, nicméně pokud jste se dostali až sem, měli byste být již schopni s Ruby plnohodnotně pracovat. Chcete-li rozšiřovat své znalosti, doporučuji projít dokumentaci na *www.ruby-doc.org*. V několika následujících bodech se pokusím shrnout pár zajímavých věcí, které by vás v souvislosti s Ruby mohly zajímat:

- Ruby/TK Tutorial (EN) [*http://members.chello.nl/k.vangelder/ruby/learntk/index.html*]
- Ruby/GTK2 Tutorial (EN) [*http://ruby-gnome2.sourceforge.jp/hiki.cgi?tut-gtk*]
- Ruby on Rails (EN) [*http://www.rubyonrails.com/*]

14.2 Kontakt

Najdete-li v tomto textu místa, která jsou nejasná (nedostatečně vysvětlená), faktické chyby či závažné nedostatky, neváhejte a kontaktujte mě na níže uvedené e-mailové adrese. V předmětu zprávy prosím uveďte slovo "Ruby". Stejně tak jsou vítány vaše ohlasy.

E-mail: *lord.blackened@seznam.cz*
web: *http://blackened.wz.cz*

Děkuji vám za pozornost a přeji hezký den (noc?). :-)