Guia do Utilizador de Ruby

Traduzido para o Português por **José A. Soares Augusto**, baseado nas seguintes versões:

Original Japonês de **Yukihiro Matsumoto.** Tradução para Inglês de **GOTO Kentaro** e **Julian Fondren**. Versão refinada em Inglês de **Mark Slagell.**

Versão Portuguesa 1.2, de Outubro de 2006

Introdução

O Ruby é uma **linguagem de programação orientada para objectos (OO)** de aprendizagem simples, originária do **Japão**, cujo criador, arquitecto e programador principal é **Yukihiro Matsumoto** (carinhosamente apelidado de **Matz** pela comunidade do Ruby). O Ruby pode causar estranheza ao primeiro contacto, mas a sua sintaxe foi desenhada com o objectivo de promover a fácil escrita e leitura de programas. Ao ler este "Guia do Utilizador de Ruby" irá, decerto, ganhar alguma experiência na utilização **trivial** desta linguagem de programação. Por vezes, nalgumas secções do documento, ir-se-á discutir a natureza do Ruby de uma forma mais aprofundada do que aquela patente no manual de referência, focando-se alguns dos detalhes da sua implementação. Se pretender apenas um conhecimento superficial da linguagem, poderá ignorar estas passagens.

A licença de distribuição do Ruby é "open source", ou seja, em termos práticos é de utilização livre e gratuita. O *site* oficial é www.ruby-lang.org. Nele encontrará muita informação: o código fonte da linguagem, distribuições binárias para as plataformas (ou sistemas operativos) mais comuns, tais como o Windows (95, 98, 98SE, ME, NT, 2000, XP) o Linux e outras variedades de Unix, *links* para documentação, *blogs*, *wikis*, etc... Em Unix/Linux é normalmente fácil compilar o código fonte utilizando um dos compiladores de C disponíveis no sistema operativo (frequentemente o GNU gcc). No site oficial do Ruby também se encontram documentos sobre a linguagem, as licenças em que esta é disponibilizada, *links* para vários *sites* da Internet com bibliotecas de extensão da linguagem, etc.

O presente guia consiste de 26 lições e foi originalmente escrito por Matz para a versão 1.4 do Ruby. Posteriormente à sua escrita já foram desenvolvidas as versões 1.6 e 1.8 do Ruby e, actualmente, em Outubro de 2006 a versão estável 'oficial' é o Ruby 1.8.5. Num futuro próximo aparecerá a versão 2.0 que irá incluir uma máquina virtual que permitirá compilar código Ruby para *bytecode*. Este projecto já está em andamento há algum tempo, denominando-se YARV .

Este "Guia do Utilizador" está organizado em 26 sub-documentos para facilitar a consulta. Cada um deles é uma unidade de conhecimento razoavelmente autónoma dedicada a um aspecto particular da linguagem.

Índice

Deixa-se já aqui uma observação preliminar: embora o autor desta tradução preze muito a Língua Portuguesa, e esta tradução pretenda contribuir directamente para o seu enriquecimento pela disponibilização de mais um conteúdo (embora não integralmente original) escrito em Português, considera que a manutenção de alguns termos não traduzidos do Inglês torna o texto mais escorreito e menos maçudo.

Na realidade não choca, hoje em dia, misturar num texto escrito em Português palavras como *hardware* e *software*, por exemplo, pois os conceitos são entendidos. Pela mesma razão iremos aqui utilizar a palavra *string* em vez de 'cadeia de caracteres', *array* em vez de fiada ou de vector, *hash* em vez de dicionário, entre outros termos que a seu devido tempo serão introduzidos.

Nota da tradução para Português

Este "Guia do Utilizador do Ruby" foi traduzido para o Português a partir de duas traduções Inglesas relativas à versão original, escrita em Japonês por Yukihiro Matsumoto, aka 'Matz', o criador da linguagem. Veja a nota de tradução mais abaixo, referente à primeira versão do guia em Inglês.

Também foi consultada a tradução 'melhorada' em Inglês realizada por Mark Slagell (o capítulo 26 destas notas é proveniente desta versão).

Tradução da versão Portuguesa

José A. Soares Augusto (Un. de Lisboa, Fac. de Ciências, Dep. de Física / Inesc-ID Lisboa)

Dirija comentários e correcções sobre esta versão Portuguesa para `jasa [arroba] inesc-id [ponto] pt`, página em http://calypso.inesc-id.pt/jasa

Seguem-se as **notas de tradução do guias em Inglês do Ruby utilizados como base deste documento.**

Translator's note

The (English version) is translated from the original version in Japanese by Matz.

Any questions for this document are welcome. There is also Ruby Language Reference Manual written by the author of Ruby. Check it out. Thanks!

Translators

GOTO Kentaro & Julian Fondren

Correspondence should be addressed to GOTO Kentaro: <URL:mailto:gotoken@notwork.org>

---

matz@netlab.co.jp

Prévio - Próximo - Índice

Considerações preliminares

Antes de mais, verifique se tem o Ruby instalado no seu computador. Numa consola, ou janela de comandos (onde a percentagem '%' serve de *prompt*) escreva

```
% Ruby -v
```

(a opção '-v' faz com que o interpretador mostre a versão do Ruby) e carregue na tecla 'ENTER' ou 'RETURN'. Se uma mensagem semelhante à que se segue fôr mostrada, o Ruby está instalado. A versão, a data, ou a plataforma são diferentes de sistema para sistema. Por exemplo, em Linux (muito antiguinho...) poderá ver

```
% Ruby -v
Ruby 1.1b5(98/01/19) [i486-linux]
%
```

e em Windows, muito recentemente:

```
C:\Ruby> Ruby -v
ruby 1.8.5 (2006-12-04 patchlevel 2) [i386-mingw32]
C:\Ruby>
```

Se o Ruby não estiver instalado, peça ao administrador do seu sistema para o instalar. É claro que pode fazê-lo você mesmo, pois o Ruby é software livre ('freeware') e não há quaisquer restrições, impostas pelos autores, à sua instalação e ao seu uso.

Vamos lá utilizar o Ruby! Pode-se escrever um programa na própria linha de comandos com a opção '-e'.

```
% ruby -e 'print "hello world\n"'
hello world
```

Em Unix (ou variantes) e Linux, um programa em Ruby pode ser guardado num ficheiro da seguinte forma:

```
% cat > test.rb
print "hello world\n"
^D
% cat test.rb
print "hello world\n"
% Ruby test.rb
hello world
```

onde (`^D` significa 'control-D', e neste caso termina a escrita de texto). Em MS-DOS pode-se fazer:

```
C:\Ruby> copy con: test.rb
print "hello world\n"
^Z
C:\Ruby> type test.rb
print "hello world\n"
C:\Ruby> Ruby test.rb
hello world
```

Para escrever código mais complexo do que o destes exemplos, deve-se utilizar um editor de texto.

O Ruby possui várias opções para a linha de comando. As mais úteis são as seguintes:

-0[DIGIT]	modo de parágrafo (com o separador DIGIT, um nómero em octal)
-a	modo "auto split"
-c	só verifica a sintaxe e sai
-e'SCRIPT'	executa o SCRIPT escrito na linha de comando
-F'DELIMITOR'	especifica o caracter delimitador
-i[extension]	modo de edição "in-place"
-I DIRECTORY	indica a directoria para carregamento de módulos ou bibliotecas
-l	remove NEWLINE da entrada e põe NEWLINE na saída
-n	ciclo automático
-p	ciclo automático nas linhas do ficheiro, mostrando essas linhas
-v	imprime a versão, e entra em modo tagarela ('verbose')

Por exemplo,

```
% Ruby -i.bak -pe 'sub "foo", "bar"' *.[ch]
```

executa a seguinte acção:

"substituir 'foo' por 'bar' em todos os ficheiros de programa ou de cabeçalhos ('headers') da linguagem C (sufixos '.c' ou '.h'), preservando os ficheiros originais (para ser possível recuperar o trabalho no caso de haver erros eventuais) em cópias com o sufixo '.bak' acrescentado ao nome original"

(e.g., se fôr feita uma alteração no ficheiro 'cab.h', uma cópia inalterada do original é colocada em 'cab.h.bak' sendo as substituições efectuadas directamente em 'cab.h').

Quanto ao comando

```
% Ruby -pe 0 file
```

é equivalente a `cat file` (ou `type file`, em MS-DOS) mas é mais lento que fazer 'cat' :-)

Para ver uma descrição completa das opções da linha de comando do Ruby escreva na consola um destes dois comandos: `Ruby -help` ou `Ruby -h`. Eis um exemplo no nosso sistema.

```
% Ruby -h
Usage: ruby [switches] [--] [programfile] [arguments]
  -0[octal]       specify record separator (\0, if no argument)
  -a              autosplit mode with -n or -p (splits $_ into $F)
  -c              check syntax only
  -Cdirectory     cd to directory, before executing your script
  -d              set debugging flags (set $DEBUG to true)
  -e 'command'    one line of script. Several -e's allowed. Omit [programfile]
  -Fpattern       split() pattern for autosplit (-a)
  -i[extension]   edit ARGV files in place (make backup if extension supplied)
  -Idirectory     specify $LOAD_PATH directory (may be used more than once)
  -Kkcode         specifies KANJI (Japanese) code-set
  -l              enable line ending processing
  -n              assume 'while gets(); ... end' loop around your script
  -p              assume loop like -n but print line also like sed
  -rlibrary       require the library, before executing your script
  -s              enable some switch parsing for switches after script name
  -S              look for the script using PATH environment variable
  -T[level]       turn on tainting checks
  -v              print version number, then turn on verbose mode
  -w              turn warnings on for your script
  -W[level]       set warning level; 0=silence, 1=medium, 2=verbose (default)
  -x[directory]   strip off text before #!ruby line and perhaps cd to directory
  --copyright     print the copyright
  --version       print the version
```

Não se vai explicar todas estas opções da linha de comando. Consulte o manual do Ruby se necessitar de informação adicional. (**Nota do Tradutor:** as opções da linha de comando são muito semelhantes às da linguagem Perl.)

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Características do Ruby

A descrição sintética do Ruby é feita com a seguinte frase: "o Ruby é uma linguagem de 'scripting' interpretada, destinada à programação orientada para objectos (OOP), simples e rápida". Eis alguns aspectos do Ruby que justificam este 'slogan'.

- rápida e fácil
  - é interpretada
  - as variáveis não são tipificadas ('typed'), i.e., não são associadas definitivamente a um tipo de dados, podendo conter dados de vários tipos ao longo da execução de um programa
  - é desnecessário declarar as variáveis
  - tem sintaxe simples
  - não é necessário gerir a memória (tem 'garbage collection' automática)
- suporta OOP *(Object-Oriented Programming)*
  - tudo é um objecto
  - suporta classes, herança, métodos, etc...
  - implementa o conceito do método singletão *(singleton)*
  - implementa 'mixin' com o conceito de módulo, em vez de herança a partir de múltiplas classes
  - possui iteradores e fechos ('closures')
- é uma linguagem de 'scripting'
  - é um interpretador
  - suporta expressões regulares e possui poderosas funções de manipulação de strings
  - permite aceder directamente ao sistema operativo (OS)
- miscelânea de características...
  - tem inteiros de precisão múltipla
  - tem um modelo de processamento que suporta excepções
  - efectua o carregamento dinâmico de bibliotecas

Está a ver o tipo de linguagem que é o Ruby? Não se preocupe com os conceitos, ainda desconhecidos para si, que acabaram de ser mencionados, pois serão explicados, posteriormente, neste guia.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Exemplos simples

Agora, vai-se examinar alguns programas simples em Ruby. Primeiro, considera-se um exemplo comum: a função factorial (n!).

A definição matemática do factorial de um número inteiro `n` é

```
n! = 1                (se n==0)
n! = n * (n-1)!       (se n!=0)
```

e é escrita em Ruby da seguinte forma

```
def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end
```

Há, nesta função, várias ocorrências de 'end'. Alguém comentou, acerca desta abundância -- '0 Ruby é como o Algol' --, certamente por razões históricas (o Algol utilizava 'end' a torto e a direito!) Na verdade, a sintaxe do Ruby é muito semelhante à da linguagem Eiffel.

Também se pode estranhar a ausência de 'return' no corpo da função. A função funciona, mesmo assim, pois o 'if' em Ruby "devolve" o valor calculado. É claro que se fôr acrescentado 'return' no final das cláusulas 'if' e 'else' não há qualquer problema, mas verifica-se que na ausência de 'return' o código é executado mais rapidamente.

Vamos utilizar esta função. Acrescente a linha `print fact(ARGV[0].to_i), "\n"` após a definição da função anterior e guarde tudo num ficheiro denominado, digamos, 'fact.rb'. `ARGV` é um vector ou sequência *(array)* pré-definido em Ruby, que contém os argumentos da linha de comando que se seguem ao nome do *script*, e `to_i` é o método que converte um objecto para inteiro se for possível (no exemplo que se segue, ARGV contém o objecto '4', do tipo *string*, que é convertido para o objecto 'integer' 4).

```
% ruby fact.rb 4
24
```

Será que funciona com 40? É que 40! dá erro de *overflow* numa calculadora vulgar...

```
% ruby fact.rb 40
815915283247897734345611269596115894272000000000
```

Funcionou! Na verdade, o Ruby pode lidar com qualquer inteiro que caiba na memória do computador: também se pode calcular 400!...

```
% ruby fact.rb 400
64034522846623895262347970319503005850702583026002959458684
44594280239716918683143627847864746326467629435057503585681
08482981628835174352289619886468029979373416541508381624264
61942352307046244325015114448670890662773914918117331955996
44070954967134529047702032243491121079759328079510154537266
72516278778900093497637657103263503315339653498683868313393
52024373788157786791506311858702618270169819740062983025308
59129834616227230455833952075961150530223608681043329725519
48526744322324386699484224042325998055516106359423769613992
31917134063858996537970147827206606320217379472010321356624
61380907794230459736069956759583609615871512991382228657857
95493616176544804532220078258184008484364155912294542753848
03558374518022675900061399560145595206127211192918105032491
```

```
0080000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000
```

A olho não se consegue verificar se está correcto, mas deve estar mesmo :-)

Quando se invoca o Ruby sem um argumento, o interpretador lê um *script* a partir da linha de comandos e executa-o após a entrada de dados terminar (o que é indicado com ^D == CTRL-D em Linux/Unix ou ^Z em Windows).

```
% ruby
print "hello world\n"
print "good-bye world\n"
^D
hello world
good-bye world
```

Há, porém, alturas em que não se quer utilizar o Ruby como um simples processador de comandos ('shell'). Utilizando o programa seguinte, pode executar *scripts* linha a linha. Não necessita de compreender este programa (para já!)

```
line = ''
indent=0
print "ruby> "
while TRUE
  l = gets
  if not l
    break if line == ''
  else
    line = line + l
    if l =~ /,\s*$/
      print "ruby| "
      next
    end
    if l =~ /^\s*(class|module|def|if|case|while|for|begin)\b[^_]/
      indent += 1
    end
    if l =~ /^\s*end\b[^_]/
      indent -= 1
    end
    if l =~ /\{\s*(\|.*\|)?\s*$/
      indent += 1
    end
    if l =~ /^\s*\}/
      indent -= 1
    end
    if indent > 0
      print "ruby| "
      next
    end
  end
  begin
    print eval(line).inspect, "\n"
  rescue
    $! = 'exception raised' if not $!
    print "ERR: ", $!, "\n"
  end
  break if not l
  line = ''
  print "ruby> "
end
print "\n"
```

Guarde este *script* no ficheiro `eval.rb` e execute-o (de facto, ele já está incluído na directoria `sample` da distribuição do Ruby),

```
% ruby eval.rb
ruby> print "hello world\n"
hello world
nil
ruby> ^D
%
```

'hello world', na segunda linha, é escrito pela execução do comando `print`, e `nil` é o valor devolvido pelo interpretador após processar `print`.

Com efeito, `nil` em Ruby significa 'void value' (valor 'vazio'), e é devolvido pelo interpretador de linha sempre que a instrução que inspecciona não retorna qualquer valor com significado, o que acontece no caso da instrução `print`.

De qualquer modo este curto programa é útil. Daqui em diante, neste documento `ruby>` será a *prompt* deste programa eval.rb que vai ser utilizado para executar os exemplos.

(Há outro programa na distribuição do Ruby, o `irb.rb`, que é ainda mais versátil. Experimente-o!)

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Strings (ou "cadeias de caracteres")

O Ruby, como qualquer linguagem de programação em geral, processa não apenas números mas também '*strings*' (cadeias de caracteres). A *string* em Ruby está enquadrada por aspas ("...") ou por plicas ('...'). Veja os seguintes exemplos:

```
ruby> "abc"      # string entre aspas...
"abc"
ruby> 'abc'      # ...exactamente igual à string entre plicas
"abc"
```

As diferenças entre as *strings* entre aspas e entre plicas são as seguintes: nas *strings* entre aspas, são interpretados caracteres especiais ou 'escapados' (*escaped characters*), começados por '\', e são executadas expressões escritas em Ruby, se enquadradas por '#{...}', cujo resultado é substituído na *string*. Veja os seguintes exemplos:

```
ruby> "\n"      # 'escaped'
"\n"
ruby> '\n'      # não 'escaped'
"\\n"
ruby> "\001"
"\001"
ruby> '\001'
"\\001"
ruby> "abcd #{5*3} efg"
"abcd 15 efg"
ruby> var = " abc "
" abc "
ruby> "1234#{var}5678"
"1234 abc 5678"
```

Uma *string* em Ruby é bastante mais versátil que em C ou noutras linguagens de programação. Por exemplo, a concatenação de *strings* é feita com o operador '+' e para repetir uma *string* n vezes escreve-se '* n' a seguir a ela:

```
ruby> "foo" + "bar"
"foobar"
ruby> "foo" * 2
"foofoo"
```

Note que para fazer a concatenação em C teria de escrever o seguinte código (muito mais complexo...):

```
char *s = malloc(strlen(s1)+strlen(s2)+1);
strcpy(s, s1);
strcat(s, s2);
```

Em Ruby o programador está livre de preocupações relativamente à gestão da memória. Nem sequer tem de entrar em conta com o espaço gasto pela *string*.

Em Ruby as *strings* podem ser processadas por bastantes operadores; apenas alguns vão ser aqui referidos. A concatenação (já mencionada, operador '+'):

```
ruby> word = "fo" + "o"
"foo"
```

A repetição (também já mencionada, com o operador '*'):

```
ruby> word = word * 2
"foofoo"
```

A extracção de um caracter (os caracteres são inteiros em Ruby):

```
ruby> word[0]
102             # 102 é o valor ASCII de 'f'
ruby> word[-1]
111             # e 111 é o valor ASCII de 'o'
```

A extracção de uma *substring*:

```
ruby> word[0,1]
"f"
ruby> word[-2,2]
"oo"
ruby> word[0..1]
"fo"
ruby> word[-2..-1]
"oo"
```

Em Ruby a *string* é uma sequência indexada por inteiros. O primeiro caracter está na posição 0 e o último estará na posição n-1, se a dimensão da *string* for n.

Para extrair o caracter na posição `k` escreve-se `string[k]`; se `k` for negativo, a extracção é feita a partir do fim (-1 refere-se à última posição, -2 à penúltima, etc...). Esta mesma notação é utilizada pelos *arrays* que iremos discutir mais adiante: a *string* pode ser vista como um *array* de caracteres.

Para extrair uma *substring*, também denominada por vezes de *slice* (fatia), de uma *string* podemos utilizar dois processos: dar os índices dos limites inferior e superior dessa *substring* separados por '..' (o operador de 'gama' ou *range*), ou dar o índice do primeiro caracter a extrair e a dimensão da *substring*, separados por uma vírgula ','. Exemplos destes dois casos, mostrados acima, são `word[0..1]` (da posição 0 até à posição 1) e `word[-2,2]` (começando na posição -2 e com dimensão 2), respectivamente.

O teste de igualdade entre *strings* faz-se com o operador '==':

```
ruby> "foo" == "foo"
true
ruby> "foo" == "bar"
false
```

Agora vai-se fazer um "pazzle" utilizando as operações com *strings* acima expostas. O pazzle chama-se 'acerte na palavra'. Note que a palavra "puzzle" é demasiado digna para aplicar ao programinha trivial que se segue `;-)`

```
palavras = ['foobar', 'baz', 'quux']   # conjunto de palavras para acertar
srand()                                # inicializa o gerador de números aleatórios
palavra = palavras[rand(3)]            # gera aleatoriamente a palavra a descobrir

print "adivinhe? "
while adivinha = STDIN.gets            # ciclo para leitura iterativa de palavras
  adivinha.chop!                       # retira \n do fim da palavra lida
  if palavra == adivinha               # se acertou termina o ciclo
    print "ganhou.\n"
    break
  else                                 # perdeu: vai continuar a pedir mais palavras
    print "perdeu.\n"
  end
  print "adivinhe? "
end
print "a palavra é ", palavra, ".\n"   # imprime a palavra certa e termina o programa
```

Não tem que compreender, para já, os detalhes deste programa, embora o tenhamos comentado brevemente para que quem tiver experiência de programação noutra linguagem possa seguir a sua lógica.

Um exemplo de uma sessão do jogo encontra-se transcrito abaixo (a minha resposta é precedida por `adivinha?`)

```
adivinha? foobar
perdeu.
adivinha? quux
perdeu.
adivinha? ^D
a palavra é baz.
```

Oh! Falhei muitas vezes, apesar de só haver uma probabilidade de errar de 1/3. Isto assim não é fixe -- não foi um bom exemplo...;-)

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Expressões Regulares

Vai-se agora construir um "pazzle" mais interessante. Desta vez testa-se se uma *string* coincide (*match*) com uma representação sintética de um conjunto de *strings*, ou seja, com um *padrão de string ('string pattern')*. Estes padrões são denominados **expressões regulares**.

Para ser possível compreender o exemplo, começa-se por indicar, na tabela abaixo, alguns dos símbolos com significado especial quando inseridos em expressões regulares.

```
[ ] gama de caracteres. (e.g., [a-z] representa qualquer letra minúscula).
\w letra ou dígito. O mesmo que [0-9A-Za-z_]
\W caracter que nem é letra nem dígito.
\s caracter 'branco' (ou espaço). O mesmo que [ \t\n\r\f]
\S caracter que não é do tipo 'espaço'.
\d dígito. O mesmo que [0-9].
\D caracter que não é dígito.
\b limite de palavra (fora da gama de caracteres).
\B que não é limite de palavra.
\b caracter relativo a 'página anterior' (0x08) (dentro de uma
especificação de gama)
.  qualquer caracter
*  zero ou mais repetições da expressão precedente
+  uma ou mais repetições da expressão precedente
{m,n}  pelo menos n repetições, mas não mais do que m repetições
da expressão anterior
?  0 ou 1 repetição da expressão precedente
|  a expressão precedente ou a expressão seguinte
( ) agrupamento
^  marca início da string
$  marca fim da string
```

Por exemplo, a expressão regular '`^f[a-z]+`' representa verbalmente "qualquer 'string' começada por um 'f', seguido por uma ou mais minúsculas". As expressões regulares são muito úteis para encontrar 'strings' (ou famílias delas) em blocos de texto, e por essa razão têm grande utilização em ambientes UNIX. Um dos comandos UNIX (ou Linux) onde são utilizadas intensivamente é o `grep`.

Para estudar as expressões regulares vai-se utilizar um pequeno programa. Guarde o programa seguinte num ficheiro denominado `regx.rb` e execute-o. (Nota: este programa só funciona correctamente em ambiente UNIX ou Linux, pois usa sequências de escape para vídeo invertido não disponíveis em MS-DOS.)

```
st = "\033[7m"
en = "\033[m"

while TRUE
  print "str> "
  STDOUT.flush
  str = gets # Lê 'string'
  break if not str
  str.chop!
  print "pat> "
  STDOUT.flush
  re = gets # Lê expressão regular
  break if not re
  re.chop!
  str.gsub! re, "#{st}\\&#{en}"
  print str, "\n"
end
print "\n"
```

O programa pede consecutivamente uma *string* e uma expressão regular, e mostra a coincidência entre elas utilizando o modo de vídeo invertido do terminal (os caracteres que coincidem com a expressão regular aparecem a vermelho). Para já, não se preocupe com os detalhes do programa, pois serão explicados posteriormente. Veja este exemplo:

```
 str> foobar
 pat> ^fo+
 foobar
 ~~~
```

foo é assinalado porque coincide com o padrão na expressão regular. Os "~~~" são mostrados para permitir que o resultado possa ser lido em browsers limitados a mostrar texto.

Vamos experimentar agora com várias entradas. Primeiro, procura-se dígitos:

```
 str> abc012dbcd555
 pat> \d
 abc012dbcd555
~~~ ~~~
```

O programa detecta ocorrências múltiplas:

```
str> foozboozer
pat> f.*z
foozboozer
~~~~~~~~
```

Não é só *fooz* que coincide neste exemplo, pois uma expressão regular tenta 'apanhar' sempre a maior *substring* que satifaz o padrão (i.e., a expressão regular é gananciosa -- *greedy*).

O padrão que se segue é demasiado complicado para que um iniciado o entenda à primeira. A sua função é detectar uma hora/data.

```
 str> Wed Feb 7 08:58:04 JST 1996
 pat> [0-9]+:[0-9]+(:[0-9]+)?
 Wed Feb 7 08:58:04 JST 1996
 ~~~~~~~~
```

Uma expressão regular em Ruby é enquadrada por '/.../' (i.e., é colocada entre *slashes*). O Ruby possui métodos que convertem automaticamente uma *string* para uma expressão regular.

```
 ruby> "abcdef" =~ /d/
 3
 ruby> "abcdef" =~ "d"
 3
 ruby> "aaaaaa" =~ /d/
 false
 ruby> "aaaaaa" =~ "d"
 false
```

'=~' é o operador de coincidência no âmbito das expressões regulares: devolve a posição inicial da coincidência na *string*, quando isso acontece, ou `false` caso não haja coincidência.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Arrays ('vectores') e *hashes* (dicionários ou 'vectores associativos')

O Ruby admite *arrays* (vectores, ou fiadas) como um tipo de dados pré-definido. Cria-se um *array* com o construtor '`[ ]`'. Os *arrays* em Ruby podem ter os elementos prenchidos com diferentes tipos de objectos. Por exemplo, aqui misturam-se inteiros com caracteres:

```
ruby> ary = [1, 2, "3"]
[1, 2, "3"]
```

Os *arrays* podem ser concatenados, ou repetidos, usando as técnicas válidas para as *strings*.

```
ruby> ary + ["foo", "bar"]
[1, 2, "3", "foo", "bar"]
ruby> ary * 2
[1, 2, "3", 1, 2, "3"]
```

Pode-se extrair parte de um *array* de forma análoga àquela aplicada às *strings*:

```
ruby> ary[0]
1
ruby> ary[0,2]
[1, 2]
ruby> ary[0..1]
[1, 2]
ruby> ary[-2]
2
ruby> ary[-2,2]
[2, "3"]
ruby> ary[-2..-1]
[2, "3"]
```

Os *arrays* e as *strings* são mutuamente convertíveis. Um *arr*ay converte-se numa *string* utilizando `join` e uma *string* é partida num vector com `split`.

```
ruby> str = ary.join(":")
"1:2:3"
ruby> str.split(":")
["1", "2", "3"]
```

Os *hashes* (ou 'vectores associativos', também conhecidos por *hashes* em Perl ou somente por dicionários em Python) são uma outra importante estrutura de dados nativa do Ruby. Num *hash* as chaves (ou índices) podem ter qualquer valor (i. e., não é necessário que sejam números inteiros, como nos *arrays*). Um *hash* tem como construtor um par de chavetas '{ }'.

```
ruby> hash = {1 => 2, "2" => "4"}
{1=>2, "2"=>"4"}
ruby> hash[1]
2
ruby> hash["2"]
"4"
ruby> hash[5]
nil
ruby> hash[5] = 10     # adiciona valor
ruby> hash
{5=>10, 1=>2, "2"=>"4"}
ruby> hash[1] = nil    # apaga valor
nil
ruby> hash[1]
nil
ruby> hash
{5=>10, "2"=>"4"}
```

Graças aos *arrays* e aos *hashes* já pré-definidos, em Ruby podem-se criar facilmente 'contentores' de dados estruturados.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

De novo os exemplos simples

Para explicar alguns detalhes do Ruby, examina-se de novo os exemplos.

O programa que se segue, que implementa a função factorial, já foi mostrado no Capítulo 3.

```
1: def fact(n)
2:   if n == 0
3:     1
4:   else
5:     n * fact(n-1)
6:   end
7: end
8: print fact(ARGV[0].to_i), "\n"
```

Uma vez que é a primeira vez que se analiza mais aprofundadamente um programa, vai-se explicá-lo linha a linha.

```
1: def fact(n)
```

Na primeira linha `def` é o termo (ou 'palavra-chave') que serve para definir uma *função* ou *método* em Ruby. No presente exemplo pode-se inferir ainda que a função `fact` tem `n` como único argumento.

```
2:   if n == 0
```

Nesta segunda linha o `if` é utilizado para testar uma condição. Quando ela se verifica (ou é verdadeira) é executada a linha, ou bloco de código, que se segue ao `if` e termina antes do `else`; quando a condição é falsa, será executada a linha de código imediatamente a seguir ao `else` (neste caso, o bloco de código entre `if` e `else` não será executado).

```
3:     1
```

O valor devolvido pelo bloco `if` é 1 se a condição for verdadeira.

```
4:   else
```

Se a condição for falsa, é executado o bloco de código situado entre `else` e `end`. No exemplo apenas a linha

```
5:     n * fact(n-1)
```

é executada neste caso, e o valor retornado do bloco `if...else...end` será o resultado da operação `n*fact(n-1)`.

```
6:   end
```

A instrução (composta) que consiste de todo o bloco `if` é terminado pelo primeiro `end` do programa.

```
7: end
```

A definição da função iniciada com `def` é terminada pelo segundo e último `end`.

```
8: print fact(ARGV[0].to_i), "\n"
```

Esta instrução escreve o resultado de `fact()` aplicada ao argumento fornecido na linha de comando.

ARGV é um *array*, pré-definido em Ruby, que contém os argumentos constantes da linha de comando quando o programa (ou *script*) é invocado. No presente exemplo `ARGV[0]` é o número cujo factorial se pretende calcular. Os membros (ou elementos) de ARGV são *strings* que têm de ser convertidas para números inteiros com o método `to_i`. O Ruby não converte automaticamente as *strings* para inteiros, como acontece, por exemplo, no Perl.

Passa-se, de seguida, a examinar o puzzle descrito no capítulo dedicado às *strings*.

```
 1:  palavras = ['foobar', 'baz', 'quux']
 2:  srand()
 3:  palavra = palavras[rand(3)]
 4:
 5:  print "adivinhe? "
 6:  while guess = STDIN.gets
 7:    guess.chop!
 8:    if word == guess
 9:      print "ganhou\n"
10:      break
11:    else
12:      print "perdeu.\n"
13:    end
14:    print "adivinhe? "
15: end # fim do 'while'
16:
17: print "a palavra é ", palavra, ".\n"
```

Se você é programador (em qualquer linguagem...), o programa é, de certeza, bastante claro. Mas há explicações importantes a fazer.

Neste programa é usada uma nova estrutura de controle: `while`. Enquanto a condição associada ao `while` for verdadeira, o bloco é executado integralmente até ao `end` que o finaliza, após o que o controlo retorna de novo à linha do `while` onde a condição é testada de novo. `srand()` serve para inicializar o gerador de números aleatórios. `rand(3)`, na linha 3, devolve um número inteiro aleatório que é, no máximo, 3, i. e., `rand(3)` vale 1, 2 ou 3.

Na linha 6 deste programa é lida a consola com o método `STDIN.gets`. Se for recebido um caracter de 'fim-de-ficheiro' (EOF ou '^D' na maioria dos terminais) `gets` devolve `nil` e termina o ciclo. Assim, o `while` vai ser repetido atá ser lido um '^D' (control-D).

`guess.chop!`, na linha 7, remove o último caracter da linha recebida. No presente caso, é removido o caracter *newline* ('\n'). Os métodos cujo nome termina em ponto de exclamação '!' indicam que é feita *uma modificação no objecto* ao qual são aplicados.

Finalmente, examina-se o programa das expressões regulares.

```
 1  st = "\033[7m"
 2  en = "\033[m"
 3
 4  while true
 5    print "str> "
 6    STDOUT.flush
 7    str = gets
 8    break if not str
 9    str.chop!
10    print "pat> "
11    STDOUT.flush
12    re = gets
13    break if not re
14    re.chop!
15    str.gsub! re, "#{st}\\&#{en}"
```

```
16   print str, "\n"
17 end
18 print "\n"
```

Como `while true` contém a condição `true`, o ciclo será infinito se for baseado somente nesta condição. Contudo, na prática não se espera que ele repita infinitas vezes, pois existem dois `break` nas linhas 8 e 13 para o terminar. Estes dois `break` ilustram também o uso do modificador `if`. Um modificador `if` executa a instrução à esquerda sempre que a condição à direita (i.e., que se segue ao `if`) é verdadeira (existe também no Perl, mas é raro noutras linguagens).

`chop!`, nas linhas 9 e 14, e `gsub!`, na linha 15, são métodos terminados por um ponto de exclamação '!'. Em Ruby pode-se colocar '!' ou '?' no final do nome de um método. Já diremos como '?' se relaciona com a funcionalidade do método; relativamente ao '!' já explicámos anteriormente:por convenção, em Ruby termina-se o nome dos métodos destrutivos (i.e., que alteram o objecto em que actuam) com '!'.

Do mesmo modo, termina-se com '?' o nome de um *predicado* (método que devolve verdadeiro, `true`, ou falso, `false`).

No presente exemplo, `chop!` remove o último caracter da *string* (é destrutivo) e o método `gsub!` ('global substitution') substitui, na *string* `str` à qual é aplicado, qualquer *substring* que concorde com a variável `re` pela expressão regular `"#{st}\\&#{en}"` dada como segundo argumento do método: também é destrutivo.

No segundo argumento `"#{st}\\&#{en}"` de `gsub!`, na linha 15, encontra-se a construção '\&', que indica que se deve substituir a *substring* que coincide com `re` precisamente neste ponto. Deve-se recordar que '\' (*backslash*) é expandido normalmente numa *string* entre aspas ("..."), pelo que tem que ser 'escapado' com mais um '\'.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Estruturas de controle

Neste capítulo vai-se examinar as estruturas de controle do Ruby. Já encontrámos `if`, `while` e `break`. As restantes vão ser aqui consideradas.

Já foi referido o modificador `if`. Existe também um modificador `while` que entra em acção neste exemplo:

```
ruby> i = 0; print i+=1, "\n" while i < 3
1
2
3
nil
```

Repare que é a instrução `print...`, anterior ao `while`, que irá ser repetida enquanto a condição `i<3` for verdadeira. Note também que se pode escrever várias instruções numa mesma linha se as separarmos com ponto e vírgula, ';'.

Usa-se a instrução `case` para escolher entre várias opções. O `case` é utilizado da seguinte forma:

```
ruby> case i
ruby| when 1, 2..5
ruby|   print "1..5\n"
ruby| when 6..10
ruby|   print "6..10\n"
ruby| end
6..10
nil
```

onde `2..5` é uma expressão que representa a gama (de inteiros) entre 2 e 5. Na linha que se segue, o resultado da expressão é utilizado para decidir se `i` está ou não incluído na gama:

```
(2..5) === i
```

No anterior exemplo `case` utiliza internamente o operador de relação `===` para verificar cada uma das condições (gamas). Como o Ruby é uma linguagem orientada para objectos (*Object-Oriented Programming* -- OOP), o operador de relação `===` é interpretado de uma forma concordante com o objecto presente na condição `when` ou, em terminologia OOP, temos aqui um exemplo de polimorfismo. Um operador polimórfico 'adapta-se' ao(s) objecto(s) a que é aplicado. Por exemplo, no código

```
ruby> case 'abcdef'
ruby| when 'aaa', 'bbb'
ruby|   print "aaa or bbb\n"
ruby| when /def/
ruby|   print "includes /def/\n"
ruby| end
includes /def/
nil
```

o `case` testa uma igualdade entre *strings* da primeira vez e a coincidência de uma *string* com uma expressão regular, da segunda (objectos diferentes em cada condição `when`).

Há quatro instruções de controle que permitem sair de um ciclo sem o terminar 'normalmente': a instrução `break` obriga, como em C, à saída incondicional do ciclo; a instrução `next` corresponde ao `continue` do C e faz com que não sejam executadas os restantes instruções até ao fim do ciclo e se continue com a próxima iteração.

Adicionalmente o Ruby possui `redo` que repete a actual iteração, i.e. 'volta atrás' devolvendo o controle ao início do ciclo. O seguinte código, em C, ajuda a compreender aquelas instruções (a quem souber C ;-):

```
while (condition) {
 label_redo:
   goto label_next        /* next */
   goto label_break       /* break */
   goto label_redo        /* redo */
   ;
   ;
 label_next:
}
label_break:
;
```

A quarta e última forma de sair de um ciclo é com `return`. Quando um `return` é executado dentro de um ciclo sai-se dele. Se o `return` tiver um argumento, o seu valor será devolvido para fora do ciclo; caso contrário, o Ruby assume um argumento `nil` e devolve-o.

Existe uma outra estrutura de controle repetitiva, o `for`, que é usado da seguinte maneira:

```
for i in obj
 ...
end
```

A instrução `for` obriga à execução do código até ao `end`, para cada elemento de `obj` (supondo que este objecto é uma colecção de vários elementos), colocando em cada iteração um deles na variável `i`. `For` é uma forma alternativa de iterador, um importante conceito em Ruby que será estudado posteriormente. Veja o exemplo de iterador que se segue, em que `i` toma, à vez, o valor de cada elemento de `obj`:

```
obj.each {|i|
 ...
}
```

Estas duas formas de realizar um ciclo são equivalentes, mas `for` é mais fácil de entender do que `each`. É esta a principal razão da existência do `for` em Ruby.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

O que é um iterador?

Os iteradores não são um conceito original do Ruby. São originários do CLU, uma linguagem que foi popular entre alguns programadores. Em Lisp também são utilizados iteradores, embora não sejam assim chamados. Uma vez que aquele conceito não é familiar à maioria dos leitores, vai-se explicá-lo com algum detalhe.

O verbo 'iterar' significa 'repetir a mesma operação'. Por isso, um *iterador* é algo que repete a mesma tarefa várias vezes.

Quando se escreve código está-se constantemente a utilizar ciclos. Em C os ciclos são normalmento do tipo `for` ou `while`. Por exemplo, atente-se neste ciclo `for`:

```
char *str;
for (str = "abcdefg"; *str; str++) {
  /* processe cada 'char' aqui */
}
```

A instrução de controle `for(...)` usa um idioma que requer o conhecimento detalhado do tipo de dados por parte do programador e, por isso, é uma instrução que propicia o aparecimento de erros. Nesta medida, o C é considerado uma linguagem de baixo nível.

Algumas linguagens de nível mais alto têm estruturas de controle para iterar. Considere o seguinte exemplo em dialecto *sh* ('shell'). Podemos questionar: "a *sh* é de mais alto nível que o C?" É, pelo menos neste caso...

```
for i in *.[ch]; do
  # fazer qualquer coisa a cada ficheiro .c ou .h, # (indexado em i)
done
```

Neste exemplo, a *sh* encarrega-se dos detalhes da iteração, manipulando os nomes dos ficheiros (localizados na directoria de trabalho, onde o *script* em questão está a ser executado) abarcados pela expressão regular e substituindo-os um a um em `i`. Este exemplo é um iterador de mais alto nível que o C, não é?

Mas há mais problemas. É bom quando a linguagem itera automaticamente nos tipos de dados nativos; porém, fica-se desapontado quando se pretende escrever ciclos num nível mais baixo, como em C, aplicados a tipos de dados definidos pelo utilizador... Em OOP, os programadores normalmente vão definindo tipos de dados uns atrás dos outros, o que faz com que este problema sej torne bastante sério!

Para ajudar nestes casos, todas as linguagens OOP incorporam mecanismos destinados a facilitar as iterações: por exemplo, algumas linguagens providenciam classes de controle de iteração. O Ruby permite a definição directa de estruturas de de controle de iteração: em Ruby estas estruturas definidas pelo utilizador denominam-se **iteradores**.

Vai-se examinar alguns exemplos. A *string* em Ruby possui iteradores e vamos apresentar alguns deles.

```
ruby> "abc".each_byte{|c| printf "<%c>", c}; print "\n"
<a><b><c>
nil
```

`each_byte` é um iterador que se aplica a cada caracter da *string*. Cada um deles é substituído, à vez, na variável local `c`. Este código seria traduzido da seguinte maneira 'a la C':

```
ruby> s="abc";i=0
0
ruby> while i<s.length
ruby|    printf "<%c>", s[i]; i+=1
ruby| end; print "\n"
<a><b><c>
nil
```

O iterador é mais simples e, provavelmente, mais rápido. Também é mais robusto a mudanças efectuadas na estrutura interna da classe *string*. Isto é uma característica requerida para a prática da 'boa programação'.

Outro iterador da classe *string* é `each_line`, que opera em cada linha.

```
ruby> "a\nb\nc\n".each_line{|l| print l}
a
b
c
nil
```

Descobrir delimitadores de linhas, gerar *substrings*, etc... muitas tarefas maçadoras podem ser feitas automaticamente pelo iterador. É um mecanismo muito conveniente.

A instrução `for`, analisada no capítulo anterior, realiza implicitamente a iteração através do iterador `each`. O `each` em *strings* funciona da mesma maneira que `each_line`, pelo que se pode reescrever o anterior exemplo de `each_line` com um `for`.

```
ruby> for l in "a\nb\nc\n"
ruby|   print l
ruby| end
a
b
c
nil
```

Dentro de um `for`, ou de um qualquer iterador, pode-se utilizar a estrutura de controle `retry`, que volta a repetir o ciclo do início (re-invoca a iteração):

```
ruby> c=0
0
ruby> for i in 0..4
ruby|   print i
ruby|   if i == 2 and c == 0
ruby|     c = 1
ruby|     print "\n"
ruby|     retry
ruby|   end
ruby| end; print "\n"
012
01234
nil
```

Podem-se construir iteradores 'à medida' em Ruby. Há algumas restrições, mas pode-se implementar iteradores bastante originais, o que é comum na prática da programação em Ruby.

A instrução `yield` aparece frequentemente na construção de um iterador. `yield` transfere o controle para o bloco com que se chamou o iterador (*'calling side'*). O exemplo que se segue, define um iterador `repeat` que executa o código passado no seu argumento um número `num` de vezes.

```
ruby> def repeat(num)
ruby|   while num > 0
ruby|     yield
ruby|     num -= 1
ruby|   end
ruby| end
nil
ruby> repeat(3) { print "foo\n" }
foo
```

```
foo
foo
nil
```

Com `retry` pode-se construir um iterador semelhante ao `while`, mas que não é prático, visto ser lento.

```
ruby> def WHILE(cond)
ruby|   return if not cond
ruby|   yield
ruby|   retry
ruby| end
nil
ruby> i=0; WHILE(i<3) { print i; i+=1 }
012nil
```

Já compreendeu o conceito de iterador?

Quando o programador define novos tipos de dados, deverá construir os iteradores apropriados. Neste aspecto, os anteriores exemplos `repeat()` e `WHILE()` não são muito úteis. Vai-se discutir iteradores de interesse prático num capítulo posterior, após abordarmos o conceito de *classe*.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Introdução à orientação para objectos

"Orientado para Objectos (OO)" é um chavão. "Orientado para objectos" qualquer coisa, soa inteligente nos dias que correm. O Ruby *também* é uma linguagem de 'scripting' orientada para objectos. Bem, para começar, o que significa "orientada para objectos"?

Parece que a definição de OO gera discussão, mesmo entre especialistas e, assim, toda a gente apresenta definições ligeiramente diferentes. Se ficarmos grudados nessas diferenças, não se pode sequer iniciar a discussão do conceito: por isso, vai-se já dar uma definição abrangente da OO.

A "orientação para objectos" mais não é do que um determinado "ponto de vista", ou perspectiva (o que é chamado normalmente de 'paradigma'), em programação. Resumidamente é

```
Um "ponto de vista" centrado em objectos.
```

Esta frase é demasiado sintética para ser entendida de imediato, e por isso vamos confrontá-la com a perspectiva de programação habitualmente aplicada.

Na perspectiva tradicional associada à computação, a resolução de problemas é abordada com uma tentativa de fazer uma decomposição em *dados* e em *procedimentos* (ou tarefas). Pode-se dizer que a filosofia é 'dividir para reinar' ;-)

Por exemplo, se for conhecida a "tarifa horária do Matz" (*allowance_data_of_matz*) e ela fôr dada como argumento a um "procedimento que calcula ordenados" (*allowance_function*), pode-se calcular a soma de dinheiro que o Matz vai ganhar este mês (*salary_of_this_month*). Isto pode ser escrito, num programa tradicional, da seguinte forma:

```
salary_of_this_month = allowance_function(allowance_data_of_matz)
```

Se o contabilista utilizar erroneamente os dados do patrão do Matz para calcular o salário, enganando-se no montante, a favor do Matz, o salário deste será indevidamente aumentado (e alguém comerá vivo o contabilista :-)

Por outro lado, numa perspectiva orientada para objectos os problemas são resolvidos através de uma decomposição em *objectos* e em *interacções entre esses objectos*. Voltando ao anterior exemplo, um objecto 'livro_de_pagamentos' (*allowance_ledger*) gere os dados básicos no cálculo de salários. Na filosofia orientada para objectos, um objecto não é inerte mas tem a capacidade de responder sempre que é questionado (ou seja, pode enviar mensagens para o exterior). Para saber qual vai ser o salário mensal, pode-se perguntar ao objecto 'livro_de_pagamentos' "qual vai ser o salário do Matz este mês?" O objecto dará uma resposta resultante de cálculos efectuados com os respectivos dados internos. Esta filosofia é esquematizada pela seguinte linha de código:

```
salary_of_this_month = salary_ledger.salary( this_month, matz )
```

Nesta perspectiva, o procedimento básico "perguntar ao livro_de_pagamentos" não será alterado se a forma interna ou externa de calcular o salário (e.g., a estrutura de dados ou o nível do posto de trabalho do Matz) forem alteradas no futuro, isolando estes detalhes internos de implementação do inquiridor do objecto.

Poderá não ter compreendido completamente esta ideia. De facto, é difícil explicar a um neófito um conceito tão etéreo como este paradigma da "orientação para objectos".

Pode-se, porém, avançar que a "orientação para objectos" apresenta os seguintes méritos, para vários níveis de profissionais da programação:

- **projectistas (ou arquitectos de sistemas):** podem projectar (ou "sentirem-se

capazes de o fazer") mais naturalmente, pois a modelação por objectos corresponde quase directamente a situações da vida real.
- **utilizadores:** (por vezes) podem notar mais claramente que o modelo é semelhante ao mundo real. Concretamente, a interface de utilização será mais *user-friendly* (amiga do utilizador) em modelos que, efectivamente, têm a ver com a realidade.
- **programadores:** o sistema torna-se robusto às mudanças que inevitavelmente vão sendo feitas à medida que o projecto evolui, devido à correspondência entre cada módulo e um objecto. Assim, a manutenção torna-se mais fácil.
- **programadores:** podem implementar sistemas novos reutilizando objectos (mais precisamente, *classes de objectos*) já existentes.

Note, porém, que estes méritos não funcionam indiscriminadamente em todos os casos! Qualquer ferramenta funciona, apenas, "se for utilizada com eficiência" e a OOP não é excepção. A OOP não é uma panaceia para todos os males: uma má utilização trará mais prejuízos que benefícios!

Ok! A explicação geral da OOP está terminada. Consulte os livros apropriados se pretender explicações mais detalhadas. Eu até recomendaria livros, se algum me ocorresse. Porém, relativamente à orientação para objectos, recomendar um livro avançado é difícil... e os livros mais simples por vezes têm ideias inadequadas ou mesmo erradas. Adiante... tente você encontrar um livro adequado ao seu nível!

Bem, para voltar ao assunto principal, como o Ruby é anunciado como sendo "uma linguagem de 'scripting' simples e fácil, interpretada, adequada à programação orientada para objectos", obviamente possui muitas características convenientes para OOP. Vamos intruduzi-las nos próximos capítulos.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Métodos

Neste guia, ainda não foi explicado o que é um *método*. Vai-se fazê-lo nesta secção.

Em OOP, quando se interroga um objecto, é ele próprio que realiza o processamento necessário à tarefa (ou tarefas) que lhe permitem dar uma resposta. Esse código de processamento, associado ao objecto, é demominado *método*.

Em Ruby a invocação de um método pertencente a um objecto (como já foi feito, muitas vezes, em capítulos anteriores) usa a seguinte sintaxe:

```
ruby> "abcdef".length
6
```

Neste exemplo, invocou-se o método `length` (comprimento) associado ao *objecto do tipo String* `denominado "abcdef"`.

Uma vez que o método é executado apenas quando é invocado (i.e., não tem memória), a resposta variará consoante o conteúdo da variável (objecto ou *instância de classe*).

```
ruby> foo = "abc"
"abc"
ruby> foo.length
3
ruby> foo = [1, 2]
[1, 2]
ruby> foo.length
2
```

O método `length` é mais um exemplo de polimorfismo, pois tanto pode ser aplicado a *strings* como a *arrays* ou a outras colecções. Os detalhes internos (e propositadamente escondidos do utilizador) dos procedimentos de obtenção da dimensão de *strings* ou de *arrays* são diferentes. Porém, o Ruby determina automaticamente, consoante o tipo de objecto, qual o processo adequado. Esta característica das linguagens OOP é denominada **polimorfismo**, como já se referiu. Se for invocado um método que o objecto receptor não conhece, será assinalado ('raised') um erro.

```
ruby> foo = 5
5
ruby> foo.length
NoMethodError: undefined method `length' for 5:Fixnum
```

Em Ruby tudo é um objecto! 5 é um objecto do tipo 'Fixnum'. Como este objecto não tem associado um método `length`, a sua invocação no anterior exemplo gerou o erro (NoMethodError:). Conclui-se, pois, que *o utilizador necessita de conhecer que métodos estão definidos e são aceites por cada objecto (ou classe), mas não precisa de saber os detalhes internos do seu processamento.*

Se forem especificados argumentos na invocação do método, deverão ser escritos entre parênteses e separados por vírgulas. Por exemplo, no caso de existirem 2 argumentos a forma geral de aplicação do método será:

```
object.method(arg1, arg2)
```

Os parênteses podem ser omitidos, a menos que possa existir ambiguidade na invocação. No anterior exemplo poderia escrever-se apenas

```
object.method arg1, arg2
```

obtendo-se, exactamente, o mesmo resultado.

Existe a variável especial `self` em Ruby, que representa o objecto que invoca o método. A invocação de métodos com `self` é utilizada com tanta frequência que foi criada uma abreviação do procedimento. A variável 'self' em

```
self.method_name(args...)
```

pode ser omitida, verificando-se que

```
method_name(args...)
```

tem o mesmo efeito. O que denominámos, mais atrás, de 'função', não é mais do que esta abreviação de um método aplicado a 'self'. Por esta razão, o Ruby poderá ser considerada uma linguagem orientada para objectos *pura*.

Em conclusão, note-se que estes "métodos funcionais" comportam-se de uma forma semelhante às "funções" suportadas pela maioria das outras linguagens de programação, para benefício dos programadores que não 'enxergam' que *tudo é um método*.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Classes

O mundo real está apinhado de objectos que podem ser classificados. Por exemplo, quando a minha filha (N.T: do Matz, obviamente...) de um ano vê um São Bernardo ou um Pastor Alemão, reconhece-o e grita 'béu-béu' ('BowHow'). Ela também poderá gritar 'béu-béu' se vir uma raposa...;-)

Piadas à parte, em termos de OOP o 'béu-béu' é uma **classe** e um objecto pertencente a uma dada classe é chamado de **instância** (dessa classe).

Para criar um objecto em Ruby, e em muitas outras OOPLs, habitualmente começa-se por definir a classe para estabelecer o comportamento do objecto, e depois cria-se uma sua instância que é o objecto pretendido. Vai-se pois definir uma classe para exemplificar este procedimento.

```
ruby> class Dog
ruby|   def bark
ruby|     print "Bow Wow\n"
ruby|   end
ruby| end
nil
```

Em Ruby, a definição da classe consiste de todo o código situado entre os termos `class` e `end`. Um bloco `def`, em termos de sintaxe de classes, define um *método da classe*.

Agora já temos a classe `Dog` e vai-se criar uma **instância** (ou objecto) desta classe.

```
ruby> pochi = Dog.new
#<Dog:0xbcb90>
```

Criou-se aqui uma (nova) instância da classe `Dog`, que foi armazenada na variável `pochi`. O método `new`, acessível e pré-definido para qualquer classe, gera uma nova instância. A variável `pochi` tem todas as propriedades definidas na classe `Dog`: por isso, consegue ladrar (bark).

```
ruby> pochi.bark
Bow Wow
nil
```

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Herança

Na maior parte das vezes, a classificação de objectos é feita hierarquicamente. Por exemplo, o 'gato' é um 'mamífero' e o 'mamífero' é um 'animal'. Algumas características na classificação de um objecto são propriedades herdadas de classificações básicas (ou antecessoras). Por exemplo, um 'animal' respira e, por isso, o 'gato' respira.

Esta **herança** de propriedades da classificação ao longo da árvore da hierarquia é implementada em Ruby da forma que se mostra de seguida.

Define-se a classe `Animal`

```
ruby> class Animal
ruby|   def respirar
ruby|     print "inala... e expira...\n"
ruby|   end
ruby| end
nil
```

e a classe `Gato`

```
ruby> class Gato<Animal
ruby|   def miar
ruby|     print "miau\n"
ruby|   end
ruby| end
nil
```

A classe `Gato` não possui qualquer definição sobre como *respirar*, mas vai herdá-la da classe `Animal`. Neste exemplo, foi apenas adicionada a propriedade `miar` à classe `Gato`.

```
ruby> tareco = Gato.new
#<Cat:0xbd80e8>
ruby> tareco.respirar
inala... e expira...
nil
ruby> tareco.miar
miau
nil
```

Como se pode ver, 'tareco', o 'Gato', consegue 'respirar' perfeitamente.

Porém, as propriedades da classe básica (denominada "classe progenitora" ou "superclasse") não são *sempre* herdadas pelas classes que dela derivam ("classes descendentes" ou "sub-classes"). Por exemplo, a 'ave' voa enquanto que o 'pinguim' não voa. Por outras palavras, os pinguins têm a maioria das outras propriedades das aves ('põem ovos', etc...) excepto voar. Neste exemplo, esta propriedade terá de ser *redefinida* na classe dos pinguins.

Vamos transcrever estas ideias para Ruby:

```
ruby> class Ave
ruby|   def poe_ovo
ruby|     # faz qualquer coisa...
ruby|   end
ruby|   def voar
ruby|     #...
ruby|   end
ruby|   #...
ruby| end
nil
```

```
ruby> class Pinguim<Ave
ruby|   def voar
ruby|     fail "Os pinguins não podem voar"
ruby|   end
ruby| end
nil
```

E assim se definiu a classe `Pinguim`.

Quando se utiliza herança para definir as propriedades que a sub-classe tem em comum com a superclasse, apenas é necessário adicionar ou redefinir as diferenças. Alguém chamou a este estilo *programação diferencial*. É um dos méritos da OOP.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Redefinição de métodos

Deve-se ter em atenção que é alterado o comportamento das instâncias numa subclasse quando são redefinidos os métodos da sua superclasse. Veja-se o seguinte exemplo:

```
ruby> class Humano
ruby|   def imprime_identidade
ruby|     print "Sou humano.\n"
ruby|   end
ruby|   def bilhete_de_comboio(idade)
ruby|     print "tarifa reduzida.\n" if idade < 12
ruby|   end
ruby| end
nil
ruby> Humano.new.imprime_identidade
Sou humano.
nil
ruby> class Estudante1<Humano
ruby|   def imprime_identidade
ruby|     print "Sou estudante.\n"
ruby|   end
ruby| end
nil
ruby> Estudante1.new.imprime_identidade
Sou estudante.
nil
```

```
ruby> class Estudante2<Humano
ruby|   def imprime_identidade
ruby|     super
ruby|     print "Sou estudante também.\n"
ruby|   end
ruby| end
nil
ruby> Estudante2.new.imprime_identidade
Sou humano.
Sou estudante também.
nil
```

Numa subclasse, quando se faz a redefinição de um método já existente na superclasse (i.e., cria-se na um método com o mesmo nome), pode-se invocar o método da superclasse com a instrução `super`. Os argumentos entregues a v são passados para o método original.

```
ruby> class Estudante3<Humano
ruby|   def bilhete_de_comboio(idade)
ruby|     super(11)        # passa 11 anos para a superclasse!
ruby|   end
ruby| end
nil
ruby> Estudante3.new.bilhete_de_comboio(25)
tarifa reduzida.
nil
```

Bem, talvez este não tenha siso um bom exemplo. Espero que tenha percebido como funciona a redefinição de métodos.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Mais acerca de métodos (controle de acesso)

Num capítulo anterior, disse-se que o Ruby não tem uma *função nativa*, mas apenas algo que é denominado de *método*.

Porém, existem vários tipos de métodos, o que complica um bocado as coisas. Estes tipos são denominados genericamente *controles de acesso* e têm como objectivo colocar restrições na invocação de métodos. Neste capítulo vai-se aprofundar este assunto.

Primeiro, vai-se definir uma função (ou método) no nível mais alto

```
ruby> def sqr(n)
ruby|   n * n
ruby| end
nil
ruby> sqr(5)
25
```

Quando `def` é utilizado no nível mais alto do programa, fora do âmbito de uma classe, como se estivéssemos a definir uma *função na forma clássica*, na realidade está-se implicitamente a acrescentar um novo método à classe `Object`. A classe `Object` é a superclasse de todas as outras classes, pelo que o método `sqr` vai poder ser invocado em todas as classes.

Uma vez que todas as classes podem invocar o método `sqr`, vamos tentar invocá-lo com `self`:

```
ruby> self.sqr(5)
ERR: private method 'sqr' called for main(Object)
```

Oops, houve um erro... deve haver um mal entendido!

Acontece que os métodos definidos no nível mais elevado (fora do bloco de definição de uma classe) só podem ser invocados como se fossem uma função clássica. Isto é consequência da seguinte política utilizada no Ruby: *"métodos que parecem funções na forma clássica, devem comportar-se e ser invocados como funções"* (ou seja, *"o que parece, é!"*).

Note, no entanto, que a mensagem de erro é diferente daquela emitida quando é invocado um método não definido:

```
ruby> undefined_method(13)
ERR: undefined method 'undefined_method' for main(Object)
```

Ah-ah! a mensagem de erro de há pouco foi `private` (privado) e não `undefined` (não definido).

Por outro lado, os métodos definidos dentro de uma classe podem ser invocados livremente no exterior:

```
ruby> class Teste
ruby|   def foo
ruby|     print "foo\n"
ruby|   end
ruby|   def bar
ruby|     print "bar -> "
ruby|     foo
ruby|   end
ruby| end
nil
ruby> teste = Teste.new
```

```
#<Teste:0xbae88>
ruby> teste.foo
foo
nil
ruby> teste.bar
bar -> foo
nil
```

No anterior exemplo o método `bar` invoca `foo` ao estilo de função.

Se um método é somente chamado como função, as invocações são obrigatoriamente realizadas dentro da classe onde foi definido ou das suas subclasses e, por isso, funciona como o *método protegido* em C++. Por outras palavras, pode-se proibir o acesso exterior a esse método.

Para alterar a condição de acessibilidade de um método utiliza-se os qualificadores `private` e `public`.

```
ruby> class Teste2
ruby|   def foo
ruby|     print "foo\n"
ruby|   end
ruby|   private :foo
ruby|   def bar
ruby|     print "bar -> "
ruby|     foo
ruby|   end
ruby| end
nil
ruby> teste2 = Teste2.new
#<Test2:0xbb440>
ruby> teste2.foo
ERR: private method 'foo' called for #<Teste2:0x8a658>(Teste2)
ruby> teste2.bar
bar -> foo
nil
```

Adicionando a linha '`private :foo`', verifica-se que o método `foo` não pode ser acedido do exterior como `test2.foo`.

Para ser possível aceder a um método privado, deve utilizar-se o código '`public` :*the_method_name*'. Não devem ser esquecidos os ':' à frente do nome do método: se forem esquecidos os ':', o nome é considerado como sendo a referência a uma variável local e não o nome de um método. Este erro de programação é muito difícil de detectar em programas de grande dimensão.

Um método privado funciona de forma semelhante a uma função clássica. Porém, traz benefícios, como por exemplo o facto de poderem existir métodos com o mesmo nome definidos em classes diferentes; isto é algo que não se pode fazer com funções: só podem ser definidas uma vez.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

O método singletão ('*singleton*')

O comportamento das instâncias é imposto pela classe que as define. Porém, por vezes pretende-se que uma dada instância, em particular, tenha um comportamento único, ligeiramente diferente do das suas 'irmãs de classe'. Na maioria das linguagens é obrigatório criar uma nova classe para essa instância. Em Ruby pode-se, simplesmente, acrescentar-lhe métodos sem ser necessário criar uma nova classe.

```
ruby> class SingletonTest
ruby|   def size
ruby|     print "25\n"
ruby|   end
ruby| end
nil
ruby> test1 = SingletonTest.new
#<SingletonTest:0xbc468>
ruby> test2 = SingletonTest.new
#<SingletonTest:0xbae20>
ruby> def test2.size
ruby|   print "10\n"
ruby| end
nil
ruby> test1.size
25
nil
ruby> test2.size
10
nil
```

Neste exemplo, `test1` e `test2` são ambas instâncias da classe `SingletonTest`. Porém, `test2` sofreu a redefinição do método `size`, pelo que `test1` e `test2` se comportam agora de maneira diferente. Um método criado para uma instância em particular é um **método singletão** (*singleton method*).

Usa-se métodos singletão em casos especiais como, por exemplo, em botões de interfaces de utilizador (GUI) iguais entre si em todos os aspectos, mas associados a acções diferentes quando premidos. Em Ruby, podem todos ser instâncias da mesma classe e pode-se definir a acção correcta para cada botão utilizando métodos singletão.

O conceito do método singletão não é original do Ruby, pois aparece em linguagens como o *CLOS*, o *Dylan*, etc... Há mesmo linguagens, como o *Self* e o *NewtonScript* que têm apenas métodos singletão. São denominadas *"linguagens baseadas em protótipo"*.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Módulos

O Ruby também possui o construtor **module** (módulo). Os módulos, em Ruby, são bastante semelhantes a classes, mas existem as seguintes três diferenças entre estas duas entidades:

1. o módulo não tem instância;
2. o módulo não tem subclasse;
3. o módulo é definido entre as directivas `module ... end`.

Na verdade, a classe `Module` dos módulos é a superclasse da classe `Class` das classes.

Numa perspectiva simplista, o módulo tem dois usos práticos. Primeiro, serve para agrupar métodos ou constantes com afinidades entre si. Por exemplo, o módulo `Math` na biblioteca padrão do Ruby desempenha este papel. Vejamos um exemplo.

```
ruby> Math.sqrt(2)
1.41421
ruby> Math::PI
3.14159
```

Oh, ainda não foi explicado o operador '`::`'! É simples: este é o operador que instancia constantes internas a um módulo ou a uma classe.

Para ser possível referir directamente os métodos ou as constantes de um módulo ou de uma classe, sem ser necessário utilizar o respectivo prefixo, usa-se a directiva `include` que tem como efeito transportar aqueles métodos e constantes para o *namespace* (espaço de nomes) actual. Este aspecto é ilustrado com o auxílio do código que se segue.

```
ruby> include Math
Object
ruby> sqrt(2)
1.41421
ruby> PI
3.14159
```

Outro uso dos módulos é na realização da denominada `mixin` (mistura). Esta utilização dos módulos é um assunto moderadamente complexo e vai ser explicada em detalhe.

Algumas OOPLs têm a possibilidade de *herdar* características de mais do que uma superclasse: esta possibilidade é denominada de *herança múltipla* ('multiple inheritance'). O Ruby não a possui, o que é propositado. Porém, pode-se conseguir o mesmo efeito utilizando módulos para fazer `mixin`.

Como já se disse, o módulo funciona de forma semelhante à classe. No entanto, os métodos e as constantes de um módulo não podem ser herdados mas podem ser acrescentados a outros módulos ou classes com a directiva `include`. Assim, com o `include` do módulo importa-se ou *mistura-se* (*mix*) as suas propriedades na (*in*) classe. Juntando as sílabas *mix* e *in*, obtém-se o termo *mixin*.

Por exemplo, se fôr feito o *mixin* do módulo `Enumerable` da biblioteca padrão, numa classe em que se definiu um método `each`, que devolve consecutivamente cada elemento da classe, esta automaticamente herda os métodos `sort`, `find`, etc..., associados a todas as estruturas enumeráveis.

Existem diferenças entre os conceitos de *herança múltipla* e de *mixin*:

- O módulo não permite a geração de instâncias (instanciação). É uma espécie de classe abstracta (que, por exemplo, existe explicitamente na definição do Java).

- A utilização de *mixins* mantém a hierarquização de classes sempre numa estrutura em árvore, pois a herança múltipla não é permitida.

A proibição da herança múltipla em Ruby inibe uma complexidade excessiva nas relações entre classes. Imagine uma situação em que uma classe apresenta várias superclasses, e a relação entre as instâncias tem uma estrutura de rede e não de árvore: este tipo de relações entre classes é demasiado complexo para ser entendido pelo cérebro humano (pelo menos pelo meu... :-), o que convida ao aparecimento de erros de programação difíceis de detectar e corrigir.

Nas linguagens que implementam herança múltipla é exigida muita disciplina ao programador para evitar aquelas situações. No entanto, em projectos envolvendo muitos programadores e que utilizam classes desenvolvidas por pessoas diferentes, é difícil controlar o problema.

O *mixin*, por outro lado, permite apenas 'adicionar à classe um conjunto de propriedades definidas no módulo'. Como não há instâncias de módulos, o problema atrás referido não se coloca.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Objectos do tipo Procedimento (Procedure)

Quando se programa, é frequente pretender-se fazer seguir um procedimento como argumento de um método, em vez de fornecer uma variável. Por exemplo, quando se processam sinais pode ser conveniente especificar procedimentos como argumentos para cada chegada do sinal.

Para especificar o processo para sinais em Ruby, usa-se o método `trap`.

```
ruby> trap "SIGUSR1", 'print "foobar\n"'
nil
ruby> Process.kill "SIGUSR1", $$
foobar
1
```

A acção (processo) realizada aquando da chegada do sinal `SIGUSR1` é a *string* que se segue a `trap "SIGUSR1"`. Funciona assim porque o Ruby é um interpretador. Porém, sem utilizar *strings* pode-se criar o procedimento directamente como um objecto.

```
ruby> trap "SIGUSR1", proc{print "foobar\n"}
nil
ruby> Process.kill "SIGUSR1", $$
foobar
1
```

O método `proc` gera um **objecto procedimento** (*'procedure object*') cujo conteúdo é o código colocado entre chavetas {}. Para executar o código incluído num *objecto procedimento* utiliza-se o método `call`.

```
ruby> proc = proc{print "foo\n"}
#<Proc:0x83c90>
ruby> proc.call
foo
nil
```

As utilizações dos objectos procedimento são semelhantes às dos apontadores para funções disponibilizados na linguagem C. Tem-se toda a liberdade de definir quaisquer funções ou métodos para estes fins, incluindo fazer a escolha do seu nome.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Variáveis

O Ruby possui três variedades de variável, uma variedade de constante e duas pseudo-variáveis. As variáveis e as constantes não têm que ser de um tipo pré-definido. Embora possa haver desvantagens numa linguagem que não força a declaração de tipos para as variáveis (isto é, o Ruby é uma *untyped language*), esta liberdade está de acordo com a classificação de *linguagem de programação fácil*, atribuída ao Ruby.

Pode-se inferir o tipo - variável ou constante - associado a um dado nome, a partir desse nome. Em Ruby **o nome permite distinguir imediatamente entre variável e constante**, por forma que não são necessárias declarações e é imediatamente visível a olho nu o tipo associado ao nome. Esta facilidade é conveniente para a programação, mas torna difícil a descoberta de erros de escrita (*typos*) em Ruby quando comparada com linguagens que obrigam à declaração das variáveis (*typed languages*).

A associação entre o nome e o tipo associado é inferida a partir do primeiro caracter da variável. Eis a convenção:

```
$       variável global
@       variável de instância
[a-z]   variável local
[A-Z]   constante
```

A excepção a esta regra são as duas **pseudo-variáveis** `self` e `nil`. Embora segundo a convenção devessem ser variáveis locais, visto começarem por uma minúscula, `nil` é na realidade constante e `self` é uma variável global mantida pelo compilador. Como só existem estas duas pseudo-variáveis em Ruby, não há grande perigo de haver confusões.

```
self    o actual objecto em execução
nil     valor nulo (indica 'false')
```

Veja-se o exemplo:

```
ruby> self
main
ruby> nil
nil
```

`main` é o objecto ao mais alto nível. O valor de `self` no interior de cada método pode variar. As pseudo-variáveis são só para ser lidas, por isso a escrita em `self` ou `nil` é proibida.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Variáveis globais

Uma variável global tem um nome começado por '$' e pode ser referida em qualquer parte do programa. Antes de ser inicializada, qualquer variável global tem o valor `nil`.

```
ruby> $foo
nil
ruby> $foo = 5
5
ruby> $foo
5
```

Pode-se invocar e modificar livremente as variáveis globais. Isto significa que o abuso da sua utilização é potencialmente perigoso, porque uma sua alteração propaga-se por todo o programa. Assim, *as variáveis globais devem ser utilizadas parcamente, e só em caso de extrema necessidade*. Quando usada, a variável global deverá ter uma denominação *forte*, para não coincidir com outra variável global definida posteriormente. O exemplo `$foo` é mau, pois *foo* é uma palavra muito vulgar - pelo menos nos livros de programação :-).

Pode-se rastrear uma variável global especificando um procedimento que é *executado de cada vez que a variável global muda*.

```
ruby> trace_var :$x, proc{print "$x = ", $x, "\n"}
nil
ruby> $x = 5
$x = 5
5
```

Quando uma variável funciona como *gatilho*, como neste exemplo, em que é invocada uma acção sempre que o seu valor é alterado, diz-se que é uma *variável activa*.

A maioria das denominações de variáveis globais em que um caracter não alfabético se segue ao '$' inicial, correspondem a variáveis nativas do sistema (Ruby) e têm um significado especial. Por exemplo, '$$' é o identificador do processo correspondente ao intrepretador de Ruby e só pode ser lido. Na lista que se segue encontram-se indicadas as variáveis de sistema mais importantes (*regexp* é uma expressão regular).

```
$!              mensagem de erro
$@              posição da ocorrência de um erro
$_              última 'string' lida com os métodos 'gets' ou 'readline'
$.              número da última linha lida pelo interpretador
$&              última 'string' apanhada pelo 'regexp'
$1, $2...       última 'string' apanhada no n-ésimo par de parênteses '()' do 'regexp'
$~              dados para o último sucesso do 'regexp'
$=              controla a sensibilidade a maiúscula/minúscula
$/              separador dos registos de entrada ("/n" por defeito)
$\              separador dos registos de saída ('nil' por defeito)
$0              nome do actual 'script' em execução pelo ruby
$*              argumentos da linha de comando com que foi chamado o ruby
$$              identificador do processo (PID) do interpretador ruby
$?              status do último processo 'filho' (child) a ser executado
```

Nesta lista, '$_' e '$~' têm âmbito local. Não é propagada a sua influência para fora do método quando o seu valor é mudado. Assim, no que respeita ao estatuto de *globalidade* (começam por '$'), estas duas variáveis devem ser consideradas *excepções* na lista acima.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Variáveis de instância

As variáveis de instância têm um caracterização especial: coloca-se '@' no início do seu nome. As variáveis de instância são únicas para o objecto referido por `self`. Objectos diferentes, mesmo pertencentes a uma mesma classe, têm valores diferentes para as respectivas variáveis de instância.

Em Ruby, as variáveis de instância não podem ser lidas ou alteradas por objectos exteriores, só o podendo ser através de métodos da classe. Uma variável de instância não inicializada tem o valor `nil`.

As variáveis de instância não precisam de ser declaradas. Isto implica uma estrutura flexível de objectos. Na realidade, as variáveis de instância em Ruby são criadas dinamicamente. Veja-se este exemplo:

```
ruby> class InstTest
ruby|   def set_foo(n)
ruby|     @foo = n
ruby|   end
ruby|   def set_bar(n)
ruby|     @bar = n
ruby|   end
ruby| end
nil
ruby> i = InstTest.new
#<InstTest:0x83678>
ruby> i.set_foo(2)
2
ruby> i
#<InstTest: @foo=2>
ruby> i.set_bar(4)
4
ruby> i
#<InstTest: @foo=2, @bar=4>
```

Neste bloco de código note que `i` não reporta a existência de `@bar` enquanto o método `set_bar` não é invocado. `@bar` não é criada pela definição da classe: é criada apenas pela execução explícita do método `set_bar`.

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Variáveis locais

As variáveis locais são definidas por um nome (ou identificador) começado por uma letra minúscula ou por '_'. Veja-se o seguinte exemplo:

```
ruby> @foo
nil
ruby> $foo
nil
ruby> foo
ERR: undefined local variable or method 'foo' for main(Object)
```

O quê? Deu erro!

Em Ruby, as variáveis locais necessitam de ser inicializadas por atribuição, o que as torna diferentes das outras variáveis, porque a primeira atribuição funciona simultaneamente como declaração. Quando se refere uma variável não declarada, o Ruby considera que se está invocando um método sem quaisquer argumentos. Assim, recebe-se a mensagem de erro `'...undefined local variable or method...'`.

Mas não é necessário efectuar a atribuição às variáveis locais na definição de métodos porque, neste caso, considera-se que as suas variáveis internas não têm valor atribuído.

O âmbito da declaração das variáveis locais é:

- `proc{ .... }`
- `loop{ .... }`
- `class .... end`
- `module .... end`
- `def .... end`
- em todo o programa (as excepções são os casos anteriores)

Uma variável local de um bloco iterador existe apenas dentro desse bloco se nele foi utilizada (ou definida) pela primeira vez no actual programa.

```
ruby> i0 = 1; print i0, "\n"; defined? i0
1
"local-variable"
ruby> loop{i1=5; print i1, "\n"; break}; defined? i1
5
FALSE
```

`defined?` é um operador que testa se o respectivo argumento está definido ou não. Como se pode observar, a variável `i1`, à qual é atribuído o valor 5 e é utilizada no ciclo `loop` pela primeira vez, não se encontra definida fora do `loop` (delimitado pelas chavetas '{....}').

Os objectos procedimento (`proc`) que se encontram num mesmo âmbito, partilham as variáveis locais.

```
ruby> i=0
0
ruby> p1 = proc{|n| i=n}
#<Proc:0x8deb0>
ruby> p2 = proc{i}
#<Proc:0x8dce8>
ruby> p1.call(5)
5
ruby> i
5
ruby> p2.call
5
```

Uma característica peculiar dos objectos procedimento é que a partilha do âmbito local é estendida para fora deste âmbito. Assim, a variável local `i` no anterior exemplo é partilhada mesmo quando `p1` e `p2` são passados para fora do âmbito. (Neste caso `i` pode ser acedida a partir de `p1` ou de `p2`). Veja este exemplo:

```
ruby> def foo
ruby|   i = 15
ruby|   get = proc{i}
ruby|   set = proc{|n| i = n}
ruby|   return get, set
ruby| end
ruby> p1, p2 = foo
#<Proc>, #<Proc>
ruby> p1.call
15
ruby> p2.call(2)
2
ruby> p1.call
2
```

Prévio - Próximo - Índice

Prévio - Próximo - Índice

A classe das constantes

Nota Prévia: este guia do Ruby foi escrito originalmente para uma versão antiga da linguagem (na tradução de Goto Kentaro e Julian Fondren, a invocação do comando `ruby -v` resulta em *ruby 1.1b5(98/01/19) [i486-linux]*). Nas versões recentes, 1.8.4 e 1.8.5, existem diferenças relativamente ao comportamento das constantes: algumas dessas diferenças são aqui mencionadas, mas não foi feita uma verificação exaustiva de todas as diferenças. Em caso de dúvida, experimente você próprio/a usando, por exemplo, o interpretador `irb.rb` já referido anteriormente, ou leia na net o *pickaxe book*, a 1ª edição do livro de Dave Thomas e Andy Hunt, *"Programming Ruby, the Pragmatic Programmers Guide"*, considerado a *bíblia do Ruby*, que pode ser consultado aqui http://www.rubycentral.com/book/ ou aqui http://whytheluckystiff.net/ruby/pickaxe/.

O próximo tópico é a classe de constantes. Uma constante tem um identificador (ou nome) começado por maiúscula, e é definida com uma atribuição. O acesso a uma constante não definida, ou a atribuição a uma constante já definida, causam uma mensagem de erro em versões antigas do Ruby:

```
ruby> FOO
ERR: Uninitialized constant FOO
ruby> FOO = 5
5
ruby> FOO
5
ruby> FOO = 5
ERR: already initialized constant FOO
```

Nas versões do Ruby superiores a 1.6 não causa erro dar um novo valor a uma constante:

```
ruby> FOO
NameError: uninitialized constant FOO
ruby> FOO = 5
5
ruby> FOO
5
ruby> FOO = 6
6
```

As constantes são referidas num âmbito de classe/módulo. Contrariamente às variáveis de instância, pode-se aceder a uma constante definida numa classe (mesmo as da classe mãe, ou de um módulo incluído nessa classe) ou num módulo.

```
ruby> module ConstTest
ruby|   CONST1 = 1
ruby|   CONST2 = 3
ruby|   CONST3 = 5
ruby|   print CONST1, CONST2, CONST3, "\n"
ruby|   def const_test
ruby|     print CONST3, CONST2, CONST1, "\n"
ruby|   end
ruby| end
135
nil
ruby> include ConstTest # makes consts be referred
Object
ruby> CONST1
1
ruby> CONST1 = 9  # can redefine consts of the ancestor (but should not)
9
ruby> const_test  # the above didn't affect to the ancestor's
531
```

O *namespace* da nova `CONST1` já não é o do módulo `ConstTest`, mas sim o seu exterior.

Pode-se fazer referência às constantes de um módulo com o operador '::'.

```
ruby> ConstTest::CONST1
1
```

Mas não é permitida uma substituição com o operador '::'.

```
ruby> ConstTest::CONST1 = 7
ERR: compile error in eval():
eval.rb:33: parse error
ConstTest::CONST1 = 7
                  ^
```

O operador '::' encontra-se disponível para uma constante que seja definida numa classe ou módulo. No anterior exemplo as constantes `CONST1`, `CONST2` e `CONST3` podem ser acedidas com '::' para o módulo `ConstTest`, mas não nos podemos referir a `CONST2` ou `CONST3` de um objecto da classe que faça `include` do módulo `ConstTest` (contudo, `CONST1` pode ser referida visto ter sido redefinida na classe).

Prévio - Próximo - Índice

Prévio - Próximo - Índice

A classe das constantes

Nota Prévia: este guia do Ruby foi escrito originalmente para uma versão antiga da linguagem (na tradução de Goto Kentaro e Julian Fondren, a invocação do comando `ruby -v` resulta em *ruby 1.1b5(98/01/19) [i486-linux]*). Nas versões recentes, 1.8.4 e 1.8.5, existem diferenças relativamente ao comportamento das constantes: algumas dessas diferenças são aqui mencionadas, mas não foi feita uma verificação exaustiva de todas as diferenças. Em caso de dúvida, experimente você próprio/a usando, por exemplo, o interpretador `irb.rb` já referido anteriormente, ou leia na net o *pickaxe book*, a 1ª edição do livro de Dave Thomas e Andy Hunt, *"Programming Ruby, the Pragmatic Programmers Guide"*, considerado a *bíblia do Ruby*, que pode ser consultado aqui http://www.rubycentral.com/book/ ou aqui http://whytheluckystiff.net/ruby/pickaxe/.

O próximo tópico é a classe de constantes. Uma constante tem um identificador (ou nome) começado por maiúscula, e é definida com uma atribuição. O acesso a uma constante não definida, ou a atribuição a uma constante já definida, causam uma mensagem de erro em versões antigas do Ruby:

```
ruby> FOO
ERR: Uninitialized constant FOO
ruby> FOO = 5
5
ruby> FOO
5
ruby> FOO = 5
ERR: already initialized constant FOO
```

Nas versões do Ruby superiores a 1.6 não causa erro dar um novo valor a uma constante:

```
ruby> FOO
NameError: uninitialized constant FOO
ruby> FOO = 5
5
ruby> FOO
5
ruby> FOO = 6
6
```

As constantes são referidas num âmbito de classe/módulo. Contrariamente às variáveis de instância, pode-se aceder a uma constante definida numa classe (mesmo as da classe mãe, ou de um módulo incluído nessa classe) ou num módulo.

```
ruby> module ConstTest
ruby|   CONST1 = 1
ruby|   CONST2 = 3
ruby|   CONST3 = 5
ruby|   print CONST1, CONST2, CONST3, "\n"
ruby|   def const_test
ruby|     print CONST3, CONST2, CONST1, "\n"
ruby|   end
ruby| end
135
nil
ruby> include ConstTest # makes consts be referred
Object
ruby> CONST1
1
ruby> CONST1 = 9  # can redefine consts of the ancestor (but should not)
9
ruby> const_test  # the above didn't affect to the ancestor's
531
```

O *namespace* da nova `CONST1` já não é o do módulo `ConstTest`, mas sim o seu exterior.

Pode-se fazer referência às constantes de um módulo com o operador '::'.

```
ruby> ConstTest::CONST1
1
```

Mas não é permitida uma substituição com o operador '::'.

```
ruby> ConstTest::CONST1 = 7
ERR: compile error in eval():
eval.rb:33: parse error
ConstTest::CONST1 = 7
                  ^
```

O operador '::' encontra-se disponível para uma constante que seja definida numa classe ou módulo. No anterior exemplo as constantes `CONST1`, `CONST2` e `CONST3` podem ser acedidas com '::' para o módulo `ConstTest`, mas não nos podemos referir a `CONST2` ou `CONST3` de um objecto da classe que faça `include` do módulo `ConstTest` (contudo, `CONST1` pode ser referida visto ter sido redefinida na classe).

Prévio - Próximo - Índice

Prévio - Próximo - Índice

Processamento de excepções

A execução e a excepção caminham lado a lado. Quando se abre um ficheiro, este pode não existir; quando se escreve num ficheiro, o disco pode estar cheio. Um programa, escrito em qualquer linguagem, é de má qualidade se não lidar correctamente com estas ocorrências excepcionais.

É aborrecido não saber se ocorreu uma excepção sem fazer constantemente uma verificação do tipo *"o processo terminou com 'status' OK?"* sempre que se pretende realizar tarefas potencialmente perigosas. Quase sempre, teremos problemas *a posteriori* se se continuar com o processamento normal sem interceptar aquela excepção. Assim, quando ela se dá o processo deverá tratar da ocorrência e parar, caso o erro seja catastrófico.

Em Ruby, a excepção dá-se apenas em situações *excepcionais*. Se não fôr interceptada, o programa pára. Assim, *em Ruby não há preocupações com problemas resultantes da execução continuada de um programa após haver uma excepção*.

```
ruby> file = open("/unexistant_file")
ERR: No such file or directory - /unexistant_file
```

Em C o programador tem que verificar explicitamente todas as situações críticas como, por exemplo, se um ficheiro foi ou não efectivamente aberto. Em Ruby pode-se assumir que 'se não houve uma excepção, o processo foi bem sucedido'. Em C, escrever-se-ia o seguinte código (equivalente ao anterior em Ruby):

```
FILE *file = fopen("/unexistant_file", "r");
if (file == NULL) { /* error recovery code...*/ }
```

A versão em Ruby é mais simples, pois não é necessario explicitar o processamento de erro.

Em Ruby o programa pára quando há uma excepção. Por vezes isto não é bom, pois pretende-se continuar com a sua execução após fazer algum processamento para corrigir a causa da excepção. Isto é possível de implementar capturando a excepção com um bloco '`begin`'. O seguinte código exemplifica este procedimento:

```
ruby> begin
ruby|   file = open("/unexistant_file")
ruby| rescue
ruby|   file = STDIN
ruby| end
#<IO:0x93490>
ruby> print file, "==", STDIN, "\n"
#<IO:0x93490> == #<IO:0x93490>
nil
```

Pode-se observar, pela igualdade dos dois códigos enviados pelo intrepretador, que `STDIN` substuíu `file` por causa da excepção devida à a falha do `open`. Uma vez que há erro, o bloco `begin` transfere o controle de processamento para o bloco que se segue à palavra `rescue`.

Ainda existe mais um nível de código para processar excepções: `retry`. Num bloco `rescue`, `retry` transfere o controle para o começo do bloco `begin`. Veja o seguinte exemplo:

```
ruby> fname = "unexistant_file"
ruby> begin
ruby|   file = open(fname)
ruby|   # something to use file
ruby| rescue
ruby|   fname = "existant_file"
```

```
ruby|   retry
ruby| end
#<File:0x68890>
```

Se ao tentar abrir o ficheiro *fname* é levantada uma excepção, no sub-bloco `rescue` é atribuído a *fname* o nome de um ficheiro existente, e a instrução `retry` faz com que se repita o processamento do código após o `begin`, isto é, repete-se a instrução `file = open(fname)` com um ficheiro já existente.

O fluxo de processamento naquele bloco de código é o seguinte:

- ocorre uma excepção na execução de `open`
- o controle é transferido para o `rescue`. `fname` é substituído
- o `retry` transfere o processamento para o início do bloco `begin`
- desta vez o `open` é bem sucedido
- continua-se o programa (que se segue ao `end`...)

Repare que se o ficheiro *`existant_file`* atribuído a `fname` na verdade não existir, o `retry` é repetido infinitamente e o programa empanca naquele bloco de código. Daqui se infere que se deve ter cuidado na utilização do `retry` para processar excepções.

Em Ruby, todas as bibliotecas levantam uma excepção quando ocorre um erro, e o utilizador pode mesmo definir e levantar excepções proprietárias. Para levantar uma excepção usa-se a instrução `raise`.

`raise` utiliza uma *string* de argumento que consiste da mensagem de explicação do erro. Este argumento pode (mas não deve) ser omitido, pois a mensagem de erro poderá ser examinada posteriormente lendo a variável pré-definida '`$!`'.

```
ruby> raise "test error"
ERR: test error
ruby> begin
ruby|   raise "test2"
ruby| rescue
ruby|   print $!, "\n"
ruby| end
test2
nil
```

Prévio - Próximo - Índice

Prévio - Índice

Porcas e parafusos

Este capítulo é dedicado a algumas **considerações práticas**. Foi escrito por *Marc Slagell*, e não consta da versão original do Guia escrita por *Yukihiro Matsumoto*.

Delimitadores de instruções

Algumas linguagens são pontuadas, utilizando frequentemente o ponto-e-vírgula ';' para terminar cada instrução. No Ruby segue-se a convenção utilizada nas *shells*, como a `sh` e a `csh`: múltiplas instruções numa mesma linha são separadas por ';' mas este sinal não é necessário no fim da linha para a terminar; o caracter de fim-de-linha (`'\n'`) é um terminador de instrução. Se uma linha terminar com uma *backslash* (`'\'`), o `'\n'` que se lhe segue é ignorado, o que permite estender uma única instrução lógica por várias linhas de programa.

Comentários

O Ruby segue a convenção comum nas linguagens de *scripting*, que é a de usar o sinal de cardinal (`#`) para começar um comentário: este comentário estende-se até ao fim da linha em que o sinal `#` foi escrito, sendo o seu conteúdo completamente ignorado pelo interpretador.

É boa prática de programação a escrita profusa de comentários juntamente com o código. Embora o código bem escrito seja quase *auto-documentado*, não faz mal fazer *anotações nas margens*, e é errado presumir que qualquer pessoa olha para um dado programa e entende, de imediato, a sua função. (Em termos práticos, você é a *outra pessoa* daqui a algumas semanas quando voltar a olhar para o código que escreveu hoje. Quantos de nós não examinamos código antigo e pensamos *"que raio é que isto faz aqui?"*...)

Para facilitar a escrita de grandes blocos de comentário, o interpretador de Ruby ignora tudo o que se encontra escrito entre uma linha começada por "`=begin`" e outra começada por "`=end`".

```
#!/usr/local/bin/Ruby

=begin
************************************************************************
  Este é um bloco de comentário, escrito em benefício dos leitores
  humanos (incluindo você!). O interpretador do Ruby ignora-o.
  Não é necessário um '#' no início de cada linha.
************************************************************************
=end

print "Hello, world.\n"
```

Organização do código

O interpretador de Ruby processa o código à medida que o lê. Não existe uma fase de compilação - o interpretador só faz uma passagem sobre o código; se aparecer algo não definido previamente, é considerado indefinido:

```
# este código resulta num erro "undefined method":

print successor(3),"\n"

def successor(x)
  x + 1
```

```
end
```

Isto não obriga, como poderia parecer à primeira vista, a organizar o código numa ordem estritamente descendente (*bottom-up*). Quando o interpretador processa a definição de um método, aceita referências não definidas. O programador terá de garantir que *elas já estarão definidas quando o método fôr invocado*:

```
# Conversão de graus Fahrenheit para Celsius, realizada em dois passos.

def f_to_c(f)
  scale (f - 32.0)  # Referência progressiva, mas OK.
end

def scale(x)
  x * 5.0 / 9.0
end

printf "%.1f is a comfortable temperature.\n", f_to_c( 72.3 )
```

Embora, neste aspecto, o Ruby pareça menos conveniente do queo Perl ou o Java, é uma linguagem muito menos restritiva do que escrever C sem usar protótipos (o que exigiria manter um ordenamento parcial sobre *'o que refere o quê...'*). Se fôr escrito o código de mais alto nível no final do programa (como se faz em Pascal), é garantido que funciona sempre. E isto não custa nada! Uma forma bastante eficaz de evitar problemas, é a de definir uma função `main` logo no topo do programa e invocá-la no fim:

```
#!/usr/local/bin/Ruby

def main
  # Ponha aqui o código de alto nível...
end

# ... ponha o código de suporte aqui, organizado da maneira que entender ...

main # ... e comece a execução aqui.
```

Também ajuda, na organização de programas, o facto do Ruby dispôr de ferramentas para partir programas complicados em blocos legíveis, reutilizáveis e logicamente relacionados. Já se discutiu o uso de `include` para aceder a módulos. Também são úteis as directivas `load` e `require`.

A directiva `load` funciona como se o ficheiro a que se refere fosse copiado e colocado exactamente nesse ponto (como a directiva `#include` do pré-processador do C). A directiva `require` é mais sofisticada, fazendo com que o código seja carregado apenas uma única vez e só quando for necessário. Há mais diferenças entre `load` e `require`; consulte o manual ou o FAQ (veja os *links* mais abaixo) para obter mais informação.

Para escrever...

Vários tópicos não foram tratados neste tutorial, incluindo:

- o método `initialize` (construtor de objectos)
- os métodos `to_s` e `inspect`
- listas com argumento de comprimento variável
- valores de defeito de parâmetros em argumentos de métodos
- suporte de editores de texto

Mas, neste ponto, já deverá estar pronto para mergulhar no *reference manual* (manual de referência) e aprender Ruby em mais detalhe. O *FAQ* (dúvidas comuns) e a *Library reference* (referência das bibliotecas) são também recursos valiosos.

Boa sorte, e feliz programação!

Prévio - Índice