Tutorial de Ruby

Eustáquio "TaQ" Rangel (eustaquiorangel@yahoo.com)

19 de julho de 2005

Este documento foi feito usando LaTeX.
Versão 0.1
Para atualizações, visite http://beam.to/taq/tutorialruby.php
A figura do logotipo de Ruby acima é de minha livre autoria, não representando um logotipo oficial

Sumário

Um pouco de bate-papo ...

Eu comecei a fazer esse tutorial como sempre começo fazendo as minhas coisas: necessidade de me lembrar depois.

Como estou começando a aprender a programar em Ruby agora (Nov/2004), quero fazer algumas anotações, me perdoem se em algum ponto eu fizer alguma bobagenzinha, mas depois de rirem disso me enviem um email (eustaquiorangel@yahoo.com) para que eu possa corrigir o que estiver errado. :-)

Tenho experiência como programador já faz alguns bons anos, e já passei por várias linguagens, e no decorrer do manual vou citar algumas comparações entre o Ruby e Java ™, Python, PHP, etc. Estou presumindo que você também programe em alguma linguagem, sendo que **esse não é um tutorial para aprender a programar**, e sim para aprender a programar em Ruby, com experiência de programação já adquirida em outras linguagens.

Ao mesmo tempo, não vou me ater muito em explicar as entranhas da linguagem aqui, apesar de mostrar algumas coisinhas. :-)

Alguns códigos também podem não estar com sua otimização "no talo", mas para efeito de tutorial, é melhor assim.

E sobre linguagens, deixe-me fazer um singelo protesto aqui. Não vou falar sobre qual é melhor ou não, por que discutir isso para mim é perda de tempo.

Não por que eu saiba a resposta de qual é, mas por que acho que são todas linguagens boas e o que difere nesse ponto são como as pessoas as veêm e as utilizam. Tenho várias citações de outras linguagens no decorrer do texto, mas em caráter puramente comparativo/informativo, e não nos termos de "aquela porcaria de XYZ faz isso dessa maneira".

Esse tipo de caça às bruxas é justamente o que você NÃO vai encontrar aqui. Já perdi muito tempo com esse tipo de coisa, e espero que ao invés de você perder também sente e escreva código na sua linguagem preferida do que ficar fazendo flamewars por aí. :-)

Então, vamos nessa.

Dedico esse trabalho à minha filhinha Ana Isabella, que me ilumina todos os dias.

Capítulo 1

Introdução

1.1 O que é Ruby

Para explicar o que é Ruby, eu faço uma tradução livre do que Yukihiro "Matz" Matsumoto, seu criador, diz a respeito dela em http://www.ruby-lang.org/en/20020101.html:

Ruby é uma linguagem de script interpretada para programação orientada a objetos de um modo fácil e rápido. Ela tem vários recursos para processar arquivos de texto e para fazer tarefas de gerenciamento de sistema (assim como o Perl). Ela é simples, direto ao ponto, extensível e portável.

Oh, preciso mencionar, é totalmente livre, o que significa não só livre de precisar pagar para usá-la, mas também a liberdade de usar, copiar, modificar e distribuí-la.

Recursos do Ruby

- Ruby tem uma sintaxe simples, parcialmente inspirada por Eiffel e Ada.
- Ruby tem recursos de tratamento de exceções, assim como Java ™e Python, para deixar mais fácil o tratamento de erros.
- Os operadores do Ruby são açucar sintático para os métodos. Você pode redefini-los facilmente.
- Ruby é uma linguagem completa e pura orientada à objetos. Isso significa que todo dado em Ruby é um objeto, não do jeito de Python e Perl, mas mais do jeito do SmallTalk: sem exceções. Por exemplo, em Ruby, o número 1 é uma instância da classe Fixnum.
- A orientação à objetos do Ruby é desenhada cuidadosamente para ser completa e aberta à melhorias. Por exemplo, Ruby tem a habilidade de adicionar métodos em uma classe, ou até mesmo em uma instância durante o runtime! Então, se necessário, a instância de uma classe *pode* se comportar diferente de outras instâncias da mesma classe.
- Ruby tem herança única, de propósito. Mas entende o conceito de módulos (chamados de Categories no Objective-C). Módulos são coleções de métodos. Toda classe pode importar um módulo e pegar seus métodos. Alguns de nós acham que isso é um jeito mais limpo do que herança múltipla, que é complexa e não é usada tanto comparado com herança única (não conte C++ aqui, pois lá não se tem muita escolha devido a checagem forte de tipo!).
- Ruby tem closures verdadeiras. Não apenas funções sem nome, mas com bindings de váriaveis verdadeiras.
- Ruby tem blocos em sua sintaxe (código delimitado por '{'...'}' ou 'do'...'end'). Esses blocos podem ser passados para os métodos, ou convertidos em closures.
- Ruby tem um garbage collector que realmente é do tipo marca-e-limpa. Ele atua em todos os objetos do Ruby. Você não precisa se preocupar em manter contagem de referências em libraries externas. É melhor para a sua saúde. ;-)

- Escrever extensões em C para Ruby é mais fácil que em Perl ou Python, em grande parte por causa do garbage collector, e em parte pela boa API de extensões. A interface SWIG também está disponível.
- Inteiros em Ruby podem (e devem) ser usados sem contar sua representação interna. Existem inteiros pequenos (instâncias da classe Fixnum) e grandes (Bignum), mas você não precisa se preocupar em qual está sendo utilizado atualmente. Se um valor é pequeno o bastante, um inteiro é um Fixnum, do contrário é um Bignum. A conversão ocorre automaticamente.
- Ruby não precisa de declaração de variáveis. Apenas usa a convenção de nomenclatura para delimitar o escopo das variáveis. Por exemplo: 'var' = variável local, '@var' = variável de instância, '$var'= variável global. E não precisa do uso cansativo do 'self' em cada membro da instância.
- Ruby pode carregar bibliotecas de extensão dinamicamente, se o sistema operacional permitir.
- Ruby tem um sistema de threading independente do sistema operacional. Então, para cada plataforma que você roda o Ruby, você tem multithreading de qualquer jeito, até no MS-DOS! ;-)
- Ruby é altamente portável: ela é desenvolvida em sua maioria no Linux, mas funciona em muitos tipos de UNIX, DOS, Windows 95/98/Me/NT/2000/XP, MacOS, BeOS, OS/2, etc.

1.2 Onde encontro Ruby

Você pode encontrar Ruby no seu site oficial na internet:

http://www.ruby-lang.org

Lá você encontra o código-fonte e versões para Windows ™. Compilar o código é rápido e fácil, no velho esquema

```
tar xvzf ruby-$<versao>$.tgz
./configure
make
make install
```

Procure na sua distribuição (se você usa GNU/Linux) algum pacote do Ruby. Se não encontrar, compile-o e o disponibilize em algum lugar. Eu fiz um pacote para o Slackware, está disponível aqui:

http://beam.to/taq/downloads.php

Se você pegar o pacote, pegue também a assinatura PGP do arquivo, importe a minha chave pública (784988BB) e faça a conferência.

Capítulo 2

Arrumando as coisas

2.1 Preparando sua fonte de consultas

É sempre bom ter uma referência das classes, módulos e métodos disponíveis em uma linguagem quando se está aprendendo.
Ruby vem com um utilitário chamado **ri** que facilita a sua consulta á essa documentação. Mas para ele poder consultar alguma coisa, primeiro temos que gera-lá.

Depois de instalado o Ruby, vá para o diretório onde você instalou *os fontes* e digite, por exemplo:

```
[taq@/usr/src/ruby-1.8.1]rdoc --ri *.c lib/*.rb
```

Isso vai produzir a documentação do **core** do Ruby, que vai ser armazenada no seu diretório **/.rdoc**.

Após isso podemos utilizar o ri dessa maneira, por exemplo:

```
[taq@ ]ri File
------------------------------------------------------------ Module:  File
A File is an abstraction of any file object accessible by the
program and is closely associated with class IO File includes the
methods of module FileTest as class methods, allowing you to write
(for example) File.exist?("foo").
...
```

Se utilizarmos um console que tenha suporte a cores, podemos criar um alias:

```
alias ri=ri -f ansi -T
```

Assim o exemplo acima fica

```
[taq@ ]ri File
------------------------------------------------------------ Module: File
A File is an abstraction of any file object accessible by the
program and is closely associated with class IO File includes the
methods of module FileTest as class methods, allowing you to write
(for example) File.exist?("foo").
...
```

2.2 Preparando o ambiente de trabalho

Não tem jeito melhor de aprender a programar em uma linguagem do que já ir colocando a mão na massa. Ruby tem um programinha muito bom para ir testando o código, o **irb** (interactive Ruby shell). Estilo do Python, que podemos chamar e já ir digitando as coisas e vendo os resultados.

Antes de abrirmos o irb, vamos alterar o arquivo **.irbrc**, que fica no seu diretório **home**.

ATENÇÃO
Estou usando GNU/Linux durante todo esse tutorial, e presumo que você está usando também. Lógico que você pode rodar Ruby em qualquer dos sistemas operacionais citados acima, mas eu realmente não tenho paramêtros para lhe dizer como fazer as configurações em outro sistema.

Insira

```
require 'irb/completion'
```

no arquivo.

Ok, agora chame o irb simplesmente digitando irb no terminal:

```
[taq@~]irb
irb(main):001:0$>$
```

Agora o irb já está pronto para ser usado. Vamos conhecer mais um pouco dele:

```
irb(main):001:0> i = 10
=> 10
irb(main):002:0> if i > 5
irb(main):003:1>    puts "maior que 5"
irb(main):004:1> else
irb(main):005:1*    puts "menor que 5"
irb(main):006:1> end
maior que 5
=> nil
irb(main):007:0>
```

Separando o prompt pelos dois pontos (:), temos algumas seções:

- O nome do programa que estamos rodando (main)
- O número de linhas que digitamos (e estamos!) no irb
- O nível de profundidade que estamos

O nível de profundidade é incrementado sempre que abrimos uma expressão que precisa de fechamento. Sempre que o irb verificar que o código precisa ser fechado, ele vai trocar o final do prompt por um asterisco, ou aspas-duplas se não fechamos uma String:

```
[taq@~]irb
irb(main):001:0> s = "um teste
irb(main):002:0" do irb"
=> "um teste \ndo irb"
irb(main):003:0>
```

Olhem o nível de profundidade alterado:

```
[taq@~]irb
irb(main):001:0> def teste(n)
irb(main):002:1>    if n > 10
irb(main):003:2>       puts "teste"
irb(main):004:2>    end
irb(main):005:1> end
=> nil
irb(main):006:0>
```

Capítulo 3

Conhecendo o básico da linguagem

3.1 Variáveis

Vamos destrinchar aqui alguns tipos básicos do Ruby, e como usar variáveis com eles.

Apesar de ter sido dito que TUDO em Ruby é um objeto, variáveis não são objetos, mas referências para esses.

As variáveis do Ruby são muito simples de serem utilizadas pois utilizam a **"duck typing"**, como é chamada carinhosamente a **tipagem dinâmica** do Ruby. Para os que vieram do Java ™, que tem uma **tipagem estática**, um bom exemplo seria

```
[taq@~]irb
irb(main):001:0> v = "teste"
=> "teste"
irb(main):002:0> v.class
=> String
irb(main):003:0> v = 1
=> 1
irb(main):004:0> v.class
=> Fixnum
irb(main):005:0>
```

Vale lembrar que Ruby também tem **tipagem forte**, isso é, vamos receber um erro se tentarmos fazer isso:

```
[taq@~]irb
irb(main):001:0> i = 1
=> 1
irb(main):002:0> s = "oi"
=> "oi"
irb(main):003:0> x = i+s
TypeError: String can't be coerced into Fixnum
  from (irb):3:in `+'
  from (irb):3
irb(main):004:0>
```

Então, não inventem de misturar uma coisa com a outra. :-)

Vamos dar uma olhadinha ali no output do irb. Só para dar uma "coceirinha", reparem que depois que declarei a primeira variável, pedi para ele verificar o valor do atributo **class** da variável. Foi retornado *String*, lógico, pois a variável v vale "teste" nesse caso.

Agora olhem na segunda atribuição de valor. Indiquei que o valor de v é 1, e verificando o class foi

retornado *Fixnum*, provando que o que o Matz disse na descrição do Ruby no começo desse tutorial é verdade: TUDO é objeto em Ruby.

Duvidam? Olhem que legal:

```
[taq@~]irb
irb(main):001:0> i = 1
=> 1
irb(main):002:0> i.methods
=> ["\%", "between?", "send", "<<", "prec", "\&", "object\_id", ">>", "size",
"singleton\_methods", "\_\_send\_\_", "round", "equal?", "taint", "method",
"id2name", "*", "next", "frozen?", "instance\_variable\_get", "divmod", "+",
"kind\_of?", "integer?", "chr", "to\_a", "instance\_eval", "-", "prec\_i",
"/", "type", "protected\_methods", "extend", "truncate", "to\_sym", "|",
"eql?", "modulo", "instance\_variable\_set", "~", "hash", "zero?", "is\_a?",
"to\_s", "prec\_f", "singleton\_method\_added", "class", "tainted?",
"private\_methods", "\^", "step", "untaint", "+@", "upto", "-@", "remainder",
"id", "**", "nonzero?", "to\_i","<", "inspect", "<=>", "==", "floor", ">",
"===", "succ", "clone", "public\_methods", "quo", "display", "downto", ">=",
"respond\_to?", "<=", "freeze", "abs", "to\_f", "\_\_id\_\_", "coerce", "=~",
"methods", "ceil","nil?", "dup", "to\_int", "div", "instance\_variables",
"[]", "instance\_of?", "times"]
```

Esses são os **métodos públicos da instância** da classe (seria equivalente à uma chamada **i.public_methods**).

Para vermos os **métodos privados da instância da classe**, podemos usar **i.private_methods**.
Lembrem-se que estamos falando de métodos da **instância da classe**, mais tarde vamos falar dos **métodos da classe**, sendo eles públicos, privados e protegidos. Mas isso mais tarde.

Agora vamos continuar a falar de variáveis.

3.1.1 Escopo das variáveis

Para declarar uma variável no escopo privado, é só fazer isso:

```
[taq@~]irb
irb(main):001:0> i = 1
=> 1
irb(main):002:0> puts i
1
=> nil
irb(main):003:0>
```

Aqui tem um método novo, o **puts**, que é um **método de kernel** do Ruby. Isso implica que podemos o utilizar sem precisar especificar o objeto à que ele faz parte (Kernel). A declaração completa é **Kernel.puts**. O puts nada mais faz que imprimir o resultado da expressão que você passou para ele, quebrando a linha no final.
Para declarar uma váriavel **pública**, utilizamos

```
[taq@~]irb
irb(main):001:0> $i = 1
=> 1
irb(main):002:0>
```

e pronto. Declarando a variável com um caracter **$** na frente do nome já a torna pública.

Tem mais um tipo de variável, que é a de instância de classe, mas só vamos falar nela quando chegarmos no assunto de classes. :-)

3.2 Tipos básicos

3.2.1 Blocos de código

Antes de apresentar qualquer um dos tipos, quero mostrar uma coisa essencial em Ruby, que são os blocos de código.
Blocos de código são umas das boas sacadas de Ruby, e podem ser pedaços de código delimitados por { e }:

```
{puts 'Oi mundo!'}
```

ou, para blocos maiores, delimitados por **do** e **end**:

```
do
   puts 'Oi mundo!'
   puts 'Aqui tem mais espaço'
end
```

A convenção é usar as chaves para blocos de uma linha e **do ... end** para blocos maiores.

Vamos ver muita coisa sobre blocos de código mais adiante. Blocos de código, juntamente com *iterators*, são **muito** utilizados em Ruby, são uma das peças-chaves da linguagem e vão aparecer em muitos exemplos dos tipos básicos abaixo. Se você não entendeu muita coisa, não tem problema, vamos ter uma explicação mas extensa sobre isso logo.

3.2.2 Números

Podemos ter variáveis dos tipos **inteiro** ou **ponto flutuante**.
Exemplos de inteiros:

```
[taq@~]irb
irb(main):001:0> i1 = 1
=> 1
irb(main):002:0> i2 = 123
=> 123
irb(main):003:0> i3 = -1000
=> -1000
irb(main):004:0> i4 = 3_000_000_000
=> 3000000000
irb(main):005:0> puts i1.class
Fixnum
=> nil
irb(main):006:0> puts i2.class
Fixnum
=> nil
irb(main):007:0> puts i3.class
Fixnum
=> nil
irb(main):008:0> puts i4.class
Bignum
=> nil
irb(main):009:0>
```

Não podemos usar vírgulas em variáveis, mas se acharmos que fica mais legível, podemos usar **sublinhado** para separar os milhares, como no exemplo dos 3 bilhões acima (alguém tem isso na conta? :-).

Outra coisa que eu fiz ali foi mostrar o tipo de cada variável. Como o Matz disse também na descrição inicial, dependendo do número que você utiliza, o Ruby já troca automaticamente o tipo. As três primeiras variáveis são do tipo **Fixnum**, enquanto a última não coube no Fixnum e foi automaticamente

usado o **Bignum**.

Exemplos de ponto flutuante:

```
[taq@~]irb
irb(main):001:0> f1 = 1.23
=> 1.23
irb(main):002:0> f2 = -57.08
=> -57.08
irb(main):003:0> f3 = 15.058e-4
=> 0.0015058
irb(main):004:0> f1.class
=> Float
irb(main):005:0> f2.class
=> Float
irb(main):006:0> f3.class
=> Float
irb(main):007:0>
```

Também podemos criar números usando as formas hexadecimal, octal e binária:

```
[taq@~]irb
irb(main):001:0> hex = 0xFF
=> 255
irb(main):002:0> oct = 0377
=> 255
irb(main):003:0> bin = 0b11111111
=> 255
irb(main):004:0>
```

3.2.3 Booleanos

São os valores **true** e **false**. Reparem que estão em **minúsculas**.
Aproveitando o gancho dos booleanos, o valor **nulo** em Ruby é definido como **nil**, sendo que *false e nil são os únicos valores que são reconhecidos como valores indicando falso*, mas são de classes e comportamentos diferentes:

```
[taq@~]irb
irb(main):001:0> t = true
=> true
irb(main):002:0> f = false
=> false
irb(main):003:0> n = nil
=> nil
irb(main):004:0> t.class
=> TrueClass
irb(main):005:0> f.class
=> FalseClass
irb(main):006:0> n.class
=> NilClass
irb(main):007:0>
```

A recomendação é que *métodos predicados* (que terminam com ? no final, mais sobre isso depois) retornem true ou false, enquanto outros métodos que precisam indicar falha retornem nil.

3.2.4 Strings

Qualquer "palavra" ou conjunto de palavras podem ser armazenados como uma String:

```
[taq@~]irb
irb(main):001:0> s1 = "oi"
=> "oi"
irb(main):002:0> s2 = "oi, amigo!"
=> "oi, amigo!"
irb(main):003:0> s3 = "oi, 'mano'!"
=> "oi, 'mano'!"
irb(main):004:0> s1.class
=> String
irb(main):005:0>
```

Vamos aproveitar que estamos falando de Strings e lembrar que, como foi dito acima, variáveis são referências para objetos. Para ter uma prova disso, vamos ver um exemplo:

```
[taq@~]irb
irb(main):001:0> my_nick = "TaQ"
=> "TaQ"
irb(main):002:0> my_nick_copy = my_nick
=> "TaQ"
irb(main):003:0> my_nick[0] = 'S'
=> "S"
irb(main):004:0> puts my_nick
SaQ
=> nil
irb(main):005:0> puts my_nick_copy
SaQ
=> nil
irb(main):006:0>
```

O que aconteceu aqui? Eu só troquei o primeiro caracter da variável my_nick, não da my_nick_copy (falando nisso, você pode manipular Strings usando [] da mesma maneira que em outras linguagens)!

Bom, nesse caso, my_nick_copy aponta para o mesmo lugar onde se encontra o valor de my_nick, por isso que quando você alterou uma, alterou a outra junto.

Se não quisermos que um objeto seja alterado, podemos usar o método **freeze** nele:

```
[taq@~]irb
irb(main):001:0> my_nick = "TaQ"
=> "TaQ"
irb(main):002:0> my_nick_copy = my_nick
=> "TaQ"
irb(main):003:0> my_nick.freeze
=> "TaQ"
irb(main):004:0> my_nick[0] = 'S'
TypeError: can't modify frozen string
  from (irb):4:in `[]='
  from (irb):4
irb(main):005:0> my_nick_copy[0] = 'S'
TypeError: can't modify frozen string
  from (irb):5:in `[]='
  from (irb):5
irb(main):006:0> my_nick
=> "TaQ"
irb(main):007:0> my_nick_copy
=> "TaQ"
irb(main):008:0>
```

Para ter resultados distintos entre as variáveis, baseando o valor de uma na outra, teríamos que fazer:

```
[taq@~]irb
irb(main):001:0> my_nick = "TaQ"
=> "TaQ"
irb(main):002:0> my_nick_copy = my_nick.dup
=> "TaQ"
irb(main):003:0> my_nick[0] = 'S'
=> "S"
irb(main):004:0> puts my_nick
SaQ
=> nil
irb(main):005:0> puts my_nick_copy
TaQ
=> nil
irb(main):006:0>
```

O método **dup duplica** o objeto, criando uma nova cópia, que foi atribuída à variável my_nick_copy. Então, quando trocamos o primeiro caracter de my_nick, estamos alterando *somente* ela e não mais my_nick_copy junto.

Podemos usar uma feature chamada de **heredoc** para criar "stringonas":

```
[taq@~]irb
irb(main):001:0> s = <<FIM
irb(main):002:0" Uma String realmente
irb(main):003:0" grande com alguns
irb(main):004:0" saltos de linhas, para
irb(main):005:0" demonstrar o heredoc!
irb(main):006:0" FIM
=> "Uma String realmente\ngrande com alguns\nsaltos de linhas, para\ndemonstrar o heredoc!\n"
irb(main):007:0> puts s
Uma String realmente
grande com alguns
saltos de linhas, para
demonstrar o heredoc!
=> nil
irb(main):008:0>
```

Definimos uma variável, que tem seu valor atribuído logo após encontrarmos uma string semelhante à que vem depois de <<. No caso usei FIM, mas fica a seu critério qual string usar de terminador. Se quiser escrever PINDAMONHANGABA, pode. :-)

Para concatenar Strings, podemos usar + ou << (lembraram de C++?):

```
[taq@~/trash]irb
irb(main):001:0> s = "Oi, "
=> "Oi, "
irb(main):002:0> s + "mundo!"
=> "Oi, mundo!"
irb(main):003:0> s << "mundo!"
=> "Oi, mundo!"
irb(main):004:0>
```

Fica a dica que << é mais rápido, pois não gera um novo objeto como +:

```
[taq@~]irb
irb(main):001:0> s = "teste"
=> "teste"
irb(main):002:0> s.object_id
=> -605086198
```

```
irb(main):003:0> s += " concatenado"
=> "teste concatenado"
irb(main):004:0> s
=> "teste concatenado"
irb(main):005:0> s.object_id
=> -605094768
irb(main):006:0> s2 = "teste"
=> "teste"
irb(main):007:0> s2.object_id
=> -605105148
irb(main):008:0> s2 << " concatenado"
=> "teste concatenado"
irb(main):009:0> s2
=> "teste concatenado"
irb(main):010:0> s2.object_id
=> -605105148
irb(main):011:0>
```

No caso de **s += "concatenado"**, é interpretado como **s = s + "concatenado"**, por isso que gera um novo objeto.

3.2.5 Constantes

Constantes são variáveis que tem seu nome começado com **uma letra maiúscula**.
"O que? Só isso?" você poderia me perguntar. Só.
"Mas não fica meio confuso com nomes de classes que começam em maiúsculas?" uhnnn até fica, mas é prático.
"Então, por exemplo, ao invés de eu ter em Java ™: **public static final double Pi = 3.1415926;** *eu só teria que escrever Pi?"*. É isso aí.
"Tem certeza que isso é uma constante?". Ok, vamos ver:

```
[taq@~]irb
irb(main):001:0> Pi = 3.1415926
=> 3.1415926
irb(main):002:0> Pi = 10
(irb):2: warning: already initialized constant Pi
=> 10
irb(main):003:0>
```

Viu? Declarei a variável, e quando fui alterar seu conteúdo, fui informado que não deveria, pois é uma constante.
Fica um fato curioso que Ruby, diferente de outras linguagens, permite que você altere a constante, (*"mas não era uma constante?"*) mas gera um warning alertando sobre isso.

3.2.6 Ranges

Intervalos. Podemos declarar de duas maneiras: incluindo ou não o último valor a que nos referimos em sua declaração. Por exemplo:

```
[taq@~/code/java]irb
irb(main):001:0> r = 0..10
=> 0..10
irb(main):002:0> r2 = 0...10
=> 0...10
irb(main):003:0>
```

No primeiro caso, a range vai conter números de 0 até 10, *incluindo* o 10. No segundo caso, vai de 0 até 9.
Como regrinha dos pontinhos, *menos é mais*, quando se tem menos pontos, se tem mais valores na range.

3.2.7 Arrays

Arrays podem conter vários tipos de objetos (ou não):

```
[taq@~]irb
irb(main):001:0> a = ['apenas',1,'teste']
=> ["apenas", 1, "teste"]
irb(main):002:0>
```

Para Arrays com somente Strings, podemos usar o atalho **%w**, ficaria assim:

```
[taq@~]irb
irb(main):001:0> a = %w(homer marge bart lisa maggie\ o\ bebezinho)
=> ["homer", "marge", "bart", "lisa", "maggie o bebezinho"]
irb(main):002:0>
```

Se precisarmos de uma String com mais de uma palavra, usamos o caracter como mostrado acima.

Inserir elementos no Array é facil, usando **push**:

```
[taq@~]irb
irb(main):001:0> a = []
=> []
irb(main):002:0> a.push "um"
=> ["um"]
irb(main):003:0> a.push "dois"
=> ["um", "dois"]
irb(main):004:0> a.push "tres"
=> ["um", "dois", "tres"]
irb(main):005:0> a.push "quatro"
=> ["um", "dois", "tres", "quatro"]
irb(main):006:0>
```

ou usando o operador <<:

```
[taq@~]irb
irb(main):001:0> a = []
=> []
irb(main):002:0> a << "um"
=> ["um"]
irb(main):003:0> a << "dois"
=> ["um", "dois"]
irb(main):004:0> a << "tres"
=> ["um", "dois", "tres"]
irb(main):005:0>
```

Podemos fazer com um Array algo semelhante às **list comprehensions** do Python, que é processar os elementos do Array e criar um novo, baseado em uma condição. Por exemplo, quero processar os elementos pares de um Array e criar um novo com o dobro dos valores pares:

```
[taq@~]irb
irb(main):001:0> a = [1,2,3,4,5]
=> [1, 2, 3, 4, 5]
irb(main):002:0> a2 = a.find_all {|v| v%2==0}.map{|v| v*2}
=> [4, 8]
irb(main):003:0>
```

Podemos usar **collect** também no lugar de **map**, é uma questão de gosto ali. :-)

Ordenando um array:

```
[taq@~]irb
irb(main):001:0> a = ["um","dois","tres"]
=> ["um", "dois", "tres"]
irb(main):002:0> a.sort
=> ["dois", "tres", "um"]
irb(main):003:0>
```

Podemos usar o método **sort** assim também:

```
[taq@~]irb
irb(main):001:0> a = ["um","dois","tres"]
=> ["um", "dois", "tres"]
irb(main):002:0> a.sort {|e1,e2| e1[0]<=>e2[0]}
=> ["dois", "tres", "um"]
irb(main):003:0>
```

Usando o método **sort** com um bloco, são enviados dois parâmetros para o bloco, que os compara usando o operador <=>. Podemos também utilizar o método **sort_by**, que é mais rápido e eficiente quando precisamos ordenar por alguma propriedade ou expressão do valor sendo avaliado:

```
[taq@~]irb
irb(main):001:0> a = ["um","dois","tres"]
=> ["um", "dois", "tres"]
irb(main):002:0> a.sort_by {|e| e[0]}
=> ["dois", "tres", "um"]
irb(main):003:0>
```

Nesse caso, foi avaliado a ordem alfabética das palavras. Se eu quisesse ordenar o Array pelo tamanho das palavras, poderia usar:

```
[taq@~]irb
irb(main):001:0> a = ["tres","dois","um"]
=> ["tres", "dois", "um"]
irb(main):002:0> a.sort_by {|e| e.length}
=> ["um", "dois", "tres"]
irb(main):003:0>
```

Podemos usar o método **any?** para verificar se tem algum elemento de um Array que se encaixe em uma condição determinada por um bloco:

```
[taq@~]irb
irb(main):001:0> a = [1,2,50,3]
=> [1, 2, 50, 3]
irb(main):002:0> a.any? {|v| v>10}
=> true
irb(main):003:0> a.any? {|v| v>100}
=> false
irb(main):004:0>
```

Se quisermos testar se *todos os elementos do Array satisfazem uma condição*, podemos utilizar o método **all?**:

```
[taq@~]irb
irb(main):001:0> a = [1,2,3,4,5]
=> [1, 2, 3, 4, 5]
irb(main):002:0> a.all? {|v| v>0}
=> true
irb(main):003:0> a.all? {|v| v>10}
=> false
irb(main):004:0>
```

Também podemos particionar um Array, usando o método **partition**:

```
[taq@~]irb
irb(main):001:0> a = [1,2,3,4,5,6,7,8,9,10]
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
irb(main):002:0> par,impar = a.partition {|e| e%2==0}
=> [[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]]
irb(main):003:0> par
=> [2, 4, 6, 8, 10]
irb(main):004:0> impar
=> [1, 3, 5, 7, 9]
irb(main):005:0>
```

O que aconteceu é que foram retornados dois novos Arrays, o primeiro com elementos que atendiam a condição especificada no bloco, e o segundo com os elementos que não atendiam a condição.

3.2.8 Hashes

Um Hash é basicamente um Array com índices para seus elementos (uma analogia com Java ™seriam os objetos HashMap e Hashtable). Vejamos um exemplo:

```
[taq@~/code/ruby]irb
irb(main):001:0> h = {1=>"elemento 1","um"=>"elemento um"}
=> {"um"=>"elemento um", 1=>"elemento 1"}
irb(main):002:0> h[1]
=> "elemento 1"
irb(main):003:0> h["um"]
=> "elemento um"
irb(main):004:0>
```

Podemos utilizar também **símbolos** para a chave do Hash:

```
[taq@~/code/ruby]irb
irb(main):001:0> h = {:um=>"elemento 1",:dois=>"elemento 2"}
=> {:um=>"elemento 1", :dois=>"elemento 2"}
irb(main):002:0> h[:um]
=> "elemento 1"
irb(main):003:0> h[:dois]
=> "elemento 2"
irb(main):004:0>
```

Uma pequena comparação

Primeiro, xô, flamewar. Vou fazer uma comparação com Arrays e Hashes aqui de uma maneira saudável.

Eu tive que fazer um pequeno teste com alguns Arrays em PHP, para um programa que fiz aqui, onde eu copiava o primeiro Array para mais dois, removia os elementos do primeiro usando pop, depois invertia e usava pop no segundo, e finalmente removia os elementos do terceiro usando shift.

No meu teste, Arrays de 15000 elementos foram preenchidos e tiveram seus elementos removidos com pop praticamente instantâneamente, invertendo e removendo idem, mas usando reverse/shift foram 31 segundos.

Fiz o mesmo teste com Ruby. Foi instantâneo! Resolvi aumentar o tamanho dos Array para 10 vezes mais, 150000 elementos. Levou 4 segundos!

Curioso com o teste, fui verificar a causa disso e descobri: ao contrário de Ruby (e outras linguagens também) **todos os Arrays em PHP são na verdade, Hashes!**. Isso dá praticidade, mas dá para ver o preço que se paga e o que se ganha usando uma linguagem com distinção entre os dois.

3.2.9 Símbolos

Pense em símbolos como "String levinhas". Eles representam nomes e algumas Strings dentro do interpretador do Ruby, e são sempre o mesmo objeto durante a execução de um programa.
Símbolos são criados usando **:** na frente do seu nome, como em:

```
:um
:dois
```

Como demonstrado acima.
Vamos dar uma olhada no fato de serem sempre o mesmo objeto, pegando o exemplo da documentação do Ruby:

```
module One
  class Fred
  end
  $f1 = :Fred
end
module Two
  Fred = 1
  $f2 = :Fred
end
def Fred()
end
$f3 = :Fred
$f1.id   #=> 2514190
$f2.id   #=> 2514190
$f3.id   #=> 2514190
```

Mesmo id! :-)

3.2.10 Expressões regulares

Expressões regulares são facilmente criadas, delimitando-as com /:

```
[taq@~]irb
irb(main):001:0> er = /^[0-9]/
=> /^[0-9]/
irb(main):002:0>
```

E facilmente testadas, usando o operador de correspondência =~ ou de não-correspondência **!~**:

```
[taq@~]irb
irb(main):001:0> er = /^[0-9]/
=> /^[0-9]/
irb(main):002:0> "123" =~ er
=> 0
irb(main):003:0> "123" !~ er
=> false
irb(main):004:0> "abc" =~ er
=> nil
irb(main):005:0> "abc" !~ er
=> true
irb(main):006:0>
```

Expressões regulares também podem ser criadas usando o construtor da classe ou **%r**:

```
[taq@~]irb
irb(main):001:0> er = Regexp.new("^[0-9]")
=> /^[0-9]/
irb(main):002:0> er2 = %r{^[0-9]}
=> /^[0-9]/
irb(main):003:0>
```

Vamos fazer um teste mais prático, usando a expressão regular acima, *"se a linha começar com um número"*, usando um *if* para deixar mais claro (mas tem jeitos de fazer isso mais eficientemente):

```
[taq@~]irb
irb(main):001:0> er = /^[0-9]/
=> /^[0-9]/
irb(main):002:0> if "123" =~ er
irb(main):003:1>    puts "casou!"
irb(main):004:1> else
irb(main):005:1*    puts "NÃO casou!"
irb(main):006:1> end
casou!
=> nil
irb(main):007:0> if "abc" =~ er
irb(main):008:1>    puts "casou!"
irb(main):009:1> else
irb(main):010:1*    puts "NÃO casou!"
irb(main):011:1> end
NÃO casou!
=> nil
```

Isso poderia ser escrito como:

```
[taq@~]irb
irb(main):001:0> er = /^[0-9]/
=> /^[0-9]/
irb(main):002:0> puts ("123" =~ er ? "":"não ")+"casou"
casou
=> nil
irb(main):003:0> puts ("abc" =~ er ? "":"não ")+"casou"
não casou
```

Mas se você não sabe o que é aquela construção *(condição?verdadeiro:falso)* talvez se eu mostrasse assim de primeira pudesse confundir um pouco a coisa aqui.

Podemos utilizar o método **match** para pegarmos os *tokens* de uma string que "casa" com nossa expressão regular:

```
[taq@~]irb
irb(main):001:0> er = /(..)\/(..)\/(....)/
=> /(..)\/(..)\/(....)/
irb(main):002:0> mt = er.match("20/06/2005")
=> #<MatchData:0xb7dd5dd0>
irb(main):003:0> mt.length
=> 4
irb(main):004:0> mt[0]
=> "20/06/2005"
irb(main):005:0> mt[1]
=> "20"
irb(main):006:0> mt[2]
```

```
=> "06"
irb(main):007:0> mt[3]
=> "2005"
irb(main):008:0> mt = er.match("20062005")
=> nil
irb(main):009:0>
```

No exemplo, se a string "casar" com a expressão, vai ser retornado um objeto do tipo *MatchData* , ou *nil* se não "casar". No objeto retornado podemos verificar o seu tamanho, que **inclui** a string que pesquisamos, seguida por seus *tokens* (iniciando em 1).

Podemos procurar em uma String onde uma expressão regular "casa" usando **index** :

```
[taq@~]irb
irb(main):001:0> "eustaquio".index(/taq/)
=> 3
irb(main):002:0>
```

Também podemos usar o método **gsub** da String para trocar seu conteúdo utilizando uma expressão regular:

```
[taq@~]irb
irb(main):001:0> "eustaquio".gsub(/qu/,'c')
=> "eustacio"
irb(main):002:0>
```

Troquei meu nome ali por "Eustácio" (que é o nome de um tiozinho ranzinza que vive assustando um pobre cachorrinho em um desenho animado que tem por aí). :-)

Expressões regulares também são uma mão-na-roda quando usadas com *iterators*, como por exemplo:

```
[taq@~]irb
irb(main):001:0> a = %w(um dois tres quatro cinco seis sete oito nove dez)
=> ["um", "dois", "tres", "quatro", "cinco", "seis", "sete", "oito", "nove", "dez"]
irb(main):002:0> a.grep(/s$/)
=> ["dois", "tres", "seis"]
```

Nesse caso usei o método **grep** no array, que por ter um método *each*, torna disponível os métodos adicionais do módulo *Enumerable*, entre eles, grep. No exemplo eu pedi para filtrar os elementos que eram terminados com "s".

Outro exemplo seria se tivermos um arquivo com os dados:

```
[taq@~]cat teste.txt
um
dois
tres
quatro
cinco
seis
sete
oito
nove
dez
[taq@~]irb
irb(main):001:0> File.open("teste.txt").grep(/s$/) do |r|
irb(main):002:1*    puts r
irb(main):003:1> end
dois
tres
seis
```

3.2.11 Procs

Procs são blocos de código associados à uma variável:

```
[taq@~]irb
irb(main):001:0> vezes3 = Proc.new {|n| n*3}
=> #<Proc:0xb7de21fc@(irb):1>
irb(main):002:0> vezes3.call(1)
=> 3
irb(main):003:0> vezes3.call(2)
=> 6
irb(main):004:0> vezes3.call(3)
=> 9
irb(main):005:0>
```

Podemos transformar um bloco de código em uma Proc usando **lambda**:

```
[taq@~]irb
irb(main):001:0> vezes3 = lambda {|n| n*3}
=> #<Proc:0xb7dd96c0@(irb):1>
irb(main):002:0> vezes3.call(1)
=> 3
irb(main):003:0> vezes3.call(2)
=> 6
irb(main):004:0> vezes3.call(3)
=> 9
irb(main):005:0>
```

3.2.12 Métodos

Métodos também são muito fáceis de serem declarados. São *def*inidos usando **def**, terminando com **end** (mais sobre isso na próxima seção):

```
[taq@~]irb
irb(main):001:0> def diga_oi
irb(main):002:1>    puts "oi!"
irb(main):003:1> end
=> nil
irb(main):004:0> diga_oi
oi!
=> nil
irb(main):005:0>
```

Retornando valores

Olhem que legal:

```
[taq@~]irb
irb(main):001:0> def vezes(n1,n2)
irb(main):002:1>    n1*n2
irb(main):003:1> end
=> nil
irb(main):004:0> vezes(3,4)
=> 12
irb(main):005:0>
```

Ué! Cadê o return ali? Não precisa. O valor do retorno do método nesse caso é a última expressão avaliada, no caso, 3 vezes 4.

Também podemos retornar mais de um resultado (na verdade, *apenas um objeto*, complexo ou não, é retornado, mas dá-se a impressão que são vários):

```
[taq@~]irb
irb(main):001:0> def cinco_multiplos(n)
irb(main):002:1>    (1..5).map {|v| n*v}
irb(main):003:1> end
=> nil
irb(main):004:0> n1, n2, n3, n4, n5 = cinco_multiplos(5)
=> [5, 10, 15, 20, 25]
irb(main):005:0> puts "múltiplos de 5: #{n1},#{n2},#{n3},#{n4},#{n5}"
múltiplos de 5: 5,10,15,20,25
=> nil
irb(main):006:0>
```

Recebendo parâmetros

Métodos também recebem parâmetros (que novidade ehehe) e retornam valores:

```
[taq@~]irb
irb(main):001:0> def vezes(n1,n2)
irb(main):002:1>    return n1*n2
irb(main):003:1> end
=> nil
irb(main):004:0> vezes(3,4)
=> 12
irb(main):005:0>
```

Métodos também podem receber, como no PHP, **argumentos com valores default**, por exemplo:

```
[taq@~]irb
irb(main):001:0> def hello(s="Desconhecido")
irb(main):002:1>    puts "Oi, #{s}!"
irb(main):003:1> end
=> nil
irb(main):004:0> hello("TaQ")
Oi, TaQ!
=> nil
irb(main):005:0> hello
Oi, Desconhecido!
=> nil
irb(main):006:0>
```

Então, se não passarmos um argumento, ele assume o que foi especificado como default.

Também podemos ter parâmetros variáveis, por exemplo:

```
[taq@~]irb
irb(main):001:0> def vargars(*args)
irb(main):002:1>    args.each {|v| puts "parametro recebido:#{v}"}
irb(main):003:1> end
=> nil
irb(main):004:0> vargars(1,2)
parametro recebido:1
parametro recebido:2
=> [1, 2]
irb(main):005:0> vargars(1,2,3,4,5)
parametro recebido:1
parametro recebido:2
parametro recebido:3
parametro recebido:4
```

```
parametro recebido:5
=> [1, 2, 3, 4, 5]
irb(main):006:0>
```

Os parâmetros variáveis vem na forma de um **Array**, e para indicar no método esse tipo de parâmetro, é só colocar um asterisco (*) na frente do nome do parâmetro.

Podemos passar um bloco para o método, executando-o com **yield**:

```
[taq@~]irb
irb(main):001:0> def teste(n)
irb(main):002:1>    yield(n)
irb(main):003:1> end
=> nil
irb(main):004:0> teste(2) {|i| i*i}
=> 4
irb(main):005:0> teste(3) {|i| i*i}
=> 9
irb(main):006:0> teste(4) {|i| i*i}
=> 16
irb(main):007:0>
```

Para testar se o método recebeu um bloco, podemos usar **block_given?**:

```
[taq@~]irb
dirb(main):001:0> def teste(n)
irb(main):002:1>    if block_given?
irb(main):003:2>       yield(n)
irb(main):004:2>    else
irb(main):005:2*       puts "não foi passado um bloco!"
irb(main):006:2>    end
irb(main):007:1> end
=> nil
irb(main):008:0> teste(2) {|i| i*i}
=> 4
irb(main):009:0> teste(3) {|i| i*i}
=> 9
irb(main):010:0> teste(4) {|i| i*i}
=> 16
irb(main):011:0> teste(5)
não foi passado um bloco!
=> nil
irb(main):012:0>
```

E se passarmos *o último argumento de um método com um* **&** *na frente*, um bloco passado como parâmetro será convertido em uma Proc e associado ao parâmetro:

```
[taq@~]irb
irb(main):001:0> def teste(n,&b)
irb(main):002:1>    puts b.call(n)
irb(main):003:1> end
=> nil
irb(main):004:0> teste(2) {|i| i*i}
4
=> nil
irb(main):005:0> teste(3) {|i| i*i}
9
=> nil
irb(main):006:0> teste(4) {|i| i*i}
```

```
16
=> nil
irb(main):007:0>
```

Para converter uma Proc em um bloco de código em um parâmetro do método, você pode usar o caracter **&** *antes do nome da Proc*:

```
[taq@~]irb
irb(main):001:0> quadrado = Proc.new {|i| i*i}
=> #<Proc:0xb7de2080@(irb):1>
irb(main):002:0> def teste(n)
irb(main):003:1>    puts yield(n)
irb(main):004:1> end
=> nil
irb(main):005:0> teste(3,&quadrado)
9
=> nil
irb(main):006:0>
```

Como os parâmetros são enviados

Os parâmetros são passados **por cópia da referência do objeto**, então, se alterarmos um objeto que foi passado como parâmetro, dentro de um método, e **ele puder ser mutável**, ele refletirá fora do método. Por exemplo:

```
[taq@~]irb
irb(main):001:0> s = "Oi, mundo!"
=> "Oi, mundo!"
irb(main):002:0> s.object_id
=> -605086718
irb(main):003:0> def maiusculas(s)
irb(main):004:1>    s.upcase!
irb(main):005:1> end
=> nil
irb(main):006:0> maiusculas(s)
=> "OI, MUNDO!"
irb(main):007:0> s
=> "OI, MUNDO!"
irb(main):008:0> s.object_id
=> -605086718
irb(main):009:0>
```

Foi criada uma String (s), e logo depois verifiquei seu id. Dentro do método alterei o parâmetro usando o método **upcase!** verifiquei seu valor (que foi alterado para maiuscúlas) e seu id, que continua o mesmo, então, o objeto foi alterado dentro do método, *usando como referência o objeto que ele apontava, que pode ser mutável.*

> **ATENÇÃO!**
> Métodos que alteram o valor e estado de um objeto geralmente tem um outro método de mesmo nome, porém com um **!** no final. Esses são os chamados **métodos destrutivos**. A versão sem **!** pega uma cópia da referência, faz a mudança, e retorna um novo objeto. Os métodos destrutivos alteram o próprio objeto.

Se **não** usássemos o **!**, somente retornando o valor convertido, teríamos retornado *outro objeto, criado dentro do método, no seu retorno* (pelo fato de *s.upcase* ser a última expressão avaliada no método). Se usássemos o !, o objeto retornado seria o mesmo que foi alterado.

No exemplo abaixo, o parâmetro não foi alterado, mas foi retornado um novo objeto *na chamada do método, atribuído à referência anterior, s.* Isso pode ser comprovado verificando seu id:

```
[taq@~]irb
irb(main):001:0> s = "Oi, mundo!"
=> "Oi, mundo!"
irb(main):002:0> s.object_id
=> -605086718
irb(main):003:0> def maiusculas(s)
irb(main):004:1>    s.upcase
irb(main):005:1> end
=> nil
irb(main):006:0> maiusculas(s)
=> "OI, MUNDO!"
irb(main):007:0> s
=> "Oi, mundo!"
irb(main):008:0> s.object_id
=> -605086718
irb(main):009:0> s = maiusculas(s)
=> "OI, MUNDO!"
irb(main):010:0> s
=> "OI, MUNDO!"
irb(main):011:0> s.object_id
=> -605122958
irb(main):012:0>
```

Para alterar o valor de variáveis do tipo **Fixnum** (Fixnum's são imutáveis), precisaríamos fazer do segundo modo, retornando um novo objeto para a sua variável apontando para o seu Fixnum, pois eles não podem ser alterados.

Deixa eu tentar deixar mais visível aqui:

```
def teste1(s)
   print "Rodando TESTE1\n"
   print "valor #{s}, id #{s.object_id}\n"
   # Aqui ele cria um novo objeto com o valor da referencia em maiusculas
   # e aponta a referencia de s (ATENÇÃO! ele troca a referência de s,
   # definida localmente aqui, para a localização do novo objeto, e não
   # interfere com a variável externa do método que gerou s como
   # parâmetro!
   s = s.upcase
   print "valor #{s}, id #{s.object_id} (aqui tem que ter um id diferente)\n\n"
# Aqui ele retorna s, com sua nova referência, apontando para o novo objeto
s
end

def teste2(s)
   print "Rodando TESTE2\n"
   print "valor #{s}, id #{s.object_id}\n"
# Método destrutivo à frente! Vai agir direto no objeto
# que se encontra na referência apontada por s
   s.upcase!
   print "valor #{s}, id #{s.object_id}\n\n"
   s
end

s1 = "teste" # s1 aponta para um objeto do tipo String, com conteúdo "teste"
s2 = s1 # s2 aponta para s1
sd = s1.dup # duplica para outro
```

```
print "Vamos verificar os objetos:\n"
print "s1 id:#{s1.object_id}, vale #{s1}\n"
print "s2 id:#{s2.object_id}, vale #{s2}\n"
print "sd id:#{sd.object_id}, vale #{sd} (objeto duplicado, id diferente!)\n\n"

s3 = teste1(s1)
print "Vamos verificar os objetos:\n"
print "s1 id:#{s1.object_id}, vale #{s1}\n"
print "s2 id:#{s2.object_id}, vale #{s2}\n"
print "s3 id:#{s3.object_id}, vale #{s3} (aqui tem que ter um id diferente)\n"
print "sd id:#{sd.object_id}, vale #{sd} (objeto duplicado, id diferente!)\n\n"

s4 = teste2(s1)
print "Vamos verificar os objetos:\n"
print "s1 id:#{s1.object_id}, vale #{s1}\n"
print "s2 id:#{s2.object_id}, vale #{s2}\n"
print "s3 id:#{s3.object_id}, vale #{s3}\n"
print "s4 id:#{s4.object_id}, vale #{s4}\n"
print "sd id:#{sd.object_id}, vale #{sd} (objecto duplicado, id diferente, valor original!)\n"
```

Resultado:

```
Vamos verificar os objetos:
s1 id:-604854218, vale teste
s2 id:-604854218, vale teste
sd id:-604854228, vale teste (objeto duplicado, id diferente!)

Rodando TESTE1
valor teste, id -604854218
valor TESTE, id -604854348 (aqui tem que ter um id diferente)

Vamos verificar os objetos:
s1 id:-604854218, vale teste
s2 id:-604854218, vale teste
s3 id:-604854348, vale TESTE (aqui tem que ter um id diferente)
sd id:-604854228, vale teste (objeto duplicado, id diferente!)

Rodando TESTE2
valor teste, id -604854218
valor TESTE, id -604854218

Vamos verificar os objetos:
s1 id:-604854218, vale TESTE
s2 id:-604854218, vale TESTE
s3 id:-604854548, vale TESTE
s4 id:-604854218, vale TESTE
sd id:-604854228, vale teste (objecto duplicado, id diferente, valor original!)
```

Tentando expressar isso graficamente:

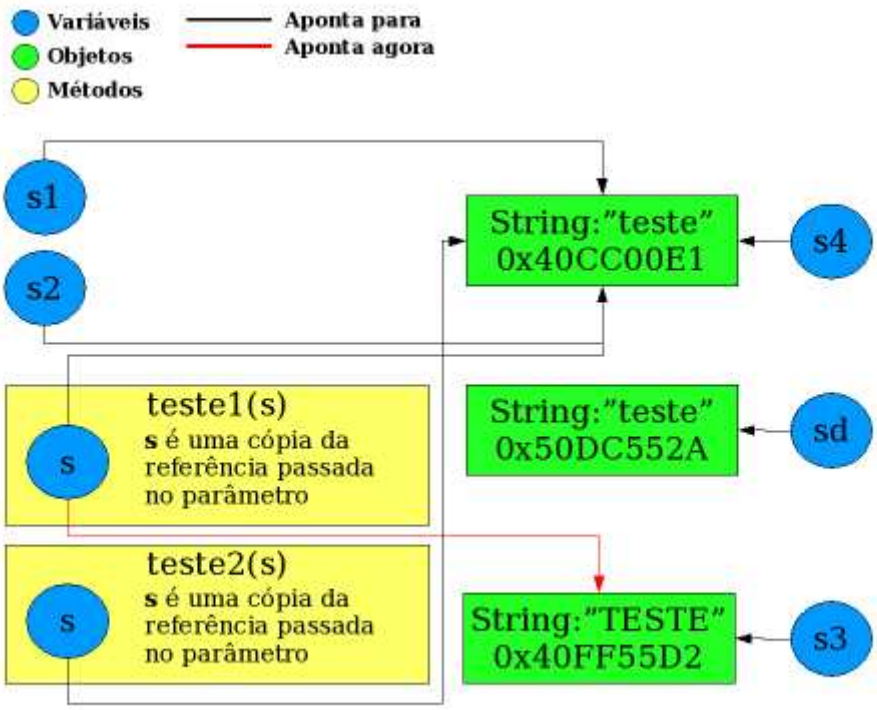

Figura 3.1: Representação de referências aos objetos

Vamos analisar detalhadamente o que ocorre aqui.

- Criei uma variável, s1, que aponta para um objeto tipo String (variáveis não são objetos, lembre-se!). Esse objeto foi criado e armazenado em uma determinada área da memória (vejam para onde s1 aponta).
- Criei uma segunda variável, s2, apontando para o mesmo lugar (mesmo objeto) que s1.
- Criei outra variável, sd, *duplicando s1*, isso é, *criando um novo objeto do tipo de s1 (String), usando como ponto de partida o valor referenciado por s1.*
- Verifiquei os ids das variáveis. As variáveis s1 e s2 tem ids idênticos, enquanto sd tem um diferente, tudo certo, pois sd aponta para outro objeto.
- Criei uma nova variável s3 com o resultado do método **teste1**, que recebe **uma cópia da referência do objeto**, como uma variável local no escopo do método corrente, executa uma operação *criando um novo objeto, cuja referência é retornada pelo método* e armazenada em s3. Vejam no gráfico o objeto para onde s3 aponta.
- Dentro de teste1 isso fica explícito na primeira chamada, onde o id do objeto **é o mesmo de s1**, e quando retornado (poderia ter retornado direto, sem a variável s, que ia dar na mesma, fiz assim só para ficar mais claro) temos o id do novo objeto.
- Criei uma nova variável s4 com o resultado do método **teste2**, que recebe uma cópia da referência do objeto, como uma variável local no escopo do método corrente, **e altera diretamente o objeto referenciado pela variável, convertendo-o para maiúsculas, nesse caso**. Reparem em teste2 como os valores estão diferentes, mas os ids são os mesmos.
- A variável s4, então, aponta para o mesmo objeto originalmente referenciado por s1 e s2, enquanto s3 continua sossegada apontando para o objeto criado por teste1, e sd, que foi duplicada no começo, continua a mesma também.

Podemos pensar em variáveis como umas caixinhas com vários bilhetinhos dentro, com as coordenadas de uma cidade. Quando você passa a variável como parâmetro, vamos imaginar que você está despejando na caixa vazia do seu amigo um dos bilhetinhos de uma de suas caixas.
O bilhetinho fica no espaço físico (escopo) da caixa do seu amigo, e pode ser utilizado para dois motivos:

- Apontar para outra cidade (ou até para uma *panela*, se você quiser, levando em conta que isso alterará o tipo original da referência que você pegar de volta da caixinha do seu amigo, e você pode ficar surpreso de não encontrar uma cidade lá). O número está escrito à lápis, pegue uma borracha, apague e escreva um novo número lá, **desde que o lugar novo exista, mesmo que tenha que ser criado ali na hora**.

- Através de um telefone no verso do bilhetinho, ligar lá na cidade e dizer "ei, pintem as ruas aí de verde-limão", se isso for possível naquela cidade (se ela tiver um *método* para isso). Nesse caso, o tipo apontado pelo bilhetinho é mantido, ou seja, continua a ser uma cidade no mesmo lugar, mas alguma característica foi alterada no local (as ruas estão pintadas de verde-limão arrrrgh!).

Para exemplificar e complicar um pouco mais, vamos ver esses exemplos:

```
[taq@~/code/ruby]irb
irb(main):001:0> i1 = 1
=> 1
irb(main):002:0> i2 = 1
=> 1
irb(main):003:0> i3 = 1
=> 1
irb(main):004:0> i4 = 2
=> 2
irb(main):005:0> i5 = 2
=> 2
irb(main):006:0> i1.object_id
=> 3
irb(main):007:0> i2.object_id
=> 3
irb(main):008:0> i3.object_id
=> 3
irb(main):009:0> i4.object_id
=> 5
irb(main):010:0> i5.object_id
=> 5
irb(main):011:0>
```

Olhem que legal! As variáveis dos Fixnum's apontam **sempre** para o mesmo objeto (verificando isso usando o object_id) do tipo Fixnum que foi atribuído à elas. Isso por que Fixnum são "immediate values", que armazenam o valor, e **não a referência do objeto**, assim como *true, false, nil e símbolos.*

Não se esqueçam que objetos do tipo Fixnum são imutáveis, e são immediate values (estão lá e ninguém muda!).

Fazendo uma comparação com nosso exemplo de Strings acima, se você criasse outra variável valendo "teste", além das que estavam lá, seria outra String com valor "teste", ao contrário dos Fixnum's que são **sempre** o mesmo objeto:

```
[taq@~/latex/ruby]irb
irb(main):001:0> s1 = "teste"
=> "teste"
irb(main):002:0> s2 = s1
=> "teste"
irb(main):003:0> sd = s1.dup
```

```
=> "teste"
irb(main):004:0> sn = "teste"
=> "teste"
irb(main):005:0> n1 = 1
=> 1
irb(main):006:0> n2 = 1
=> 1
irb(main):007:0> n3 = 2
=> 2
irb(main):008:0> n4 = 3
=> 3
irb(main):009:0> print "s1:#{s1.object_id}, s2:#{s2.object_id}, sd:#{sd.object_id},
sn:#{sn.object_id}, n1:#{n1.object_id}, n2:#{n2.object_id}, n3:#{n3.object_id},
n4:#{n4.object_id}"
s1:-605084180, s2:-605084180, sd:-605092590, sn:-605096770, n1:3, n2:3, n3:5, n4:7=> nil
irb(main):010:0>
```

Fixnum's representam números com **capacidade** até 31 bits, indo de -(1<<30) até (1<<30)-1 (reparem que estão contidos números *com sinal*, negativos e positivos).

Dando uma olhadinha no object_id dos Fixnum's, vemos que ele é bem interessante. O último bit do object_id é usado para indicar se a referência é de um Fixnum, e a verificação disso é feita bem rapidamente, pois Fixnum's são os únicos objetos que tem o LSB (least significant bit) do seu object_id ativo, ou seja, sendo 1, então basta fazer um *and* lógico com 1 no object_id:

```
[taq@~]irb
irb(main):001:0> puts "#{123.class}=>#{123.object_id & 0x1}"
Fixnum=>1
=> nil
irb(main):002:0> puts "#{123456789.class}=>#{123456789.object_id & 0x1}"
Fixnum=>1
=> nil
irb(main):003:0> puts "#{1234567890.class}=>#{1234567890.object_id & 0x1}"
Bignum=>0
=> nil
irb(main):004:0> puts "#{123.45.class}=>#{123.45.object_id & 0x1}"
Float=>0
=> nil
irb(main):005:0> puts "#{"oi".class}=>#{"oi".object_id & 0x1}"
String=>0
=> nil
irb(main):006:0> puts "#{true.class}=>#{true.object_id & 0x1}"
TrueClass=>0
=> nil
```

Para mostrar como os ids dos Fixnum's são calculados:

```
[taq@~]irb
irb(main):001:0> 2.object_id
=> 5
irb(main):002:0> (2 << 1) | 0x1
=> 5
irb(main):003:0> 3.object_id
=> 7
irb(main):004:0> (3 << 1) | 0x1
=> 7
irb(main):005:0>
```

Desloquei um bit à esquerda no número, e substitui o LSB com 1, usando um *or* lógico (como disse, os Fixnum's tem que ter esse último bit ativo!) e comparei com o object_id de n. No fim das contas, deslocando o bit à esquerda, eu *dupliquei* o número, e o *or* lógico *somou 1* ao mesmo. Lembram-se do limite de 30 bits? Deslocando (e duplicando) um número de 30 bits à esquerda e fazendo um *or* lógico com 1, tenho um de 31 bits com todos os bits ativos, olha o limite da capacidade dos Fixnum's aí.
Fazendo um deslocamento inverso, acontece isso:

```
[taq@~]irb
irb(main):001:0> 2.object_id
=> 5
irb(main):002:0> 5 >> 1
=> 2
irb(main):003:0> 3.object_id
=> 7
irb(main):004:0> 7 >> 1
=> 3
irb(main):005:0>
```

Ou seja, além de identificar se um objeto é um Fixnum, o object_id armazena efetivamente o valor do Fixnum (nem precisei subtrair 1 por que o último bit foi perdido mesmo)!

Isso trás **muita** velocidade em cálculos com esse tipo de objeto, pois o valor do mesmo se encontra na própria referência, não é necessário acesso à um espaço de memória alocado (ele não existe, nesse caso) ou acesso à métodos para fazer a aritmética. Os métodos, quando são requisitados, são consultados na instñcia (única, como qualquer outra classe do Ruby) da classe Fixnum.

Métodos destrutivos

Como explicado acima, são os métodos que tem um **!** no fim do seu nome, e que alteram o conteúdo do próprio objeto, na posição de memória que ele se encontra.

Métodos predicados

São métodos que terminam com **?**, e retornam **true** ou **false**. São utilizados para testar uma condição. A analogia seriam os métodos **is_condição**, que usamos em Java:

```
[taq@~]irb
irb(main):001:0> a = ["um","dois","tres"]
=> ["um", "dois", "tres"]
irb(main):002:0> a.include? "dois"
=> true
irb(main):003:0>
```

Métodos ou funções

Você pode falar "puxa, esse lance de métodos é legal, mas você não tem funções tipo as de C?". Temos e não temos.

Ruby **é** uma linguagem puramente orientada à objetos, então **tudo** é baseado em objetos, e "funções" que chamamos tipo **puts** são na verdade métodos de classes derivadas de **Object**, no caso de puts, mixadas através do módulo **Kernel**. Olhem que legal:

```
[taq@~/code/ruby]irb
irb(main):001:0> puts "Oi!"
Oi!
=> nil
irb(main):002:0> Kernel::puts "Oi!"
Oi!
=> nil
```

```
irb(main):003:0> def funcao_maluca
irb(main):004:1>    puts "Oi!oi!oi!oi!oi!"
irb(main):005:1> end
=> nil
irb(main):006:0> funcao_maluca
Oi!oi!oi!oi!oi!
=> nil
irb(main):007:0> Kernel::funcao_maluca
Oi!oi!oi!oi!oi!
=> nil
irb(main):008:0> Object::funcao_maluca
Oi!oi!oi!oi!oi!
=> nil
irb(main):009:0> class Teste
irb(main):010:1> end
=> nil
irb(main):011:0> Teste::funcao_maluca
Oi!oi!oi!oi!oi!
=> nil
irb(main):012:0>
```

As duas primeiras chamadas são similares. Depois criei uma "função maluca", que ficou acessível através do nosso contexto, através do Kernel, através de Object e através da classe Teste que criamos ali!!! Interessante, não?

Experimentem digitar no irb **self.class** e vejam o retorno ... depois experimentem ver os métodos da classe que foi retornada (como indicado no começo do tutorial). ;-)

3.2.13 Operadores

Em Ruby temos vários operadores que são métodos e que podem ser redefinidos nas suas classes.
Vamos dar uma olhada neles, da ordem mais alta para a baixa, os que estão indicados como métodos podem ser redefinidos:

Método	**Operador**	**Descrição**
Sim	**	Exponenciação
Sim	-, +, !,	unários menos e mais, não (not) e complementa
Sim	*, /, %	vezes, dividido, módulo
Sim	+, -	mais, menos
Sim	<<, >>	deslocamento para esquerda e direita
Sim	&	"E" (and) binário
Sim	— ^	"Ou" (or) binário, regular e exclusivo
Sim	>, >=, <, <=	maior, maior igual, menor, menor igual
Sim	<=>, ==, ===, ! =, = ,!	Operadores de igualdade e de reconhecimento de padrões
Não	&&	"E" (and) lógico
Não	——	"Ou" (or) lógico
Não	.., ...	Ranges (inclusiva e exclusiva)
Não	? :	If-then-else ternário
Não	=, %=, /=, -=, +=, |=, &=, >>=, <<=, *=, &&=, ||=, **=	Atribuição
Não	not	Negação lógica
Não	and or	Composição lógica

Espera aí! Onde estão o ++ e , que tem em C e em PHP? Bom, em Ruby eles não existem. Utilize **variável += 1** e **variável -= 1**.

Vamos ver como redefinir operadores quando estudarmos Classes.

3.2.14 Juntando tudo

Vamos testar o escopo das variáveis, dentro de um método:

```
[taq@~]irb
irb(main):001:0> $i = 1
=> 1
irb(main):002:0> i = 2
=> 2
irb(main):003:0> def teste
irb(main):004:1>    puts $i
irb(main):005:1>    puts i
irb(main):006:1> end
=> nil
irb(main):007:0> teste
1
NameError: undefined local variable or method `i' for main:Object
  from (irb):5:in `teste'
  from (irb):7
irb(main):008:0>
```

Ele imprimiu a primeira variável ($i) mas não a segunda (i), por que $i é pública e o método consegue enxergar ela "lá fora", mas i é privada "lá fora" e o método não consegue enxergá-la lá dentro.

"Mas que rolo, como vou saber se é mesmo a variável pública impressa ali?" você pode perguntar. "Quero colocar uma descrição antes". Bom, se você fizer isso:

```
[taq@~]irb
irb(main):001:0> $i = 1
=> 1
irb(main):002:0> i = 2
=> 2
irb(main):003:0> def teste
irb(main):004:1>    puts "variavel publica:"+$i
irb(main):005:1>    puts "variavel privada:"+i
irb(main):006:1> end
=> nil
irb(main):007:0> teste
TypeError: cannot convert Fixnum into String
  from (irb):4:in `+'
  from (irb):4:in `teste'
  from (irb):7
irb(main):008:0>
```

Vai dar pau (*deu* pau!). Por que? Você tentou "grudunhar" uma String e uma Fixnum. Ruby não aceita (lembra da tipagem forte?).
E como fazer isso então? Lembra que eu pedi para você alterar (ou criar) o arquivo */.irbrc* lá no começo? Se você fez isso, quando você digitar

```
[taq@~]irb
irb(main):001:0> $i = 1
=> 1
irb(main):002:0> $i.
```

e apertar a tecla TAB depois ali do ponto (talvez você tenha que apertá-la duas vezes), o irb vai te retonar

```
                          $i.modulo
$i.__id__                 $i.next
```

```
$i.__send__                $i.nil?
$i.abs                     $i.nonzero?
$i.between?                $i.object_id
$i.ceil                    $i.prec
$i.chr                     $i.prec_f
$i.class                   $i.prec_i
$i.clone                   $i.private_methods
$i.coerce                  $i.protected_methods
$i.display                 $i.public_methods
$i.div                     $i.quo
...
```

que são os **métodos que o objeto armazenado em $i tem**. Legal não? Toda a documentação ao seu alcance ali (lembre-se que você tem que ter rodado o rdoc do jeito que mencionei no começo do tutorial para isso acontecer também).

Bom, de volta ao problema. Se você der uma olhada nos métodos, vai achar um chamado **to_s**, que nada mais faz do que converter o objeto em uma String. Assim sendo:

```
[taq@~]irb
irb(main):001:0> $i = 1
=> 1
irb(main):002:0> i = 2
=> 2
irb(main):003:0> def teste
irb(main):004:1>    puts "variavel publica:"+$i.to_s
irb(main):005:1>    puts "variavel privada:"+i.to_s
irb(main):006:1> end
=> nil
irb(main):007:0> teste
variavel publica:1
NameError: undefined local variable or method `i' for main:Object
  from (irb):5:in `teste'
  from (irb):7
irb(main):008:0>
```

Mostrou o valor da sua variável pública, mas continou corretamente dando pau na privada (frase estranha essa). Esse tipo de conversão funciona com todos os objetos que tem um método de conversão para String:

```
[taq@~]irb
irb(main):001:0> v = "valor:"+1.23
TypeError: cannot convert Float into String
  from (irb):1:in `+'
  from (irb):1
irb(main):002:0> v = "valor:"+1.23.to_s
=> "valor:1.23"
irb(main):003:0>
```

Já que estamos fuçando nos métodos dos objetos, vamos juntar Ranges e blocos tudo no meio. Se você declarar um objeto do tipo Range, verá que ele tem um método chamado **each**, que significa *para cada valor da range, execute o bloco tal*. Vamos ver como isso funciona:

```
[taq@~]irb
irb(main):001:0> r = 0...3
=> 0...3
irb(main):002:0> r.each {puts "oi"}
oi
oi
oi
```

```
=> 0...3
irb(main):003:0>
```

Foi impresso o conteúdo do bloco três vezes (3 pontinhos, sem o valor final, lembra?). Que tal passar o valor que está sendo processado para o bloco?

```
irb(main):001:0> r = 0...3
=> 0...3
irb(main):002:0> r.each {|v| puts "oi,"+v.to_s}
oi,0
oi,1
oi,2
=> 0...3
irb(main):003:0>
```

O parâmetro foi passado para o bloco usando —**v**—, sendo que **v** é o valor sendo processado. Note-se que eu usei **to_s** ali também para fazer a conversão do valor em String.

Vamos testar agora a passagem de vários parametros, usando um Hash:

```
[taq@~]irb
irb(main):001:0> h = {1=>"um",2=>"dois",3=>"tres"}
=> {1=>"um", 2=>"dois", 3=>"tres"}
irb(main):002:0> h.each {|k,v| puts "key:"+k.to_s+" val:"+v.to_s}
key:1 val:um
key:2 val:dois
key:3 val:tres
=> {1=>"um", 2=>"dois", 3=>"tres"}
irb(main):003:0>
```

Dois parâmetros passados usando —k,v—.
Para Arrays é a mesma coisa:

```
[taq@~]irb
irb(main):001:0> a = ['primeiro','segundo','terceiro']
=> ["primeiro", "segundo", "terceiro"]
irb(main):002:0> a.each {|v| puts "estou no "+v.to_s}
estou no primeiro
estou no segundo
estou no terceiro
=> ["primeiro", "segundo", "terceiro"]
irb(main):003:0>
```

Coisa interessante: esses tipos de blocos podem ser escritos utilizando **do / end**, dessa maneira:

```
irb(main):001:0> a = ['primeiro','segundo','terceiro']
=> ["primeiro", "segundo", "terceiro"]
irb(main):002:0> a.each do |v|
irb(main):003:1*    puts "estou no "+v.to_s
irb(main):004:1> end
estou no primeiro
estou no segundo
estou no terceiro
=> ["primeiro", "segundo", "terceiro"]
irb(main):005:0>
```

Questão de gosto e de como deixar a coisa mais legível.
Outra coisa interessante: você pode concatenar métodos. Vamos ver o que acontece se eu utilizar o método **sort** ANTES de **each**:

```
[taq@~]irb
irb(main):001:0> a = ['zabumba','triangulo','pandeiro']
=> ["zabumba", "triangulo", "pandeiro"]
irb(main):002:0> a.sort.each {|v| puts "ordenado:"+v.to_s}
ordenado:pandeiro
ordenado:triangulo
ordenado:zabumba
=> ["pandeiro", "triangulo", "zabumba"]
irb(main):003:0>
```

Mais uma coisa interessante: apesar do escopo, os blocos podem ver (e alterar) variáveis "de fora" deles se você a passar como parâmetros:

```
[taq@~]irb
irb(main):001:0> s = "nada"
=> "nada"
irb(main):002:0> a = ['alguma','coisa','aqui']
=> ["alguma", "coisa", "aqui"]
irb(main):003:0> a.each {|s| puts s}
alguma
coisa
aqui
=> ["alguma", "coisa", "aqui"]
irb(main):004:0> puts s
aqui
=> nil
irb(main):005:0>
```

Viu o que aconteceu? O parâmetro do bloco é **s**, que é uma variável já existente. Quando você chama o each, ele usa a variável como receptora do parâmetro e altera o valor dela dentro do bloco. Então quando você verifica o valor de s no final, ela vale o último valor processado pelo bloco, que é "aqui". Digamos que a variável fica "permeável".

Também temos uma keyword para executar o bloco, que é **yield**. Vamos ver:

```
[taq@~]irb
irb(main):001:0> def teste
irb(main):002:1>    yield
irb(main):003:1>    yield
irb(main):004:1> end
=> nil
irb(main):005:0> teste {puts "oi"}
oi
oi
=> nil
irb(main):006:0>
```

O método teste executou o bloco que lhe foi passado duas vezes, pois temos dois yields lá.
Podemos também passar valores para o bloco, por exemplo:

```
[taq@~]irb
irb(main):001:0> def repete(n)
irb(main):002:1>    while n > 0
irb(main):003:2>       yield n
irb(main):004:2>       n -= 1
irb(main):005:2>    end
irb(main):006:1> end
=> nil
irb(main):007:0> repete(3) {|v| puts "numero:"+v.to_s}
```

```
numero:3
numero:2
numero:1
=> nil
irb(main):008:0>
```

Ops! O que é aquele while e -= ali? Como se vocês não soubessem, mas vamos examiná-los na próxima seção.

3.3 Estruturas de controle

3.3.1 Condicionais

if

Vamos dar uma olhada no nosso velho amigo **if**.

O interessante é que em Ruby, o if por exemplo não termina com endif, ou é limitado por chaves, ou está delimitado por tabulações. Termina em **end**. Tudo termina em end! Afinal, *é* o fim, não é mesmo? :-)

```
[taq@~]irb
irb(main):001:0> i = 10
=> 10
irb(main):002:0> if i > 10
irb(main):003:1>    puts "maior que 10"
irb(main):004:1> elsif i == 10
irb(main):005:1>    puts "igual a 10"
irb(main):006:1> else
irb(main):007:1*    puts "menor que 10"
irb(main):008:1> end
igual a 10
=> nil
irb(main):009:0>
```

A forma é **if . . . elsif . . . else . . . end**. Prestem atenção no **elsif**!

unless

Você pode usá-lo como uma forma negativa do if também, "se não":

```
[taq@~]irb
irb(main):001:0> i = 10
=> 10
irb(main):002:0> unless i >= 10 # se i NÃO FOR maior ou igual a 10
irb(main):003:1>    puts "menor que 10"
irb(main):004:1> else
irb(main):005:1*    puts "maior igual que 10"
irb(main):006:1> end
maior igual que 10
=> nil
irb(main):007:0> puts "menor que 11" unless i > 11 # se i NÃO FOR maior que 11, imprima!
menor que 11
=> nil
irb(main):008:0>
```

Você pode utilizá-lo também em uma linha, como mostrado acima.

case

Podemos utilizar o case assim:

```
[taq@~]irb
irb(main):001:0> i = 10
=> 10
irb(main):002:0> case i
irb(main):003:1>    when 0..5
irb(main):004:1>       puts "entre 0 e 5"
irb(main):005:1>    when 6..10
irb(main):006:1>       puts "entre 6 e 10"
irb(main):007:1>    else
irb(main):008:1*       puts "???"
irb(main):009:1> end
entre 6 e 10
=> nil
irb(main):010:0>
```

Reparem que foram usados Ranges nas comparações.
A forma é **case ... when ... else ... end**.
Uma coisa legal é que dá para capturar o valor retornado (assim como em outras estruturas aqui mostradas) em uma variável, por exemplo:

```
[taq@~]irb
irb(main):001:0> i = 10
=> 10
irb(main):002:0> n =
irb(main):003:0* case i
irb(main):004:1>    when 0..5
irb(main):005:1>       "entre 0 e 5"
irb(main):006:1>    when 6..10
irb(main):007:1>       "entre 6 e 10"
irb(main):008:1>    else
irb(main):009:1*       "???"
irb(main):010:1> end
=> "entre 6 e 10"
irb(main):011:0> n
=> "entre 6 e 10"
irb(main):012:0>
```

3.3.2 Loops

Importante notar que há quatro maneiras de interagir com o conteúdo do loop, por dentro:

- **break** sai do loop completamente
- **next** vai para a próxima iteração
- **return** sai do loop e do método onde o loop está contido
- **redo** restarta o loop

while

Simples:

```
[taq@~]irb
irb(main):001:0> i = 0
=> 0
```

```
irb(main):002:0> while i < 5
irb(main):003:1>    puts i
irb(main):004:1>    i += 1
irb(main):005:1> end
0
1
2
3
4
=> nil
irb(main):006:0>
```

for

O for em Ruby utiliza coleções para fazer seu loop, por exemplo com uma Range:

```
[taq@~]irb
irb(main):001:0> for i in (0..5)
irb(main):002:1>    puts i
irb(main):003:1> end
0
1
2
3
4
5
=> 0..5
irb(main):004:0>
```

Com um Array:

```
[taq@~]irb
irb(main):001:0> a = ['um','dois','tres']
=> ["um", "dois", "tres"]
irb(main):002:0> for s in a
irb(main):003:1>    puts s
irb(main):004:1> end
um
dois
tres
=> ["um", "dois", "tres"]
irb(main):005:0>
```

Com uma faixa de caracteres:

```
[taq@~]irb
irb(main):001:0> for c in ('a'..'e')
irb(main):002:1>    puts c
irb(main):003:1> end
a
b
c
d
e
=> "a".."e"
irb(main):004:0>
```

until

"Faça até que":

```
[taq@~]irb
irb(main):001:0> i = 0
=> 0
irb(main):002:0> until i == 5
irb(main):003:1>    puts i
irb(main):004:1>    i += 1
irb(main):005:1> end
0
1
2
3
4
=> nil
irb(main):006:0>
```

begin

Podemos combinar o **begin** com **while** e **until**, no seu final:

```
[taq@~]irb
irb(main):001:0> i = 0
=> 0
irb(main):002:0> begin
irb(main):003:1*    puts i
irb(main):004:1>    i += 1
irb(main):005:1> end while i < 5
0
1
2
3
4
=> nil
irb(main):006:0> begin
irb(main):007:1*    puts i
irb(main):008:1>    i -= 1
irb(main):009:1> end until i == 0
5
4
3
2
1
=> nil
irb(main):010:0>
```

loop

É, nos loops tem um loop. :-)

```
[taq@~]irb
irb(main):001:0> i = 0
=> 0
irb(main):002:0> loop do
irb(main):003:1*    break unless i < 5
```

```
irb(main):004:1>    puts i
irb(main):005:1>    i += 1
irb(main):006:1> end
0
1
2
3
4
=> nil
irb(main):007:0>
```

Importante notar que o loop parece uma estrutura de controle, mas é um método definido no Kernel.

3.4 Comentários no código

Comentários simples podem ser feitos usando # para comentar uma linha, a partir do seu início ou de algum ponto após o código, ou usando **=begin ... =end** para comentar várias linhas:

```
[taq@~]irb
irb(main):001:0> # comentario simples
irb(main):002:0* puts "Oi!" # vou mostrar um comentario grande agora
Oi!
=> nil
irb(main):003:0> =begin
irb(main):004:0= Exemplo de um comentario
irb(main):005:0= com varias linhas dentro
irb(main):006:0= do codigo
irb(main):007:0= =end
irb(main):008:0>
```

Capítulo 4

Classes

Vamos dar uma olhada agora como criar algumas classes. Para fazer isso agora vou deixar de lado o irb e criar um arquivo no editor de texto, para não ter que ficar redigitando tudo de novo.
Arquivos em Ruby tem extensão **.rb**, então vou criar o **Carro.rb**:

```
class Carro
   def initialize(marca,modelo,cor,tanque)
      @marca = marca
      @modelo = modelo
      @cor = cor
      @tanque = tanque
   end
end

gol = Carro::new("Volkswagen","Gol",:azul,50)
puts gol
```

Bom, vamos dar uma geral aqui: o método **initialize** é o construtor das classes em Ruby. Passei parâmetros para o construtor, que são utilizados em suas **variáveis de instância**, que são **marca, modelo, cor e tanque**.

4.1 Variáveis de instância

Variáveis de instância são definidas usando uma **@** na frente do seu nome. Seriam uma analogia às variáveis privadas do Java ™, por exemplo.
O exemplo acima reflete isso, vou repeti-lo aqui só para deixar bem claro:

```
class Carro
   def initialize(marca,modelo,cor,tanque)
      @marca = marca # variavel de instancia
      @modelo = modelo # variavel de instancia
      @cor = cor # variavel de instancia
      @tanque = tanque # variavel de instancia
   end
end

gol = Carro::new("Volkswagen","Gol",:azul,50)
puts gol
```

Prosseguindo, o método new é um **método de classe** (mais sobre isso daqui a pouco), por isso que foi chamado utilizando **::** ao invés de **.**. Ele pode ser chamado usando **.** também, invocando o método da instância, que também pode chamar o da classe.

Depois que temos uma instância de **Carro** na variável **gol**, usamos puts para verificar seu conteúdo:

```
#<Carro:0xb7e56cb4>
```

Feio, não? Vamos dar um jeito nisso. Lembra que usamos **to_s** para converter objetos para Strings em alguns exemplos anteriores? Bom, vamos criar um método to_s para Carro também:

```
class Carro
   def initialize(marca,modelo,cor,tanque)
      @marca = marca
      @modelo = modelo
      @cor = cor
      @tanque = tanque
   end
   def to_s
      "marca:"+@marca+" modelo:"+@modelo+" cor:"+@cor.to_s+" capac.tanque:"+@tanque.to_s
   end
end

gol = Carro::new("Volkswagen","Gol",:azul,50)
puts gol
```

Resultado:

```
marca:Volkswagen modelo:Gol cor:azul capac.tanque:50
```

Melhor não? :-) Reparem que fiz algumas conversões ali. Na cor, fiz o símbolo ser convertido para a sua representação em String (símbolos parecem, mas não são Strings, lembra?) e converti a capacidade do tanque em String também.

Podemos dar uma encurtada no código e eliminar algumas conversões se fizermos:

```
class Carro
   def initialize(marca,modelo,cor,tanque)
      @marca = marca
      @modelo = modelo
      @cor = cor
      @tanque = tanque
   end
   def to_s
      "marca:#{@marca} modelo:#{@modelo} cor:#{@cor} capac.tanque:#{@tanque}"
   end
end

gol = Carro::new("Volkswagen","Gol",:azul,50)
puts gol
```

Reparem no **#{}**: isso é conhecido como **interpolação de expressão**, onde a sequência **#{expressão}** é substituída pelo valor da **expressão**. E ele foi esperto o bastante para invocar o método to_s lá e fazer a conversão da expressão para uma String.

Agora precisamos de um jeito de acessar essas variáveis de instância. Se tentarmos usar

```
puts gol.marca
Carro.rb:15: undefined method ‘marca’ for #<Carro:0xb7e56930> (NoMethodError)
```

vamos receber o erro acima, por que marca é uma variável privada ali da classe. Podemos fazer:

```
class Carro
   def initialize(marca,modelo,cor,tanque)
      @marca = marca
      @modelo = modelo
```

```
      @cor = cor
      @tanque = tanque
   end
   def to_s
      "marca:#{@marca} modelo:#{@modelo} cor:#{@cor} capac.tanque:#{@tanque}"
   end
   def marca
      @marca
   end
   def marca=(marca_nova)
      @marca=marca_nova
   end
end

gol = Carro::new("Volkswagen","Gol",:azul,50)
puts gol
puts gol.marca
gol.marca = "Fiat"
puts gol.marca
```

A analogia aí seria o "setter" e o "getter" do variável marca. Para encurtar ainda mais, Ruby permite que declaremos métodos de acesso à essas variáveis dessa maneira:

```
class Carro
   attr_reader :marca, :modelo, :cor, :tanque
   attr_writer :cor

   def initialize(marca,modelo,cor,tanque)
      @marca = marca
      @modelo = modelo
      @cor = cor
      @tanque = tanque
   end
   def to_s
      "marca:#{@marca} modelo:#{@modelo} cor:#{@cor} capac.tanque:#{@tanque}"
   end
end

gol = Carro::new("Volkswagen","Gol",:azul,50)
puts gol
puts gol.modelo
gol.cor = :branco
puts gol.cor
```

Vejam que eu tenho **readers** e **writers**, que eles usam **símbolos** (olha eles aí) para fazerem referências às variáveis da classe e que só deixei como writer a cor do carro, afinal, que conseguiria mudar um Gol para a Fiat? Pintar ele vocês podem. :-)

E tem mais um jeito de encurtar ainda mais! Podemos usar **attr_accessor** para especificar acessos de escrita e leitura:

```
[taq@~/code/ruby]irb
irb(main):001:0> class Carro
irb(main):002:1>    attr_accessor :cor
irb(main):003:1>    def initialize(cor)
irb(main):004:2>       @cor = cor
irb(main):005:2>    end
irb(main):006:1> end
```

```
=> nil
irb(main):007:0> c = Carro::new(:azul)
=> #<Carro:0xb7dd9458 @cor=:azul>
irb(main):008:0> c.cor.to_s
=> "azul"
irb(main):009:0> c.cor = :branco
=> :branco
irb(main):010:0> c.cor.to_s
=> "branco"
irb(main):011:0>
```

Vocês também podem criar **atributos virtuais**, que são métodos que agem como se fossem atributos da classe. Se por exemplo, tivéssemos um indicador de quantos litros ainda restam no tanque (sendo que cada carro novo sai com o tanque cheio, olha que jóia!) e a montadora exportasse para os EUA e tivesse que indicar quantos galões (unidade de medida dos EUA que equivalem à 3,785 litros) ainda restam no tanque, poderíamos fazer:

```
class Carro
   def initialize(marca,modelo,cor,tanque)
      @marca = marca
      @modelo = modelo
      @cor = cor
      @tanque = tanque
      @litros_no_tanque = tanque
   end
   def to_s
      "marca:#{@marca} modelo:#{@modelo} cor:#{@cor} capac.tanque:#{@tanque}"
   end
   def marca
      @marca
   end
   def marca=(marca_nova)
      @marca=marca_nova
   end
   def litros_no_tanque
      @litros_no_tanque
   end
   def litros_no_tanque=(litros)
      @litros_no_tanque=litros
   end
   def galoes_no_tanque
      @litros_no_tanque/3.785
   end
   def galoes_no_tanque=(galoes)
      @litros_no_tanque = galoes*3.785
   end
end

gol = Carro::new("Volkswagen","Gol",:azul,50)
puts gol
puts "O carro tem #{gol.litros_no_tanque.to_s} litros de combustivel ainda."
puts "Isso corresponde à #{gol.galoes_no_tanque} galões sobrando ainda."
gol.galoes_no_tanque=10
puts "Agora o carro tem #{gol.galoes_no_tanque.to_s} galões de combustivel."
puts "Isso corresponde à #{gol.litros_no_tanque} litros."
```

Rodando o exemplo vamos ter:

```
marca:Volkswagen modelo:Gol cor:azul capac.tanque:50
O carro tem 50 litros de combustivel ainda.
Isso corresponde à 13.2100396301189 galões sobrando ainda.
Agora o carro tem 10.0 galões de combustivel.
Isso corresponde à 37.85 litros.
```

Os métodos **galoes_no_tanque** são os atributos virtuais, métodos que se comportam como se fossem atributos da classe, aos quais você pode atribuir e ler valores normalmente.
Note-se que deu um número bem extenso ali, mas não vamos falar de formatação por enquanto.

Uma coisa interessante em Ruby é que as classes nunca são fechadas. Então, se precisarmos adicionar alguns novos métodos em Carro, *dinâmicamente*, por exemplo, só teríamos que fazer:

```
class Carro
   def novo_metodo
      puts "novo método"
   end
   def mais_um_novo_metodo
      puts "mais um novo método"
   end
end
```

Feito isso, as instâncias de carro já podem acessar seus novos métodos.

Isso pode ser bastante poderoso para alterarmos as classes sem ter que reescrevê-las novamente. Mas se quisermos uma definição nova sem algum método sem precisar sair do programa (e alterar o código antes), podemos fazer:

```
class Carro
   remove_method :mais_um_novo_metodo
end
```

Só para completar nosso papo sobre variáveis de instância, deixe-me dizer que podemos acessar constantes de classe facilmente, assim (saindo um pouco do nosso exemplo do carro):

```
[taq@~]irb
irb(main):001:0> class Valores
irb(main):002:1>    Pi = 3.1515926
irb(main):003:1> end
=> 3.1515926
irb(main):004:0> Valores::Pi
=> 3.1515926
irb(main):005:0>
```

E vale mencionar uma coisa muito importante sobre variáveis de instância: elas podem ser criadas dentro de *qualquer* método da sua classe (no nosso exemplo, usamos o **initialize**, mas elas podem "pipocar" em qualquer método), mas se forem declaradas **logo após a definição da classe** (dentro do corpo da classe) você tem uma **variável de instância de classe** (!!!), que só pode ser acessada usando um método de classe (próxima seção), como no exemplo:

```
[taq@~]irb
irb(main):001:0> class Carro
irb(main):002:1>    @qtde_de_rodas = 4
irb(main):003:1>    def get_qtde_rodas
irb(main):004:2>       @qtde_de_rodas
irb(main):005:2>    end
irb(main):006:1>    def Carro.get_qtde_rodas
irb(main):007:2>       @qtde_de_rodas
irb(main):008:2>    end
```

```
irb(main):009:1> end
=> nil
irb(main):010:0> c = Carro::new
=> #<Carro:0xb7dd2540>
irb(main):011:0> c.get_qtde_rodas
=> nil
irb(main):012:0> Carro::get_qtde_rodas
=> 4
irb(main):013:0>
```

Podemos ver que **c.get_qtde_rodas** retornou nulo, pois ele não consegue ver a variável **qtde_de_rodas**. Então, sempre lembre de declarar suas variáveis de instância dentro de algum método.

Tentando exemplicar graficamente o que ocorreu ali em cima: Onde

- A é a variável de instância da classe. Cada instância da classe Carro (sempre que você usa *new*) tem uma dessas, que pode ser acessada pelo método
- B, *get_qtde_rodas*, que apesar que *parece* que está na instância, pertence a definição da classe Carro e retorna A, que é diferente de
- C, que se encontra na instância *da classe* Carro, e somente pode ser acessada usando o método
- D, que é um *método de classe* e se encontra na metaclasse de Carro.

Tentando resumir, A é propagada para cada instância nova de Carro, e pode ser acessada por B. C pertence à única instância *da classe* Carro, e só pode ser acessada usando D.

4.2 Variáveis de classe

Temos também **variáveis de classe**, que fazendo uma analogia com Java ™seriam as variáveis **estáticas**. São variáveis que estão no contexto **da classe**, e não **da instância**. Há somente uma cópia de uma dessas variável por classe. Variáveis de classe começam com **@@**, e **tem** que ser inicializadas antes de serem usadas.

Vamos imaginar que temos algumas fábricas de *automóveis*, e queremos que criem e guardem estatística de quantos *automóveis* elas produziram, mas também queremos saber quantos *automóveis* foram produzidos no mundo pelas montadoras:

```
class Automovel
   attr_reader :marca, :modelo, :cor, :tanque
   attr_writer :cor

   def initialize(marca,modelo,cor,tanque)
      @marca = marca
      @modelo = modelo
      @cor = cor
```

```
      @tanque = tanque
   end

   def to_s
      "marca:#{@marca} modelo:#{@modelo} cor:#{@cor} capac.tanque:#{@tanque}"
   end
end

class Montadora
   attr_reader :nome, :modelos, :qtde_fabricados

   @@qtde_total_de_automoveis = 0

   def initialize(nome,modelos)
      @nome = nome
      @modelos = modelos
      @qtde_fabricados = 0
   end

   def fabrica(modelo,cor)
   if !@modelos.include? modelo
         puts "ATENÇÃO! A montadora #{@nome} não fabrica o modelo #{modelo}"
         return nil
      end
      puts "Fabricando um #{modelo} na montadora #{@nome} ..."
      @@qtde_total_de_automoveis += 1
      @qtde_fabricados += 1
      Automovel::new(@nome,modelo,cor,@modelos[modelo])
   end

   def Montadora.qtde_total_de_automoveis
      @@qtde_total_de_automoveis
   end
end

# criando as montadoras
volks = Montadora::new("Volkswagen",{"Gol"=>50,"Golf"=>55,"Polo"=>60})
ford  = Montadora::new("Ford",{"Ka"=>45,"Fiesta"=>50,"Focus"=>60})

# criando os automoveis da Volkswagen (com um errinho ali no meio)
automoveis_volks = []
automoveis_volks.push(volks.fabrica("Gol",:azul))
automoveis_volks.push(volks.fabrica("Golf",:preto))
automoveis_volks.push(volks.fabrica("Ka",:verde))
puts "#{volks.nome} fabricou #{volks.qtde_fabricados} automóveis até o momento"
automoveis_volks.each {|v| puts v unless v.nil?}
puts

# criando os automoveis da Ford
automoveis_ford  = []
automoveis_ford.push(ford.fabrica("Ka",:verde))
automoveis_ford.push(ford.fabrica("Fiesta",:vermelho))
automoveis_ford.push(ford.fabrica("Focus",:preto))
puts "#{ford.nome} fabricou #{ford.qtde_fabricados} automóveis até o momento"
automoveis_ford.each {|v| puts v unless v.nil?}
puts
```

```
puts "Qtde total de automóveis no mundo: #{Montadora::qtde_total_de_automoveis}"
```

Quero declarar que eu não faço a mínima idéia das capacidades dos tanques desses veículos, são meramente ilustrativas. :-)
Rodando isso vamos ter:

```
Fabricando um Gol na montadora Volkswagen ...
Fabricando um Golf na montadora Volkswagen ...
ATENÇÃO! A montadora Volkswagen não fabrica o modelo Ka
Volkswagen fabricou 2 automóveis até o momento
marca:Volkswagen modelo:Gol cor:azul capac.tanque:50
marca:Volkswagen modelo:Golf cor:preto capac.tanque:55

Fabricando um Ka na montadora Ford ...
Fabricando um Fiesta na montadora Ford ...
Fabricando um Focus na montadora Ford ...
Ford fabricou 3 automóveis até o momento
marca:Ford modelo:Ka cor:verde capac.tanque:45
marca:Ford modelo:Fiesta cor:vermelho capac.tanque:50
marca:Ford modelo:Focus cor:preto capac.tanque:60

Qtde total de automóveis no mundo: 5
```

Comentando o código, vamos perceber que declarei attr_readers para *nome, modelos e qtde_fabricados*, mas não para a variável de classe *qtde_total_de_automoveis*, declarada logo abaixo. Por que?

O attr_reader cria um métodos de instância (sim, cria!), que lê por exemplo @nome, mas se ela fosse uma variável de classe chamada @@nome, ele não ia conseguir a ver. Então, esse tipo de coisa não serve para variáveis de classe.

Foi por isso que eu declarei um método de classe (???) chamado **qtde_total_de_automoveis**, que é o mesmo nome da variável da classe, e retorna seu valor.

Percebam que eu passei um Hash no construtor de Montadora com os modelos que ela fabrica, juntamente com o seu nome. Quando invocado o método fabrica, ela verifica se fabrica o modelo requisitado.

Se não fabrica, retorna **nil** (olha aí como especificamos valores nulos no Ruby!) e mostra uma mensagem de erro. Ah, sim, podemos usar **?** e **!** em nomes de métodos, inclusive fica melhor até para ler. Melhor que por exemplo is_included.

Então, se a montadora fabricar o modelo que foi requisitado, é incrementado o número de automóveis fabricados e criada e retornada uma nova instância de Carro.

Logo depois criei instâncias de montadoras para a Volskwagen e a Ford (espero não levar um processo por mencionar os nomes das montadoras e dos automóveis, mas é puramente para questões educativas! :-p), e tentei fabricar 3 automóveis em cada.

Podem ver que não consegui fabricar um Ka na Volkswagen, pois ela não fabrica esse automóvel, não é? :-) Depois mostrei o total de automóveis fabricados, 2, e criei 3 na Ford e os listei, mostrando o total de 3 automóveis fabricados.

No final, acessei o **método da classe** para retornar a quantidade total de automóveis montados pelas montadoras.

Uma coisa interessante é que se você tem uma variável de classe e cria uma subclasse com uma variável de classe com o mesmo nome, a da classe pai é a que fica valendo. Por exemplo:

```
class Montadora
   @@qtde_total_de_automoveis = 10
   def Montadora.qtde_total
      @@qtde_total_de_automoveis
   end
def qtde_total
@@qtde_total_de_automoveis
end
end

class GrandeMontadora < Montadora
   @@qtde_total_de_automoveis = 1000
   def GrandeMontadora.qtde_total
      @@qtde_total_de_automoveis
   end
end

puts Montadora.qtde_total
puts GrandeMontadora.qtde_total

t = Montadora.new
puts t.qtde_total

g = GrandeMontadora.new
puts g.qtde_total
```

Todos os resultados vão ser 1000, pois @@qtde_total_de_automoveis é a mesma em Montadora e em GrandeMontadora (que no caso em questão alterou o seu valor para 1000, inclusive de Montadora).

4.3 Métodos de classe

Métodos de classe são aqueles que são acessíveis usando o nome da classe, e não são propagados para suas instâncias. São declarados na forma **Classe.método**, como no exemplo, Montadora.qtde_total_de_automoveis.

Podem ser definidos também com métodos *singleton* (mais sobre isso logo abaixo) da própria classe, e também não precisam de uma instância da classe para serem chamados (algo como os métodos estáticos em Java ™).

Lógico que eu poderia ter criado esse método como um método de instância e chamado, por exemplo, *ford.qtde_total_de_automoveis* ou *volks.qtde_total_de_automoveis* que ia dar na mesma. Mas usando como um método da classe fica mais visível e legível o escopo global da coisa, além der ser uma melhor prática.

4.4 Executando métodos da classe pai

Podemos executar métodos da classe pai usando **super**. Importante notar que super envia todos os parâmetros recebidos no método para o método da classe pai. Se recebemos uma exceção do tipo ArgumentError quando chamou super, verifique o número de parâmetros que está recebendo na classe filha e também na classe pai.

Se o número de parâmetros for diferente, podemos usar **super(p1,p2...)** para adequar a chamada de acordo com os parâmetros da classe pai.

Isso funciona:

```
class Teste
```

```
   def initialize(p1,p2)
      @v1 = p1
      @v2 = p2
   end
   def show
      puts "#{@v1},#{@v2}"
   end
end
class Teste2 < Teste
   def initialize(p1,p2)
      super
   end
end

t1 = Teste::new(1,2)
t2 = Teste2::new(3,4)

t1.show
t2.show
```

Resultado:

```
1,2
3,4
```

Isso **não** funciona:

```
class Teste
def initialize(p1,p2)
@v1 = p1
@v2 = p2
end
def show
puts "#{@v1},#{@v2}"
end
end
class Teste2 < Teste
def initialize(p1)
super
end
end

t1 = Teste::new(1,2)
t2 = Teste2::new(3)

t1.show
t2.show
```

Resultado:

```
super.rb:12:in `initialize': wrong number of arguments (1 for 2) (ArgumentError)
        from super.rb:12:in `initialize'
        from super.rb:17:in `new'
        from super.rb:17
```

Mas isso funciona:

```
class Teste
   def initialize(p1,p2)
```

```
      @v1 = p1
      @v2 = p2
   end
   def show
      puts "#{@v1},#{@v2}"
   end
end
class Teste2 < Teste
   def initialize(p1)
      super(p1,0)
   end
end

t1 = Teste::new(1,2)
t2 = Teste2::new(3)

t1.show
t2.show
```

Resultado:

```
1,2
3,0
```

4.5 Redefinindo operadores

Lembram quando eu falei que era permitido redefinir *alguns* operadores? Vamos fazer aqui um teste, com o operador +, e já que falei tanto de carros e montadoras, vou criar uma classe chamada CaixaDeParafusos, onde uma pode ser somada com outra:

```
class CaixaDeParafusos
   attr_accessor :qtde
   def initialize(qtde)
      raise Exception::new("Qtde tem que ser maior q zero") unless qtde > 0
      @qtde = qtde
   end
   def +(outra)
      if outra.is_a? CaixaDeParafusos
         @qtde += outra.qtde
         outra.qtde = 0
      elsif outra.is_a? Fixnum
         @qtde += outra
      end
   self
   end
end

caixa1 = CaixaDeParafusos::new(10)
caixa2 = CaixaDeParafusos::new(30)
print "caixa1 tem #{caixa1.qtde} parafusos\n"
print "caixa2 tem #{caixa2.qtde} parafusos\n\n"

caixa1 += caixa2
print "caixa1 tem #{caixa1.qtde} parafusos\n"
print "caixa2 tem #{caixa2.qtde} parafusos\n\n"

caixa2 += 100
```

```
print "caixa1 tem #{caixa1.qtde} parafusos\n"
print "caixa2 tem #{caixa2.qtde} parafusos\n\n"
```

Resultado:

```
caixa1 tem 10 parafusos
caixa2 tem 30 parafusos

caixa1 tem 40 parafusos
caixa2 tem 0 parafusos

caixa1 tem 40 parafusos
caixa2 tem 100 parafusos
```

Para evitar que sejam criadas caixas vazias, eu gerei uma **exceção** (que vamos estudar mais tarde), onde somente caixas com uma quantidade maior que 0 sejam criadas. Logo depois, *redefini o operador (método) +*, utilizando **def +(outra)**.

Para dar um segurança maior ali, vou permitir que a classe seja somada apenas com outra caixa de parafusos (despeja uma dentro da outra!) ou com parafusos individuais, assim sendo, vou verificar se o parâmetro que foi passado é uma caixa de parafusos ou um número, utilizando o método **is_a?**, que retorna o tipo do objeto.

Se for um objeto do tipo **CaixaDeParafusos**, eu adiciono a quantidade da segunda caixa na primeira, e lógico, como não há o milagre da multiplicação dos parafusos, faço uma consistência ali e zero a quantidade da segunda caixa.

Se forem adicionados apenas parafusos individuais (Fixnum), eu somente aumento a quantidade da caixa e pronto.
Reparem no final, que estou *retornando o próprio objeto*, isso é muito importante. Se eu deixasse sem esse retorno, *caixa1* valeria o que fosse avaliado por último ali e perderíamos nosso objeto!

4.6 Herança

Viram que eu troquei a classe Carro por uma chamada Automovel? Fiz isso por que podemos criar mais do que carros em cada uma dessas montadoras, certo?

Por exemplo, caminhões também! Então, criamos uma classe básica chamada Automovel e derivamos Carro e Caminhao dela, e quando fabricando na montadora, se tiver tanque menor que 100 litros, crio um carro, se for maior, crio um caminhão (aproveitei e entre nos sites das montadoras para ver a capacidade real dos tanques de combustível ehehe).

Também sobreescrevi o método *to_s* para retornar se é um Carro ou Caminhao:

```
class Automovel
   attr_reader :marca, :modelo, :cor, :tanque
   attr_writer :cor

   def initialize(marca,modelo,cor,tanque)
      @marca = marca
      @modelo = modelo
      @cor = cor
      @tanque = tanque
   end

   def to_s
      "marca:#{@marca} modelo:#{@modelo} cor:#{@cor} capac.tanque:#{@tanque}"
```

```
   end
end

class Carro < Automovel
   def to_s
      "Carro:"+super
   end
end

class Caminhao < Automovel
   def to_s
      "Caminhão:"+super
   end
end

class Montadora
   attr_reader :nome, :modelos, :qtde_fabricados

   @@qtde_total_de_automoveis = 0

   def initialize(nome,modelos)
      @nome = nome
      @modelos = modelos
      @qtde_fabricados = 0
   end

   def fabrica(modelo,cor)
      if !@modelos.include? modelo
         puts "ATENÇÃO! A montadora #{@nome} não fabrica o modelo #{modelo}"
         return nil
      end
      puts "Fabricando um #{modelo} na montadora #{@nome} ..."
      @@qtde_total_de_automoveis += 1
      @qtde_fabricados += 1
      # se tiver um tanque com menos de 100 litros, é um carro
      if @modelos[modelo] < 100
         Carro::new(@nome,modelo,cor,@modelos[modelo])
      else
         Caminhao::new(@nome,modelo,cor,@modelos[modelo])
      end
   end

   def Montadora.qtde_total_de_automoveis
      @@qtde_total_de_automoveis
   end
end

# criando as montadoras
volks = Montadora::new("Volkswagen",{"Gol"=>51,"Golf"=>55,"Polo"=>45,"17210C"=>275})
ford  = Montadora::new("Ford",{"Ka"=>42,"Fiesta"=>42,"Focus"=>55,"C-815"=>150})

# criando os automoveis da Volkswagen (com um errinho ali no meio)
automoveis_volks = []
automoveis_volks.push(volks.fabrica("Gol",:azul))
automoveis_volks.push(volks.fabrica("Golf",:preto))
automoveis_volks.push(volks.fabrica("Ka",:verde))
```

```
automoveis_volks.push(volks.fabrica("17210C",:branco))
puts "#{volks.nome} fabricou #{volks.qtde_fabricados} automóveis até o momento"
automoveis_volks.each {|v| puts v unless v.nil?}
puts

# criando os automoveis da Ford
automoveis_ford  = []
automoveis_ford.push(ford.fabrica("Ka",:verde))
automoveis_ford.push(ford.fabrica("Fiesta",:vermelho))
automoveis_ford.push(ford.fabrica("Focus",:preto))
automoveis_ford.push(ford.fabrica("C-815",:branco))
puts "#{ford.nome} fabricou #{ford.qtde_fabricados} automóveis até o momento"
automoveis_ford.each {|v| puts v unless v.nil?}
puts

puts "Qtde total de automóveis no mundo: #{Montadora::qtde_total_de_automoveis}"
```

Rodando isso vamos ter:

```
Fabricando um Gol na montadora Volkswagen ...
Fabricando um Golf na montadora Volkswagen ...
ATENÇÃO! A montadora Volkswagen não fabrica o modelo Ka
Fabricando um 17210C na montadora Volkswagen ...
Volkswagen fabricou 3 automóveis até o momento
Carro:marca:Volkswagen modelo:Gol cor:azul capac.tanque:51
Carro:marca:Volkswagen modelo:Golf cor:preto capac.tanque:55
Caminhão:marca:Volkswagen modelo:17210C cor:branco capac.tanque:275

Fabricando um Ka na montadora Ford ...
Fabricando um Fiesta na montadora Ford ...
Fabricando um Focus na montadora Ford ...
Fabricando um C-815 na montadora Ford ...
Ford fabricou 4 automóveis até o momento
Carro:marca:Ford modelo:Ka cor:verde capac.tanque:42
Carro:marca:Ford modelo:Fiesta cor:vermelho capac.tanque:42
Carro:marca:Ford modelo:Focus cor:preto capac.tanque:55
Caminhão:marca:Ford modelo:C-815 cor:branco capac.tanque:150

Qtde total de automóveis no mundo: 7
```

Viram que para criar uma classe a partir de outra, é só usar o sinal $<$ (*extends* no Java ™e PHP), ficando **classe filha** $<$ **classe pai**? :-)

Fica uma dica: se quisermos reter um método que foi reescrito (ou simplesmente criarmos um novo método a partir de um já definido), podemos usar **alias** para fazer isso. Por exemplo:

```
[taq@~]irb
irb(main):001:0> def teste
irb(main):002:1>    puts "Oi!"
irb(main):003:1> end
=> nil
irb(main):004:0> alias :teste_antigo :teste
=> nil
irb(main):005:0> teste
Oi!
=> nil
irb(main):006:0> teste_antigo
Oi!
```

```
=> nil
irb(main):007:0> def teste
irb(main):008:1>    puts "Tchau!"
irb(main):009:1> end
=> nil
irb(main):010:0> teste
Tchau!
=> nil
irb(main):011:0> teste_antigo
Oi!
=> nil
irb(main):012:0>
```

No caso de redefinição em uma classe:

```
[taq@~/code/ruby]irb
irb(main):001:0> class Teste
irb(main):002:1>    attr_reader :teste
irb(main):003:1>    def teste
irb(main):004:2>       puts "#{self.class}: Oi!"
irb(main):005:2>    end
irb(main):006:1> end
=> nil
irb(main):007:0> class Teste2 < Teste
irb(main):008:1>    alias :teste_antigo :teste
irb(main):009:1>    def teste
irb(main):010:2>       puts "#{self.class}: Tchau!"
irb(main):011:2>    end
irb(main):012:1> end
=> nil
irb(main):013:0> t1 = Teste::new
=> #<Teste:0xb7e4b4cc>
irb(main):014:0> t2 = Teste2::new
=> #<Teste2:0xb7e3f410>
irb(main):015:0> t1.teste
Teste: Oi!
=> nil
irb(main):016:0> t2.teste
Teste2: Tchau!
=> nil
irb(main):017:0> t2.teste_antigo
Teste2: Oi!
=> nil
irb(main):018:0>
```

4.7 Duplicando

Já vimos como duplicar objetos usando uma String. Vamos dar uma olhada usando uma classe definida por nós:

```
class Teste
   attr_reader :desc, :criada_em
   def initialize(desc)
      @desc = desc
      @criada_em = Time.now
   end
end
```

```
t1 = Teste::new("Classe de teste")
sleep(2) # lalalalá ...
t2 = t1.dup

puts "id de t1:#{t1.object_id}"
puts "id de t2:#{t2.object_id}"

puts "t1 (#{t1.desc}) criada em #{t1.criada_em}"
puts "t2 (#{t2.desc}) criada em #{t2.criada_em}"
```

Resultado:

```
id de t1:-604851218
id de t2:-604851228
t1 (Classe de teste) criada em Thu Jan 20 09:17:10 BRST 2005
t2 (Classe de teste) criada em Thu Jan 20 09:17:10 BRST 2005
```

Dando uma olhadinha no programa, vamos ver que estou guardando o horário de criação do objeto usando **Time.now**, e para criar a primeira e a segundo eu espero dois segundos (através de **sleep(2)**). Verificando os *id's* podemos ver que são objetos distintos, que foram duplicados corretamente.

Mas isso ás vezes pode ser um problema no caso do horário de criação do objeto. No caso da segunda classe, o horário teria que ser dois segundos a mais que na primeira. Para que possamos resolver esse tipo de problema, Ruby nos dá o método **initialize_copy**, que é acessado sempre que um objeto é duplicado, recebendo como parâmetro o objeto original. Então, podemos fazer:

```
class Teste
attr_reader :desc, :criada_em
def initialize(desc)
@desc = desc
@criada_em = Time.now
end
def initialize_copy(outra)
@desc = outra.desc + " - clone!"
@criada_em = Time.now
end
end

t1 = Teste::new("Classe de teste")
sleep(2) # lalalalá ...
t2 = t1.dup

puts "id de t1:#{t1.object_id}"
puts "id de t2:#{t2.object_id}"

puts "t1 (#{t1.desc}) criada em #{t1.criada_em}"
puts "t2 (#{t2.desc}) criada em #{t2.criada_em}"
```

Resultado:

```
id de t1:-604851488
id de t2:-604851498
t1 (Classe de teste) criada em Thu Jan 20 09:23:20 BRST 2005
t2 (Classe de teste - clone!) criada em Thu Jan 20 09:23:22 BRST 2005
```

Agora sim! A segunda classe teve seu horário de criação corretamente apontado, e de quebra lemos a descrição da primeira e concatenamos a string - *clone!* no final para indicar que é um objeto duplicado.

Quando duplicamos um objeto usando **dup**, tem um pequeno porém: ele não copia o estado de um objeto. Por exemplo, se usarmos uma string "congelada" com **freeze**:

```
[taq@~]irb
irb(main):001:0> s1 = "teste"
=> "teste"
irb(main):002:0> s1.freeze
=> "teste"
irb(main):003:0> s2 = s1.dup
=> "teste"
irb(main):004:0> s1.frozen?
=> true
irb(main):005:0> s2.frozen?
=> false
```

O que aconteceu foi que, usando *dup*, o *estado* de "congelado" do objeto não foi copiado.
Para que o estado seja copiado, temos que usar **clone**:

```
[taq@~]irb
irb(main):001:0> s1 = "teste"
=> "teste"
irb(main):002:0> s1.freeze
=> "teste"
irb(main):003:0> s2 = s1.clone
=> "teste"
irb(main):004:0> s1.frozen?
=> true
irb(main):005:0> s2.frozen?
=> true
irb(main):006:0>
```

Convém mencionar que Ruby faz, com =, *dup* e *clone*, uma *shallow copy* do objeto, ou seja, duplicando efetivamente o objeto, mas copiando, **não duplicando**, as referências dos objetos dentro dele.

Vamos dar uma olhada nisso:

```
class Num
   def initialize(v)
      @valor=v
   end
   def proximo(p)
      @proximo=p
   end
   def imprime
      puts "#{@valor}:#{self.object_id}"
      @proximo.imprime if !@proximo.nil?
   end
end

n1 = Num.new(1)
n2 = Num.new(2)
n3 = Num.new(3)

n1.proximo(n2)
n2.proximo(n3)
puts "listando"
n1.imprime
```

```
n4 = n2.dup
n1.proximo(n4)
puts "listando"
n1.imprime
```

Rodando:

```
listando
1:-604869880
2:-604869960
3:-604869970
listando
1:-604869880
2:-604869990 <- diferente!
3:-604869970
```

Vamos dar uma analisada na coisa ali: criei três instâncias de *Num*, indiquei n2 como o próximo de n1 e n3 como próximo de n2, e imprimi os id's.

Logo depois criei n4 **duplicando** n2, indiquei n4 como próximo de n1 e imprimi os id's novamente. O que aconteceu?

O id de n1 permanece o mesmo (-604869880) pois não o alteramos. O id do objeto seguinte de n1 foi alterado, (indiquei com <–) pois **duplicamos** n2, e apesar de permanecer o mesmo valor, o objeto foi alterado, é um objeto novo, e conferindo os id's podemos ver isso: na primeira impressão era -604869960 e na segunda, -604869990.

O interessante ali é que, mesmo que seja um objeto novo, reparem no id do objeto seguinte: continua o mesmo (-607869970)! Isso por que quando duplicamos n2, foi criado um objeto novo a partir de n2, **com todas as referências originais de n2 dentro dele**, ou seja, n3 está intocado lá. Esse é o conceito de *shallow copy*.

Se quisermos um *deep copy*, *duplicando também* os objetos contidos dentro do objeto que estamos duplicando, temos que usar o conceito de "marshaling", ou *serializar* o objeto, gravando ele (e seus objetos internos) como um fluxo de bytes e recriando um novo objeto (com novos objetos dentro) novamente:

```
class Num
   def initialize(v)
      @valor=v
   end
   def proximo(p)
      @proximo=p
   end
   def imprime
      puts "#{@valor}:#{self.object_id}"
      @proximo.imprime if !@proximo.nil?
   end
end

n1 = Num.new(1)
n2 = Num.new(2)
n3 = Num.new(3)

n1.proximo(n2)
n2.proximo(n3)
puts "listando"
n1.imprime
```

```
n4 = Marshal.load(Marshal.dump(n2))
n1.proximo(n4)
puts "listando"
n1.imprime
```

Rodando:

```
listando
1:-604869930
2:-604870010
3:-604870020
listando
1:-604869930
2:-604870190 <- diferente!
3:-604870220 <- diferente!
```

Opa! Olhem as duas últimas referências agora: os id's estão *ambos* diferentes, ou seja, duplicamos n2 e os objetos que ele continha (n3). O que aconteceu ali foi que usei **Marshal.dump** para converter o objeto em um fluxo de bytes e depois **Marshal.load** para recriá-lo novamente.

4.8 Controle de acesso

Podemos também limitar o controle de acesso aos métodos, sendo:

- **Públicos**
  Podem ser acessados por qualquer método em qualquer objeto.
- **Privados**
  Só podem ser chamados dentro de seu próprio objeto, mas nunca é possível acessar um método privado de outro objeto, mesmo se o objeto que chama seja uma subclasse de onde o método foi definido.
- **Protegidos**
  Podem ser acessados em seus descendentes.

Note que Ruby é diferente de algumas outras linguagens de programação em relação aos métodos privados e protegidos.

Vamos fazer alguns exemplos aqui, vou ter 4 objetos: dois da mesma classe (que vamos chamar de classe principal aqui nos exemplos), um derivado da classe principal (a classe filha, que tem um método a mais, "novo_oi" e outra completamente estranho no contexto da classe principal. Vamos ter um método chamado "diga_oi" e um chamado "cumprimente", que recebe um outro objeto como seu parâmetro, executa o seu próprio "diga_oi" e depois (tenta, em alguns casos) executar o "diga_oi" do objeto que foi passado como parâmetro.

4.8.1 Métodos públicos

```
class TestePublico
   def diga_oi
      "oi!"
   end
   def cumprimente(outra)
      puts "Eu cumprimento:"+diga_oi
      puts "A outra classe cumprimenta:"+outra.diga_oi
   end
   public :diga_oi # nao precisa, está default
end
```

```
class TestePublicoFilha < TestePublico
   def novo_oi
      puts "Filha cumprimenta:"+diga_oi
   end
end

class ClasseEstranha
   def cumprimente(outra)
      puts "Classe estranha acessando:"+outra.diga_oi
   end
end

pub1 = TestePublico::new
pub2 = TestePublico::new
filha= TestePublicoFilha::new
estr = ClasseEstranha::new

pub1.cumprimente(pub2)
filha.novo_oi
estr.cumprimente(pub1)
```

Resultado:

```
Eu cumprimento:oi!
A outra classe cumprimenta:oi!
Filha cumprimenta:oi!
Classe estranha acessando:oi!
```

A instância de pub1 chamou o método "diga_oi" de pub2. Perfeitamente aceitável, o método de pub2 é público e pode ser chamado até pelo Zé do Boteco, inclusive pela classe derivada (TestePublicoFilha) e por outra que não tem nada a ver, hierárquicamente falando. **Todos** acessam o método "diga_oi" da classe TestePublico.

> No método *cumprimente* eu mostrei a classe acessando outra classe usando como referência o parâmetro *outra*, que é outra classe recebida. Podemos referenciar a própria classe usando **self** também:
>
> ```
> puts ‘‘Eu cumprimento:’’+self.diga_oi
> ```
>
> mas isso não é necessário de explicitar.

4.8.2 Métodos privados

```
class TestePrivado
   def diga_oi
      "oi!"
   end
   def cumprimente(outra)
      puts "Eu cumprimento:"+diga_oi
      puts "A outra classe cumprimenta:"+outra.diga_oi
   end
   private :diga_oi
end

class TestePrivadoFilha < TestePrivado
   def novo_oi
      puts "Filha cumprimenta:"+diga_oi
```

```
   end
end

class ClasseEstranha
   def cumprimente(outra)
      puts "Classe estranha acessando:"+outra.diga_oi
   end
end

pri1 = TestePrivado::new
pri2 = TestePrivado::new
filha= TestePrivadoFilha::new
estr = ClasseEstranha::new

filha.novo_oi
pri1.cumprimente(pri2) # pri1 não consegue acessar pri2
estr.cumprimente(pri1) # estranha não consegue acessar pri1
```

Resultado:

```
TestePrivado.rb:7:in ‘cumprimente’: private method ‘diga_oi’
called for #<TestePrivado:0xb7e573e0> (NoMethodError)
from TestePrivado.rb:30
Filha cumprimenta:oi!
Eu cumprimento:oi!
```

Vejam que aqui eu deliberadamente chamei o método da filha primeiro, e ela aciona o método herdado “diga_oi” (privado, mas herdado - Java ™dá um erro de compilação nesse caso, pois a classe só herda métodos *públicos ou protegidos*) e a classe principal aciona seu “diga_oi” mas **não consegue acessar o método da outra classe, mesmo que as duas pertençam ao mesmo objeto**.
O erro foi ocasionado por causa disso.

4.8.3 Métodos protegidos

```
class TesteProtegido
   def diga_oi
      "oi!"
   end
   def cumprimente(outra)
      puts "Eu cumprimento:"+diga_oi
      puts "A outra classe cumprimenta::"+outra.diga_oi
   end
   protected :diga_oi
end

class TesteProtegidoFilha < TesteProtegido
   def novo_oi
      puts "Filha cumprimenta:"+diga_oi
   end
end

class ClasseEstranha
   def cumprimente(outra)
      puts "Classe estranha acessando:"+outra.diga_oi
   end
end
```

```
pro1 = TesteProtegido::new
pro2 = TesteProtegido::new
filha= TesteProtegidoFilha::new
estr = ClasseEstranha::new

pro1.cumprimente(pro2) # pro1 consegue acessar pro2
filha.novo_oi
estr.cumprimente(pro1) # estranha não consegue acessar pro1
```

Rodando isso, temos:

```
TesteProtegido.rb:20:in `cumprimente': protected method `diga_oi'
called for #<TesteProtegido:0xb7e57430> (NoMethodError)
from TesteProtegido.rb:31
Eu cumprimento:oi!
A outra classe cumprimenta::oi!
Filha cumprimenta:oi!
```

A própria classe chamou seu método, a classe que foi passada como parâmetro (que é do mesmo tipo) teve seu método chamado também, a classe filha (que é derivada, portanto "da mesma família") chamou também, mas uma classe "estranha" não teve exito ao chamar aquele método. Ele só é acessível para quem é da "turminha". :-)

Fazendo uma analogia ...

Vamos supor que você seja dono de um restaurante. Como você não quer que seus fregueses fiquem apertados você manda fazer um banheiro para o pessoal, e nada impede também que apareça algum maluco da rua apertado, entre no restaurante e use seu banheiro. Esse banheiro é **público**.

Para seus empregados, você faz um banheirinho mais caprichado, que só eles tem acesso. Esse banheiro é **protegido**, sendo que só quem é do restaurante tem acesso.

Mas você sabe que tem um empregado seu lá que tem uns problemas e ao invés de utilizar o banheiro, ele o inutiliza. Como você tem enjoôs com esse tipo de coisa, manda fazer um banheiro **privado** para você, que só você pode usar.

Se foi uma boa analogia não sei, mas já dei a dica de como fazer se um dia você tiver um restaurante. :-)

4.8.4 Especificando o tipo de acesso

Utilizamos, nos exemplos acima, símbolos para especificar o tipo de acesso:

```
class Teste
   def diga_oi
      "oi"
   end
   def cumprimente
      puts diga_oi
   end
   public  :cumprimente
   private :diga_oi
end

t = Teste::new
t.cumprimente
```

Ali foi dado o tipo de acesso (**public**, que é assumido por default, ou **private**), logo em seguida os símbolos que representam os métodos, separados por vírgula, se necessário.
Se tivéssemos uma situação do tipo:

```
class Teste
   def initialize
      @qtde = 0
   end
   def qtde=(qtde)
      @qtde=qtde
   end
   def qtde
      @qtde
   end
   def insira_dez(outra)
      outra.qtde = 10
   end
protected :qtde
end

t1 = Teste::new
t2 = Teste::new
t1.insira_dez(t2)
puts t1.qtde
puts t2.qtde
```

Dando uma analisada no código, eu gostaria que a quantidade só pudesse ser alterada por uma instância da mesma classe ou filha de Teste, mas gostaria que qualquer um pudesse ler a quantidade. Utilizando a forma **protected: qtde**, eu limitei ambos os métodos, então vou ter um erro quando tentar ler a quantidade:

```
TipoAcesso.rb:20: protected method `qtde' called for #<Teste:0xb7e57980 @qtde=0> (NoMethodError)
```

Uma maneira de solucionar o problema seria

```
class Teste
   def initialize
      @qtde = 0
   end
   def qtde=(qtde)
      @qtde=qtde
   end
   def qtde
      @qtde
   end
   def insira_dez(outra)
      outra.qtde = 10
   end
   protected :qtde=
   public    :qtde
end

t1 = Teste::new
t2 = Teste::new
t1.insira_dez(t2)
puts t1.qtde
puts t2.qtde
```

Resultado:

```
0
10
```

Deixei explícito que o método **qtde=** é protegido, enquanto que o **qtde** é público.
Outra forma de solucionar esse tipo de coisa é especificando quais partes do código serão públicas, protegidas ou privadas:

```
class Teste
   def initialize
      @qtde = 0
   end

   # métodos protegidos aqui
   protected
   def qtde=(qtde)
      @qtde=qtde
   end

   # métodos públicos aqui
   public
   def qtde
      @qtde
   end

   def insira_dez(outra)
      outra.qtde = 10
   end

   # métodos privados aqui
private
end

t1 = Teste::new
t2 = Teste::new
t1.insira_dez(t2)
puts t1.qtde
puts t2.qtde
```

O resultado é o mesmo acima.

Uma coisa que vale lembrar aqui é que métodos protegidos ou privados não podem ser chamados na forma <**instância**>.<**método**>, de fora do objeto. Se colocarmos o método **insira_dez** na seção privada ou protegida, não poderemos o chamar da forma ali acima mesmo sendo t1 ou t2 classes Teste, que é onde ele está definido.

Uma pequena nota: o método **initialize** é automaticamente marcado como **private**.

E o controle de acesso é determinado dinâmicamente.

Essa é uma das grandes diferenças de Ruby com outras linguagens de programação. Se você tiver um código com controle restritivo, só vai ter um erro de acesso quando tentar acessá-lo quando rodar o programa. Mas também pode, dinâmicamente, alterar o tipo de controle de acesso, do mesmo jeito que faz com os métodos da classe.

4.9 Classes e métodos singleton

Para mostrar um pouco os métodos de acesso, inclusive um que não vimos aqui, vamos construir uma classe **singleton**, que é uma classe que só tem uma instância, compartilhada durante a execução de um programa.

Vamos imaginar que você tenha uma classe que abra uma conexão com seu banco de dados, **uma vez somente**, e retorne o handle dessa conexão:

```
class DatabaseConnection
   # new se transforma em um método privado DA CLASSE,
   # ou seja, você não pode mais invocá-lo de fora da classe.
   private_class_method :new

   # handle da conexão
   @@my_instance = nil
   @connection_handle = nil

   def DatabaseConnection.create
      @@my_instance = new unless @@my_instance
      @@my_instance
   end

   def get_connection
   puts "Retornando a conexão com o banco ..."
      @connection_handle = open_connection unless @connection_handle
      @connection_handle
   end

   # imagine que aqui é onde você abre a conexão e retorna o handle
   def open_connection
      puts "Abrindo conexão com o banco ..."
      Time.now.to_i # vou retornar o número de segundos na hora como referencia
   end
end

d1 = DatabaseConnection::create
d2 = DatabaseConnection::create

puts "id de d1: #{d1.object_id}"
puts "id de d2: #{d2.object_id}"

puts "handle de d1: #{d1.get_connection}"
puts "handle de d2: #{d2.get_connection}"
```

Rodando o exemplo, teremos:

```
id de d1: -604849620
id de d2: -604849620
Retornando a conexão com o banco ...
Abrindo conexão com o banco ...
handle de d1: 1104665183
Retornando a conexão com o banco ...
handle de d2: 1104665183
```

Repare que o id da instância é o mesmo, assim como o handle (criado como um número único ali só para efeito de exemplo) é o mesmo também. Teremos apenas uma instância dessa classe, com apenas um handle de conexão, durante a execução do seu programa.

Também podemos ter métodos singleton, que são *métodos que residem em apenas uma instância da classe*. Por exemplo:

```
[taq@~/code/ruby]irb
irb(main):001:0> class Motocicleta
irb(main):002:1> end
=> nil
irb(main):003:0> intruder = Motocicleta::new
=> #<Motocicleta:0xb7de0640>
irb(main):004:0> minha_intruder = Motocicleta::new
=> #<Motocicleta:0xb7dddd14>
irb(main):005:0> def minha_intruder.acende_farol_de_caveirinha
irb(main):006:1>    puts "Farol de caveirinha aceso!"
irb(main):007:1> end
=> nil
irb(main):008:0> minha_intruder.acende_farol_de_caveirinha
Farol de caveirinha aceso!
=> nil
irb(main):009:0> intruder.acende_farol_de_caveirinha
NoMethodError: undefined method `acende_farol_de_caveirinha' for #<Motocicleta:0xb7de0640>
        from (irb):9
irb(main):010:0>
```

Defini uma classe Motocicleta (eu adoro motos!), criei duas classes (eu tinha uma Intruder!), e logo após criadas as classes, *defini um método único e exclusivamente para minha_intruder* (customização é legal!).

Só esse objeto que tem um método chamado *acende_farol_de_caveirinha*, a outra instância da Motocicleta não tem, tanto que quando fui chamar o método na "original de fábrica", recebi um erro.

Um jeito **muito** mais fácil de se usar singleton em Ruby é utilizar o módulo *Singleton* (módulos tem seu próprio capítulo mais á frente), dessa maneira:

```
irb(main):001:0> require "singleton"
=> true
irb(main):002:0> class Teste
irb(main):003:1>    include Singleton
irb(main):004:1> end
=> Teste
irb(main):005:0> c1 = Teste.new
NoMethodError: private method `new' called for Teste:Class
        from (irb):5
irb(main):006:0> c1 = Teste.instance
=> #<Teste:0xb7db8948>
irb(main):007:0> c2 = Teste.instance
=> #<Teste:0xb7db8948>
irb(main):008:0> c1.object_id
=> -605174620
irb(main):009:0> c2.object_id
=> -605174620
irb(main):010:0>
```

Olhem que facilidade. Pedi para incluir o módulo Singleton, e pronto!

Podem notar que eu gerei um erro proposital: tentei utilizar o método **new** para criar uma instância nova de Teste, e fui proibido terminantemente de fazer isso. Quando se utiliza o módulo *Singleton*, fazemos isso usando o método *instance*, como no exemplo acima. Notem também que *c1* e *c2* tem o mesmo *object_id*.

Capítulo 5

Entrada e saída

5.1 Fluxos simples

Os fluxos simples de entrada e saída são funções como:

```
[taq@~]irb
irb(main):001:0> puts "Digite seu nome:"
Digite seu nome:
=> nil
irb(main):002:0> nome = gets
TaQ
=> "TaQ\n"
irb(main):003:0> puts "Oi, #{nome}"
Oi, TaQ
=> nil
irb(main):004:0>
```

Usamos puts para mostrar uma mensagem no fluxo de saída (lembre-se que puts usa um salto de linha no final) e **gets** para pegar a entrada do usuário, na variável **nome**.

Fica a dica que a variável **$_** se refere ao último input que ocorreu.

Vamos declarar duas variáveis, nome="TaQ" e idade=33 (por enquanto, o tempo urge ...) e ver algumas outras funções de entrada e saída simples:

- **print** - Imprime para a saída. Ex: **print nome," tem ",idade," anos."**
- **printf** - Saída formatada similar à printf da linguagem C. Ex: **printf "%s tem %02d anos n"**
- **putc** - Imprime o caracter. Ex: **putc 7** vai soar um bip.
- **readlines** - Lê várias linhas da entrada até terminar com Ctrl+D.

Podemos utilizar também as constantes globais:

- **STDIN** - Input padrão, default é o teclado
- **STDOUT** - Output padrão, default é o monitor
- **STDERR** - Output de erro padrão, default é o monitor

Por exemplo

```
[taq@~]irb
irb(main):001:0> STDOUT.print "digite uma mensagem (termine com ENTER):"
```

```
digite uma mensagem (termine com ENTER):=> nil
irb(main):002:0> msg = STDIN.gets
Oi, mundo!
=> "Oi, mundo!\n"
irb(main):003:0> STDERR.print "Imprimindo na saída de erros:",msg
Imprimindo na saída de erros:Oi, mundo!
=> nil
irb(main):004:0>
```

5.2 Fluxos mais complexos

5.2.1 Arquivos

Lendo arquivos

Podemos criar um handle para um arquivo dessa maneira:

```
[taq@~]irb
irb(main):001:0> f = File.new("teste.txt","r")
=> #<File:teste.txt>
irb(main):002:0> f.close
=> nil
irb(main):003:0>
```

Esse é um bom modo de se ter controle sobre o handle do arquivo. Se você quiser abrir um arquivo, processar um pouco de código, e automagicamente fechá-lo, você pode utilizar o método **open**, que aceita um bloco de código:

```
[taq@~]irb
irb(main):001:0> File.open("teste.txt") do |handle|
irb(main):002:1*    handle.each_line do |linha|
irb(main):003:2*       puts linha
irb(main):004:2>    end
irb(main):005:1> end
Teste de leitura, primeira linha.
Aqui é a linha 2.
Estou na linha 3, acho que vou terminar.
Terminei na linha 4, tchau.
=> #<File:teste.txt (closed)>
irb(main):006:0>
```

Aqui usei o método **open** para abrir o handle e o método **each_line**, que lê cada linha do arquivo. Então, **open** abriu o arquivo, passando como parâmetro seu handle, e executou each_line com esse handle, e para cada linha lida, executou o bloco de código informado.

Você pode alterar o caracter (ou String) que quebra as linhas do arquivo, usando *each_line(expr)*. Por exemplo, quebrando a cada palavra "linha":

```
[taq@~]irb
irb(main):001:0> File.open("teste.txt") do |handle|
irb(main):002:1*    handle.each_line("linha") do |linha|
irb(main):003:2*       puts linha
irb(main):004:2>    end
irb(main):005:1> end
Teste de leitura, primeira linha
.
Aqui é a linha
 2.
```

```
Estou na linha
 3, acho que vou terminar.
Terminei na linha
 4, tchau.
=> #<File:teste.txt (closed)>
irb(main):006:0>
```

Note-se que *each_line* é um iterador, que lê as linhas de acordo com a demanda. Se você quiser ler todas as linhas do arquivo de uma vez, pode usar **readlines**:

```
[taq@~]irb
irb(main):001:0> a = File.readlines("teste.txt")
=> ["Teste de leitura, primeira linha.\n", "Aqui \351 a linha 2.\n", "Estou na linha 3, acho qu
irb(main):002:0>
```

Você pode fazer alguns truques legais, por exemplo, pegar somente as linhas que começam com "Te" (lembram-se das expressões regulares?):

```
[taq@~]irb
irb(main):001:0> a = File.readlines("teste.txt").find_all {|linha| linha =~ /^Te/}
=> ["Teste de leitura, primeira linha.\n", "Terminei na linha 4, tchau. \n"]
irb(main):002:0>
```

Ou encurtando mais ainda, usando **grep**:

```
[taq@~]irb
irb(main):001:0> a = File.open("teste.txt") do |handle|
irb(main):002:1*    handle.grep(/^Te/)
irb(main):003:1> end
=> ["Teste de leitura, primeira linha.\n", "Terminei a linha 4, tchau.\n"]
```

Ah, e não podemos esquecer que podemos ler o arquivo byte-a-byte:

```
[taq@~]irb
irb(main):001:0> File.open("teste.txt") do |handle|
irb(main):002:1*    handle.each_byte do |b|
irb(main):003:2*       print "b:",b
irb(main):004:2>    end
irb(main):005:1> end
b:84b:101b:115b:116b:101b:32b:100b:101b:32b:108b:101b:105b:116b:117b:114b:97b:44b:32b:112
b:114b:105b:109b:101b:105b:114b:97b:32b:108b:105b:110b:104b:97b:46b:10b:65b:113b:117b:105
b:32b:233b:32b:97b:32b:108b:105b:110b:104b:97b:32b:50b:46b:10b:69b:115b:116b:111b:117b:32
b:110b:97b:32b:108b:105b:110b:104b:97b:32b:51b:44b:32b:97b:99b:104b:111b:32b:113b:117b:101
b:32b:118b:111b:117b:32b:116b:101b:114b:109b:105b:110b:97b:114b:46b:10b:84b:101b:114b:109
b:105b:110b:101b:105b:32b:110b:97b:32b:108b:105b:110b:104b:97b:32b:52b:44b:32b:116b:99b:104
b:97b:117b:46b:32b:10=> #<File:teste.txt (closed)>
irb(main):006:0>
```

Que bagunça! Isso por que o que lemos foi o código do caracter. Para lermos os valores em ASCII, use o método **chr** da Fixnum (que é a classe dos bytes que lemos):

```
[taq@~]irb
irb(main):001:0> File.open("teste.txt") do |handle|
irb(main):002:1*    handle.each_byte {|byte| print "c:#{byte.chr} "}
irb(main):003:1> end
c:T c:e c:s c:t c:e c:  c:d c:e c:  c:l c:e c:i c:t c:u c:r c:a c:, c:  c:p c:r c:i c:m c:e c:i
c:l c:i c:n c:h c:a c:. c:
 c:A c:q c:u c:i c:  c:é c:  c:a c:  c:l c:i c:n c:h c:a c:  c:2 c:. c:
 c:E c:s c:t c:o c:u c:  c:n c:a c:  c:l c:i c:n c:h c:a c:  c:3 c:, c:  c:a c:c c:h c:o c:  c:c
 c:t c:e c:r c:m c:i c:n c:a c:r c:. c:
```

```
 c:T c:e c:r c:m c:i c:n c:e c:i c:  c:n c:a c:  c:l c:i c:n c:h c:a c:  c:4 c:, c:  c:t c:c c:
 => #<File:teste.txt (closed)>
irb(main):004:0>
```

Para fazer o contrário, isso é, converter uma String para seu valor em ASCII, utilize:

```
[taq@~]irb
irb(main):001:0> s = "a"
=> "a"
irb(main):002:0> a = s[0]
=> 97
irb(main):003:0> a.chr
=> "a"
irb(main):004:0>
```

Escrevendo em arquivos

Para escrever em um arquivo, podemos usar **puts** ou **write**, a diferença é que puts insere uma quebra de linha, e write retorna a quantidade de caracteres escritos. Também podemos utilizar o operador << (que retorna o objeto de IO):

```
# teste de puts
handle = File.new("teste_puts.txt","w")
handle.puts "linha um"
handle.puts "linha dois"
handle.puts "linha três"
handle.close

File.readlines("teste_puts.txt").each {|linha| puts linha}

# teste de write
handle = File.new("teste_write.txt","w")
handle.write "linha um\n"
handle.write "linha dois\n"
handle.write "linha três\n"
handle.close

File.readlines("teste_write.txt").each {|linha| puts linha}

# teste de <<
handle = File.new("teste_write.txt","w")
handle <<  "linha um\n"
handle << "linha dois\n"
handle << "linha três\n"
handle.close

File.readlines("teste_write.txt").each {|linha| puts linha}
```

Resultado dos três exemplos acima:

```
linha um
linha dois
linha três
```

5.2.2 Fluxos

Vou dar ums exemplos rapidinhos aqui, Sockets são um assunto muito rico e vocês podem aprender mais consultando a API.

TCP

Vamos pegar um exemplo lendo de um servidor SMTP:

```
[taq@~]irb
irb(main):001:0> require 'socket'
=> true
irb(main):002:0> smtp = TCPSocket.open("localhost",25)
=> #<TCPSocket:0xb7ddf8a8>
irb(main):003:0> smtp.gets
=> "220 taq.server.com.br ESMTP Sendmail 8.13.1/8.12.11; Mon, 3 Jan 2005 18:11:53 -0200\r\n"
irb(main):004:0> smtp.puts("EHLO taq.com.br")
=> 16
irb(main):005:0> smtp.gets
=> "250-taq.server.com.br Hello localhost [127.0.0.1], pleased to meet you\r\n"
irb(main):006:0> smtp.close
=> nil
irb(main):007:0>
```

Está bem explícito ali: abra o socket, leia uma linha, envie uma linha (se você precisar enviar caracteres de controle antes do
n, use *send*), leia outra e feche o socket.

UDP

Vamos escrever um programinha que consulta o serviço **daytime**:

```
require 'socket'

udp = UDPSocket.open
udp.connect("ntp2.usno.navy.mil",13)
udp.send("\n",0)
resposta = udp.recvfrom(256)[0] # retorna como Array, pega o primeiro elemento
puts resposta[0,resposta.index("\r")] # limpa o "lixo"
```

Resultado:

```
[taq@~/code/ruby]ruby UDPTest.rb
Tue Jan 04 11:34:39 UTC 2005
```

Vamos fazer um exemplo agora que permite enviar e receber pacotes na nossa máquina local. Primeiro, o código do servidor, **UDPServer.rb**:

```
require 'socket'

porta = 12345
server = UDPSocket.new
server.bind("localhost",porta)
puts "Servidor conectado na porta #{porta}, aguardando ..."
loop do
   msg,sender = server.recvfrom(256)
   host = sender[3]
   puts "Host #{host} enviou um pacote UDP: #{msg}"
   break unless msg.chomp != "kill"
end
puts "Kill recebido, fechando servidor."
server.close
```

Agora o código do cliente, **UDPClient.rb**:

```
require 'socket'

porta = 12345
client = UDPSocket.open
client.connect("localhost",porta)
loop do
   puts "Digite sua mensagem (quit termina, kill finaliza servidor):"
   msg = gets
   client.send(msg,0)
   break unless !"kill,quit".include? msg.chomp
end
client.close
```

Vou rodar duas vezes o cliente:

```
[taq@~/code/ruby]ruby UDPClient.rb
Digite sua mensagem (quit termina, kill finaliza servidor):
Oi!
Digite sua mensagem (quit termina, kill finaliza servidor):
Tudo bem? Vou sair!
Digite sua mensagem (quit termina, kill finaliza servidor):
quit
[taq@~/code/ruby]ruby UDPClient.rb
Digite sua mensagem (quit termina, kill finaliza servidor):
Eu de novo!
Digite sua mensagem (quit termina, kill finaliza servidor):
Vou finalizar o servidor ...
Digite sua mensagem (quit termina, kill finaliza servidor):
kill
[taq@~/code/ruby]
```

Vamos dar uma olhada no servidor ...

```
[taq@~/code/ruby]ruby UDPServer.rb
Servidor conectado na porta 12345, aguardando ...
Host 127.0.0.1 enviou um pacote UDP: Oi!
Host 127.0.0.1 enviou um pacote UDP: Tudo bem? Vou sair!
Host 127.0.0.1 enviou um pacote UDP: quit
Host 127.0.0.1 enviou um pacote UDP: Eu de novo!
Host 127.0.0.1 enviou um pacote UDP: Vou finalizar o servidor ...
Host 127.0.0.1 enviou um pacote UDP: kill
Kill recebido, fechando servidor.
[taq@~/code/ruby]
```

HTTP

Vamos criar um fluxo HTTP e consultar o site do kernel do Linux, e ver quais versões estão disponíveis:

```
require 'net/http'

host = Net::HTTP.new("www.kernel.org",80)
resposta = host.get("/",nil)
print "Versões disponíveis do kernel do Linux\n(sem patches ou snapshots):\n\n"
if resposta.message != "OK"
   puts "Sem dados."
   return
end
resposta.body.scan(/[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}/).uniq.each {|k| puts "kernel #{k}"}
```

Abrimos o fluxo para o documento raiz / do site, verificamos a resposta, e filtramos os dados recebidos (o código HTML do site) através de uma expressão regular para determinar onde estão os números das versões disponíveis do kernel. Resultado (atualmente):

```
[taq@~/code/ruby]ruby http.rb
Versões disponíveis do kernel do Linux
(sem patches ou snapshots):

kernel 2.6.10
kernel 2.4.28
kernel 2.4.29
kernel 2.2.26
kernel 2.2.27
kernel 2.0.40
[taq@~/code/ruby]
```

Capítulo 6

Exceções

6.1 Begin ... rescue ... ensure ... end

Ruby, assim como Python, Java e PHP5, tem controle de exceções:

```
[taq@~/code/ruby]irb
irb(main):001:0> i1 = 1
=> 1
irb(main):002:0> i2 = "dois"
=> "dois"
irb(main):003:0> expr =
irb(main):004:0* begin
irb(main):005:1*    i1+i2
irb(main):006:1> rescue StandardError => exc
irb(main):007:1>    puts "Erro: #{exc}"
irb(main):008:1>    -1
irb(main):009:1> end
Erro: String can't be coerced into Fixnum
=> -1
irb(main):010:0> expr
=> -1
irb(main):011:0>
```

Nesse caso o bloco **begin ... end** é combinado com **rescue**, que será invocado caso alguma coisa de errado aconteça ali.

Reparem que depois de rescue eu informei o tipo de exceção que quero tratar. Usei a classe **StandardError** , mas tem várias outras mais especializadas (SyntaxError,NameError por exemplo) , e logo depois informei uma variável para ser armazenada a exceção gerada.

Reparem que tentei somar um Fixnum e uma String, deliberadamente, e foi gerada a exceção.

Também podemos omitir o tipo de exceção ali, que vai dar na mesma. Vou aproveitar e ver o tipo de exceção que foi gerada:

```
[taq@~/code/ruby]irb
irb(main):001:0> i1 = 1
=> 1
irb(main):002:0> i2 = "dois"
=> "dois"
irb(main):003:0> expr =
irb(main):004:0* begin
irb(main):005:1*    i1+i2
```

```
irb(main):006:1> rescue => exc
irb(main):007:1>    puts "Erro:#{exc} Tipo:#{exc.class}"
irb(main):008:1>    -1
irb(main):009:1> end
Erro:String can't be coerced into Fixnum Tipo:TypeError
=> -1
irb(main):010:0> expr
=> -1
irb(main):011:0>
```

Foi uma exceção do tipo **TypeError**.

Para mostrar como funciona o tratamento de várias exceções, vamos utilizar a vírgula e também um rescue para cada, gerando deliberadamente de novo outra exceção, agora do tipo NameError (vamos tentar usar uma variável que não existe):

```
[taq@~/code/ruby]irb
irb(main):001:0> i1 = 1
=> 1
irb(main):002:0> i2 = "dois"
=> "dois"
irb(main):003:0> expr =
irb(main):004:0* begin
irb(main):005:1*    i1+i3
irb(main):006:1> rescue NameError, TypeError => exc
irb(main):007:1>    puts "Erro: #{exc} Tipo:#{exc.class}"
irb(main):008:1> end
Erro: undefined local variable or method `i3' for main:Object Tipo:NameError
=> nil
irb(main):009:0>
```

Agora uma para cada:

```
[taq@~/code/ruby]irb
irb(main):001:0> i1 = 1
=> 1
irb(main):002:0> i2 = "dois"
=> "dois"
irb(main):003:0> expr =
irb(main):004:0* begin
irb(main):005:1*    i1+i3
irb(main):006:1> rescue NameError => exc
irb(main):007:1>    puts "NameError: #{exc}"
irb(main):008:1> rescue TypeError => exc
irb(main):009:1>    puts "TypeError: #{exc}"
irb(main):010:1> rescue StandardError => exc
irb(main):011:1>    puts "Erro genérico: #{exc}"
irb(main):012:1> end
NameError: undefined local variable or method `i3' for main:Object
=> nil
irb(main):013:0>
```

No segundo exemplo, o código foi direcionado para a exceção correta.

Vocês podem estar se perguntando que diabos é aquele **expr** lá. Aquela variável vai reter o valor retornado pelo bloco de código. No caso de ocorrer uma exceção, ele vai pegar a última expressão do rescue correspondente:

```
i1 = 1
```

```
i2 = "dois"

expr =
begin
   i1+i2
rescue NameError => exc
   puts "NameError: #{exc}"
   -1
rescue TypeError => exc
   puts "TypeError: #{exc}"
   -2
rescue StandardError => exc
   puts "Erro genérico: #{exc}"
   -3
end
puts expr
```

Como tive uma exceção do tipo TypeError ali, meu valor de retorno em expr foi -2.

Também temos um meio de fazer um bloco de código ser sempre executado executado depois dos rescues, usando **ensure**:

```
i1 = 1
i2 = "dois"

expr =
begin
   i1+i2
rescue NameError => exc
   puts "NameError: #{exc}"
   -1
rescue TypeError => exc
   puts "TypeError: #{exc}"
   -2
rescue StandardError => exc
   puts "Erro genérico: #{exc}"
   -3
ensure
   puts "Executando código ..."
   -4
end
puts "Expr:#{expr}"
```

Resultado:

```
TypeError: String can't be coerced into Fixnum
Executando código ...
Expr:-2
```

Reparem que o código foi executado (apareceu um "Executando código . . . " ali), e também no valor da expressão, que *foi gerada pelo último rescue*, e não pelo ensure.

Aí vocês podem argumentar "mas que coisa, para que que preciso desse ensure, eu poderia imprimir o 'Executando código . . . ' logo na sequência". Ah é? Então vamos testar, *fazendo dar "pau" dentro do rescue*:

```
i1 = 1
i2 = "dois"
```

```
expr =
begin
   i1+i2
rescue NameError => exc
   puts "NameError: #{exc}"
   -1
rescue TypeError => exc
   puts "TypeError: #{exc}"
   i1+i3
   -2
rescue StandardError => exc
   puts "Erro genérico: #{exc}"
   -3
ensure
   puts "Executando código ..."
   -4
end
puts "Expr:#{expr}"
```

Resultado:

```
exception.rb:12: undefined local variable or method ‘i3’ for main:Object (NameError)
TypeError: String can’t be coerced into Fixnum
Executando código ...
```

Viram? "Executando código ..." está lá, mas o final do programa (o puts final) não, pois a exceção gerada pulou para fora do escopo do programa que estava rodando ali. :-p

Uma coisa interessante do **rescue** é que ele pode ser utilizado em um statement para interceptar alguma exceção que ocorra e retornar um valor. Por exemplo:

```
[taq@~]irb
irb(main):001:0> i, s, x = 1, "dois", -1
=> [1, "dois", -1]
irb(main):002:0> x = i+s
TypeError: String can’t be coerced into Fixnum
        from (irb):2:in ‘+’
        from (irb):2
irb(main):003:0> x
=> -1
irb(main):004:0> x = i+s rescue nil
=> nil
irb(main):005:0> x
=> nil
```

Lendo o que aconteceu ali: se somando *i* e *s* der algum problema (se lembrem que Ruby tem tipagem forte, não vai permitir somar um Fixnum e uma String!), o valor que vai ser atribuído à *x* é o valor à direita do statement, **após** rescue, ou seja, *i+s*. Seria algo como *x é igual á i+s, se der problema, à nil*.

6.2 Throw . . . catch

Throw e catch podem ser bem úteis quando quisermos sair de códigos "aninhados", por exemplo:

```
[taq@~/code/ruby]irb
irb(main):001:0> def le_comando
irb(main):002:1>    print "Digite algo:"
irb(main):003:1>    texto = gets.chomp
irb(main):004:1>    throw :sair if texto == "sair"
```

```
irb(main):005:1>    return texto
irb(main):006:1> end
=> nil
irb(main):007:0> catch :sair do
irb(main):008:1*    while true
irb(main):009:2>       digitado = le_comando
irb(main):010:2>       puts "Foi digitado: #{digitado}"
irb(main):011:2>    end
irb(main):012:1> end
Digite algo:Oi!
Foi digitado: Oi!
Digite algo:Tudo beleza?
Foi digitado: Tudo beleza?
Digite algo:sair
=> nil
irb(main):013:0>
```

O que aconteceu ali foi:

- Definimos um método para ler uma resposta do usuário
- Pedimos para que, se a resposta for "sair", fosse executado um throw com o símbolo :sair
- Do contrário, retorna o texto
- Iniciamos um bloco catch, esperando um "sinal" de sair, usando o símbolo :sair, que foi utilizado no throw.
- Ficamos em loop até que seja digitado "sair", quando o método *le_comando* vai disparar o throw, que será interceptado pelo catch.

Capítulo 7

Módulos

Módulos são um jeito de "grudar" vários métodos, e não quer definir uma classe, ou quer fazer esses métodos disponíveis para um "mixin". Módulos também são úteis para criar namespaces e evitar conflito de nomes. Também são um meio de simular algo do tipo de herança múltipla, disponível em outras linguagens.

Usando nosso exemplo de automóveis, vamos imaginar que temos um módulo (esqueçam a classe agora) chamado **Automovel**, que tem um método chamado **ligar** nele. Ao mesmo tempo, **ligar** é um método também no nosso módulo **Celular**, que vamos utilizar de dentro do automóvel (não façam isso! é uma contravenção! :-p). Esses métodos nada mais fazem do que mostrar os barulhos relacionados (coisinha besta, mas serve para o exemplo aqui):

```
[taq@~]irb
irb(main):001:0> module Automovel
irb(main):002:1>    def Automovel.ligar
irb(main):003:2>       "Vrrummmm!"
irb(main):004:2>    end
irb(main):005:1> end
=> nil
irb(main):006:0> module Celular
irb(main):007:1>    def Celular.ligar
irb(main):008:2>       "Bip!"
irb(main):009:2>    end
irb(main):010:1> end
=> nil
irb(main):011:0> a = Automovel.ligar
=> "Vrrummmm!"
irb(main):012:0> c = Celular.ligar
=> "Bip!"
irb(main):013:0>
```

Esses métodos são chamados de **métodos de módulo**, ou até de **funções de módulo**.

No exemplo acima, não precisamos criar classes, e pudemos separar os dois métodos nos respectivos namespaces.
Podemos também separar os módulos em arquivos distintos, **automovel.rb** com a definição de Automovel e **Celular.rb** com a definição de celular, e carregá-los dessa maneira:

```
require "automovel"
require "celular"

a = Automovel.ligar
c = Celular.ligar
puts "Um automóvel faz #{a} quando é ligado"
```

```
puts "Um celular faz #{c} quando é ligado"
```

Resultado:

```
Um automóvel faz Vrrummmm! quando é ligado
Um celular faz Bip! quando é ligado
```

O comando **require** carrega e executa arquivos com código, e pode carregar também módulos de extensão binários.
Também faz consistência para verificar se o conteúdo do arquivo foi carregado apenas uma vez, ao contrário de **load**, que se executado recarrega o arquivo (e precisa de uma extensão .rb explícita).

Você pode utilizar também **module_function** para fazer de um método uma função (método) de módulo:

```
module Automovel
   def ligar
      "Vrrummmm!"
end
module_function :ligar
end
```

Módulos diferem de classes por que são coleções de métodos e constantes, não podendo criar instâncias, e podem ser "mixados" em classes e outros módulos.

Vamos supor que você tem um celular e um palmtop. A classe Celular já é derivada de outra classe Telefone, e você quer que ela tenha também um método legal que tem no módulo CoisasQueFazemBip, *bip*. Como Ruby tem herança única, você pode fazer:

```
class Telefone
   def disca
      "discando ..."
   end
   def atende
      "atendendo ..."
   end
end

module CoisasQueFazemBip
   def bip
      "Bip!"
   end
end

class Celular < Telefone
   include CoisasQueFazemBip
end

class Palmtop
include CoisasQueFazemBip
end

c = Celular::new
p = Palmtop::new

puts "celular: #{c.bip}"
puts "palmtop: #{c.bip}"
```

Resultado:

```
celular: Bip!
palmtop: Bip!
```

O comando **include** fez o *mixin* entre a classe Celular e o módulo CoisasQueFazemBip. Reparem como eu não declarei o método do módulo usando *módulo.método*, somente o nome, que foi mixado na classe Celular.

Se você quiser fazer um mixin no seu objeto depois de criado, pode utilizar **extend**:

```
[taq@~/code/ruby]irb
irb(main):001:0> class Teste
irb(main):002:1> end
=> nil
irb(main):003:0> t = Teste::new
=> #<Teste:0xb7de0ba4>
irb(main):004:0> require "CoisasQueFazemBip"
=> true
irb(main):005:0> t.extend CoisasQueFazemBip
=> #<Teste:0xb7de0ba4>
irb(main):006:0> t.bip
=> "Bip!"
irb(main):007:0>
```

Uma coisa legal é que você pode utilizar os módulos do Ruby em suas classes, aumentando o poder delas. Por exemplo, com apenas a definição de um método <=> na sua classe você pode ter o poder de processamento do módulo **Comparable**:

```
[taq@~]irb
irb(main):001:0> class Teste
irb(main):002:1>    attr_reader :valor
irb(main):003:1>    include Comparable
irb(main):004:1>    def initialize(v)
irb(main):005:2>       @valor = v
irb(main):006:2>    end
irb(main):007:1>    def <=>(outra)
irb(main):008:2>       self.valor <=> outra.valor
irb(main):009:2>    end
irb(main):010:1> end
=> nil
irb(main):011:0> t1 = Teste.new(1)
=> #<Teste:0xb7db6d5c @valor=1>
irb(main):012:0> t2 = Teste.new(2)
=> #<Teste:0xb7e3e630 @valor=2>
irb(main):013:0> t1 > t2
=> false
irb(main):014:0> t2 > t1
=> true
irb(main):015:0> t1 < t2
=> true
irb(main):016:0> t2 < t1
=> false
irb(main):017:0> t1 == t1
=> true
irb(main):018:0> t2 == t2
=> true
```

Prático! Ganhamos todas as comparações apenas adicionando o módulo.

Outro exemplo é o módulo **Enumerable**. É só definir um método **each**:

```
[taq@~]irb
irb(main):001:0> class Teste
irb(main):002:1>    include Enumerable
irb(main):003:1>    def initialize(a)
irb(main):004:2>       @array = a
irb(main):005:2>    end
irb(main):006:1>    def each
irb(main):007:2>       @array.each do |v|
irb(main):008:3*          yield v
irb(main):009:3>       end
irb(main):010:2>    end
irb(main):011:1> end
=> nil
irb(main):012:0> t = Teste.new([3,5,1,2,4])
=> #<Teste:0xb7e4abd8 @array=[3, 5, 1, 2, 4]>
irb(main):013:0> t.sort
=> [1, 2, 3, 4, 5]
irb(main):014:0> t.include?(3)
=> true
irb(main):015:0> t.include?(10)
=> false
irb(main):016:0> t.min
=> 1
irb(main):017:0> t.max
=> 5
irb(main):018:0> t.sort.map {|v| v*10}
=> [10, 20, 30, 40, 50]
```

Capítulo 8

Threads

Threads em Ruby são implementadas pelo interpretador, mantendo-as portáveis entre os sistemas operacionais onde Ruby roda, inclusive em alguns que não tem multithreading, como o DOS, mas ao mesmo tempo eximindo os benefícios que teríamos com threads nativas do sistema operacional.

Vamos ver um exemplo simples de thread:

```
t = Thread.new do
   puts "thread inicializada ..."
   5.times do |i|
      puts "loop #{i}"
   end
end
t.join
puts "thread finalizada."
```

Criamos uma thread nova através de **Thread.new**, *passando um bloco de código como parâmetro.* Dentro dessa thread, iniciamos uma contagem de 0 à 4, mostrando o número corrente, e *pedimos para a thread terminar* usando **t.join**.

Nesse exemplo específico, não o método *join* não vai fazer muita diferença ali, por que o processamento dentro da thread vai ser tão rápido que nem precisaríamos esperar o seu término. Vamos fazer um teste: remova o *t.join* de lá e rode o programa. Mesma coisa, certo? Agora, troque o *5* por *5000* e rode novamente.

Dependendo da velocidade do seu computador (se ele não for muito rápido!) você irá notar que a contagem parou bem antes de 5000. Aqui no meu Pentium III - aceito doações :-) - parou entre 500 e 1100. Agora mantenha o 5000 lá (se no último exemplo você conseguiu ver toda a contagem, aumente para uns 20000) e retorne o *t.join*. Você vai ver todos os números da contagem lá agora.

Então não se esqueçam: *join* faz o programa esperar a thread acabar de fazer o seu processamento.

Nesse programa mostramos duas coisas:

- Usando *join aparentemente* ali temos uma "ilusão" de que o processamento é linear, oras, um loop regular ali faria o mesmo efeito, não faria? Usaríamos um em 5000.times —n— puts n e *aparentemente* seria a mesma coisa.
- Mas ao mesmo tempo, *retirando o join e aumentando o limite*, pudemos ver que a thread está rodando em paralelo com o processamento principal, pois o programa prosseguiu (e foi finalizado) enquanto o processamento da thread estava ativo ainda.

Vamos dar mais tempo para a thread mostrar que está ativa, e que roda em paralelo *mesmo*:

```
t = Thread.new do
```

```
   puts "thread inicializada ..."
   5000.times do |i|
      puts "loop #{i}"
   end
end
puts "dormindo 2 segundos ..."
sleep(2)
```

Pedimos para a thread ser criada, e que o processamento principal esperasse 2 segundos antes de terminar.

É interessante você rodar esse programa, se estiver no Linux, usando um pipe com *less*, para poder ver todas as linhas impressas. O que acontece é que após alguns milhares de números (a thread está rodando!), veremos a mensagem "dormindo 2 segundos ..." e pedimos para o processamento principal aguardar 2 segundos, enquanto vemos os números da thread ainda sendo impressos. Após 2 segundos de espera, o programa termina. Pelo menos aqui deu tempo de imprimir todos os 5000 números. :-) Agora ficou bem claro o processamento em paralelo da thread.

8.1 Timeout

Alterando um pouco o nosso último programa e misturando-o com o anterior, vamos utilizar um tempo de espera direto no *join*:

```
t = Thread.new do
   puts "thread inicializada ..."
   10.times do |i|
      puts "loop #{i}"
      sleep(1)
   end
end
puts "dormindo um pouco ..."
t.join(5)
puts "terminando."
```

Resultado:

```
thread inicializada ...
loop 0
dormindo um pouco ...
loop 1
loop 2
loop 3
loop 4
terminando.
```

Vemos aqui que pedimos na thread um loop de 10 vezes, *pedindo para esperar um segundo a cada vez, usando sleep*, e pedimos para o processamento principal esperar a thread terminar em até 5 segundos. Oras, 10 vezes 1 segundo vão dar 10 segundos, então, antes do processamento total da thread terminar, teremos finalizado o programa (timeout).

8.2 Criando usando Proc

Podemos criar Proc's para usarmos para criarmos nossas threads depois. Por exemplo:

```
# define a Proc aqui
p = Proc.new do |n|
   t = 2.0/n
   sleep(t)
```

```
   printf "thread %d, aguardou %.2f segundos\n",n,t
end

# criando as threads aqui
t1 = Thread.new(1,&p)
t2 = Thread.new(2,&p)
t3 = Thread.new(3,&p)
t4 = Thread.new(4,&p)
t5 = Thread.new(5,&p)
puts "aguardando thread 1 ..."
t1.join
puts "terminei."
```

Uma coisa interessante é como enviamos os parâmetros para a thread. A thread executa o bloco usando os parâmetros que passamos (os números de 1 à 5).

8.3 Sincronização

Vamos fazer um pequeno teste aqui: vamos ter duas variáveis, *mais* e *menos*, e dentro de um loop em uma thread, vou *incrementar mais* e *decrementar menos* em 1, enquanto outra thread computa a diferença entre as duas variáveis e insere a diferença em um Array. Vamos esperar dois segundos e imprimir todos os valores *únicos* (sem repetição), computados nesse Array. Mas espera aí! Valores únicos? Sem repetição? Mas não tem que ser sempre 0, se são incrementadas e decrementadas ao mesmo tempo?

Bom, não precisamente ... vamos ver esse exemplo:

```
require "thread"

mais = menos = 0
log  = []

t1 = Thread.new do
   loop do
      mais  += 1
      menos -= 1
   end
end

t2 = Thread.new do
   loop do
      log << mais + menos
   end
end

sleep 2
puts log.uniq
puts "maior:#{mais},menor:#{menos}"
```

Rodando o programa, temos (os seus valores podem ser diferentes):

```
1
0
-14504
-15022
-15025
-14747
-14757
```

```
-14696
-14505
-14764
-14790
-15015
maior:1391416,menor:-1391415
```

O que acontece é que não há *sincronização* entre as duas threads, então a segunda thread pode estar pegando o valor de alguma das variáveis ali fora de sincronia com a outra. Não temos "*atomicidade*". Mesmo ali nas variáveis *mais* e *menos* no final do programa temos uma discrepância. Por isso que temos essa grande divergência em nosso Array.

Para podermos ter essa sincronia, precisamos de um **Mutex**, que permite acesso exclusivo aos recursos "travados" por ele. Ou seja, apenas uma thread de cada vez pode acessar os recursos, enquanto as outras podem esperar na fila ou informarem um erro notificando que o recurso está alocado. Vamos alterar nosso programa para usar um Mutex:

```
require "thread"

mais = menos = 0
log  = []
lock = Mutex.new # criado o Mutex

t1 = Thread.new do
   loop do
      lock.synchronize do
         mais  += 1
         menos -= 1
      end
   end
end

t2 = Thread.new do
   loop do
      lock.synchronize do
         log << mais + menos
      end
   end
end

sleep 2
puts log.uniq
puts "maior:#{mais},menor:#{menos}"
```

Resultado:

```
0
maior:159992,menor:-159992
```

Agora sim. Tivemos a sincronização, feita através dos blocos *Mutex.synchronize do ... end*. O lock é adquirido, as variáveis são incrementadas/decrementadas, o lock é liberado, é adquirido novamente e é computado a diferença, que nos retorna sempre 0, e as variáveis são exatamente iguais (levando em conta que uma é positiva e a outra é negativa) no final do programa.

Mas ... nem tudo são flores. O preço a pagar para a sincronização é que o processamento fica mais demorado. Deêm uma olhada nos valores finais das variáveis: enquanto *sem* sincronização os valores batiam na casa de mais de 1 milhão, *com* sincronização os valores não passam de 200 mil.

Vamos fazer outro exemplo aqui usando uma exercício do conceito de *produtor/consumidor*. Teremos uma thread que produz alguns itens em quantidade randômica (alguns Fixnum's), insere em um Array e espera 1 segundo. Enquanto isso, outra thread roda, removendo os itens inseridos. Ambas usam blocos sincronizados para o acesso ao Array de itens:

```
require "thread"

items = []
lock = Mutex.new
count = 0

t1 = Thread.new do
   loop do
      lock.synchronize do
         (0..rand(5)).each do
            print "produzindo (+) #{count}\n"
            items << count
            count += 1
         end
         print "\n"
end
      sleep(1)
   end
end

t2 = Thread.new do
   loop do
      lock.synchronize do
         size = items.size
         for i in (0...size)
            print "consumindo (-) #{items[0]}\n"
            items.shift
         end
         if size > 0
            print "-----------------\n"
         end
      end
   end
end

sleep 5
puts "quantidade de itens disponíveis para consumo: #{ite\mathcal{N}os.size}"
```

Resultado:

```
produzindo (+) 0

consumindo (-) 0
-----------------
produzindo (+) 1

consumindo (-) 1
-----------------
produzindo (+) 2
produzindo (+) 3

consumindo (-) 2
consumindo (-) 3
```

```
-----------------
produzindo (+) 4

consumindo (-) 4
-----------------
produzindo (+) 5
produzindo (+) 6
produzindo (+) 7

consumindo (-) 5
consumindo (-) 6
consumindo (-) 7
-----------------
quantidade de itens disponíveis para consumo: 0
```

ATENÇÃO! Você pode estar sentindo falta de um *iterator* ali no loop onde removemos os elementos do Array, mas se usarmos um *items.each* e logo depois um *items.shift*, vamos *alterar os elementos do Array, alterando o iterator*. Nesses casos, prefira o método sugerido.

Funcionou direitinho, em grande parte pelo fato do *sleep* estar fora do bloco do *synchronize*, dando chance para a thread consumidora adquirir o lock e fazer o seu trabalho. Mas e se quiséssemos um jeito de avisar a thread consumidora sempre que tiverem itens disponíveis para consumo, sem necessariamente esperar um tempo pré-determinado na thread produtora?

8.4 Condition variables

Condition variables são associadas a um Mutex e utilizadas para esperar ou indicar que um recurso está ocupado ou liberado.
Os métodos principais de uma condition variable são:

- **wait(mutex)** Libera o lock no Mutex correspondente e aguarda um sinal da condition variable para adquirir o lock novamente e prosseguir. Nesse ponto a thread é considerada como "em dormência".
- **signal** "Acorda" a primeira thread esperando por esse sinal da condition variable, fazendo-a readquirir o lock e prosseguir.

Para mostrar isso, alterei o nosso programa anterior para executar apenas 5 loops (você pode remover essa limitação para ver o processamento) e ao invés de esperar um determinado tempo, usei *join* na primeira thread para esperarmos ela finalizar seu trabalho:

```
require "thread"

items = []
lock = Mutex.new
cond = ConditionVariable.new # criada a condition variable
count = 0

puts "inicializando ..."
t1 = Thread.new do
   loops = 1
   loop do
      puts "produtor: iniciando produção (loop #{loops})."
      lock.synchronize do
         (0..rand(5)).each do
            puts "produtor: produzindo (+) #{count}"
            items.push(count)
            count += 1
         end
```

```
            puts "produtor: aguardando consumo ..."
            # libera lock no mutex e espera - vai readquirir o lock no wakeup (signal)
            cond.wait(lock)
            puts "produtor: consumo efetuado."
            puts "------------------------------------------------------------"
            loops += 1
         end
         break unless loops <= 5
      end
   end

   t2 = Thread.new do
      puts "consumidor: aguardando para consumir ..."
      loop do
         puts "consumidor: consumindo itens produzidos ..."
         lock.synchronize do
            size = items.size
            for i in (0...size)
               puts "consumidor: consumindo (-) #{items[0]}"
               items.shift
            end
            puts "consumidor: consumidos todos os itens, aguardando produção ..."
            cond.signal # acorda thread, que reassume o lock
         end
      end
   end

   t1.join
   puts "quantidade de itens disponíveis para consumo: #{items.size}"
```

Resultado:

```
inicializando ...
produtor: iniciando produção (loop 1).
produtor: produzindo (+) 0
produtor: aguardando consumo ...
consumidor: aguardando para consumir ...
consumidor: consumindo itens produzidos ...
consumidor: consumindo (-) 0
consumidor: consumidos todos os itens, aguardando produção ...
produtor: consumo efetuado.
------------------------------------------------------------
produtor: iniciando produção (loop 2).
produtor: produzindo (+) 1
produtor: produzindo (+) 2
produtor: aguardando consumo ...
consumidor: consumindo itens produzidos ...
consumidor: consumindo (-) 1
consumidor: consumindo (-) 2
consumidor: consumidos todos os itens, aguardando produção ...
produtor: consumo efetuado.
------------------------------------------------------------
produtor: iniciando produção (loop 3).
produtor: produzindo (+) 3
produtor: aguardando consumo ...
consumidor: consumindo itens produzidos ...
consumidor: consumindo (-) 3
```

```
consumidor: consumidos todos os itens, aguardando produção ...
produtor: consumo efetuado.
------------------------------------------------------------
produtor: iniciando produção (loop 4).
produtor: produzindo (+) 4
produtor: produzindo (+) 5
produtor: produzindo (+) 6
produtor: produzindo (+) 7
produtor: produzindo (+) 8
produtor: aguardando consumo ...
consumidor: consumindo itens produzidos ...
consumidor: consumindo (-) 4
consumidor: consumindo (-) 5
consumidor: consumindo (-) 6
consumidor: consumindo (-) 7
consumidor: consumindo (-) 8
consumidor: consumidos todos os itens, aguardando produção ...
produtor: consumo efetuado.
------------------------------------------------------------
produtor: iniciando produção (loop 5).
produtor: produzindo (+) 9
produtor: produzindo (+) 10
produtor: produzindo (+) 11
produtor: produzindo (+) 12
produtor: produzindo (+) 13
produtor: aguardando consumo ...
consumidor: consumindo itens produzidos ...
consumidor: consumindo (-) 9
consumidor: consumindo (-) 10
consumidor: consumindo (-) 11
consumidor: consumindo (-) 12
consumidor: consumindo (-) 13
consumidor: consumidos todos os itens, aguardando produção ...
produtor: consumo efetuado.
------------------------------------------------------------
quantidade de itens disponíveis para consumo: 0
```

Como vimos, agora a thread de consumo é disparada sempre que algum item é produzido.

Capítulo 9

Acessando banco de dados

Ruby vem com interfaces para acesso à banco de dados similares ao modo que são acessados no Perl. Para saber mais sobre a **Ruby-DBI**, acesse:
http://ruby-dbi.sourceforge.net/

A vantagem de uma interface uniforme é que você pode portar seus programas entre diversos banco de dados apenas alterando os parâmetros da chamada inicial de abertura da conexão. A Ruby-DBI *abstrai* os diferentes métodos de acesso de cada banco e permite você "rodar macio" por entre os vários bancos que existem por aí.

Vou mostrar os exemplos usando o MySQL(http://www.mysql.com/), que é um banco de uso mais comum (apesar que eu acesso o Oracle aqui da mesma forma).

> **ATENÇÃO!** Não, eu disse *não* se esqueça de efetuar o download do módulo do MySQL em http://www.tmtm.org/en/mysql/ruby/, configurá-lo e instalá-lo (as instruções vem junto com os arquivos). Do contrário os exemplos não funcionarão.

9.1 Abrindo a conexão

Vamos abrir e fechar a conexão com o banco:

```
[taq@~]irb
irb(main):001:0> require "dbi"
=> true
irb(main):002:0> con = DBI.connect("DBI:Mysql:taq:localhost","taq","******")
=> #<DBI::DatabaseHandle:0xb7c6e290 @trace_output=#<IO:0xb7e5d064>, @trace_mode=2,
@handle=#<DBI::DBD::Mysql::Database:0xb7c6e038 @have_transactions=true, @attr={"AutoCommit"=>tru
@mutex=#<Mutex:0xb7c6dee4 @locked=false, @waiting=[]>, @handle=#<Mysql:0xb7c6e04c>>>
irb(main):003:0> con.disconnect
=> nil
irb(main):004:0>
```

No método **connect**, no primeiro parâmetro, especifiquei:

- O driver do banco de dados a ser utilizado (com DBI: antes). No nosso caso, o Mysql.
- O database a ser utilizado (taq)

Logo depois especifiquei o **nome de usuário** e a **senha**, o que me retornou uma conexão válida, que fechei usando o método **disconnect**.

Só para dar uma idéia, o que eu precisaria para conectar do mesmo jeito em um banco de dados Oracle, seria instalar o driver Ruby para ele e trocar a linha do connect para

```
con = DBI.connect("DBI:OCI8:banco_oracle","usuario","******")
```

Prático não?

> **ATENÇÃO!** Em todo lugar onde estiver escrito **"******"** aqui nesse capítulo, entendam como a senha de acesso. Eu estava usando umas strings e senhas malucas aqui, mas para padronizar vou usar a sequência de * que fica mais bonito. :-)

9.2 Executando queries que não retornam resultado

Vamos criar uma tabela nova e inserir alguns dados nela, usando o método **do**:

```
require "dbi"

con = DBI.connect("DBI:Mysql:taq:localhost","taq","******")
con.do("drop table if exists pedidos")

# cria tabela
sql = <<FIMSQL
create table pedidos
(numero    int(6) not null,
 cliente   int(6) not null,
 descricao varchar(50) not null,
 valor     decimal(10,2) not null)
FIMSQL
con.do(sql)

# insere alguns valores
con.do("insert into pedidos values (1,1,'CD - Anthrax - Greater of Two Evils',25.00)")
con.do("insert into pedidos values (2,1,'DVD - Slayer - Still Reigning',55.00)")
con.do("insert into pedidos values (3,2,'CD - Judas Priest - Angel of Retribution',30.00)")

con.disconnect
```

Reparem que eu usei o **heredoc** para construir o comando SQL mais longo.

Fica a dica que **do** retorna a quantidade de linhas afetadas pelo comando.

9.3 Recuperando dados do banco de dados

Para consultamos dados no banco, usamos **execute**, que retorna um **result set**, que, com **fetch** (que nesse caso funciona igual um *each*) pode ser usado como um **iterator**:

```
require "dbi"

con = DBI.connect("DBI:Mysql:taq:localhost","taq","******")
con.do("drop table if exists pedidos")

# cria tabela
sql = <<FIMSQL
create table pedidos
(numero    int(6) not null,
 cliente   int(6) not null,
 descricao varchar(50) not null,
 valor     decimal(10,2) not null)
FIMSQL
con.do(sql)
```

```
# insere alguns valores
con.do("insert into pedidos values (1,1,'CD - Anthrax - Greater of Two Evils',25.00)")
con.do("insert into pedidos values (2,1,'DVD - Slayer - Still Reigning',55.00)")
con.do("insert into pedidos values (3,2,'CD - Judas Priest - Angel of Retribution',30.00)")

# consulta todos os registros
rst = con.execute("select * from pedidos")
rst.fetch do |row|
printf "pedido:%d cliente:%d produto:%s valor:%.2f\n",
row["numero"],row["cliente"],row["descricao"],row["valor"]
end
rst.finish

con.disconnect
```

Resultado:

```
pedido:1 cliente:1 produto:CD - Anthrax - Greater of Two Evils valor:25.00
pedido:2 cliente:1 produto:DVD - Slayer - Still Reigning valor:55.00
pedido:3 cliente:2 produto:CD - Judas Priest - Angel of Retribution valor:30.00
```

Duas coisas a reparar aqui: usei **printf**, que é usado igual se usa em C, e no final do loop do fetch, usei o método **finish**, que é usado para liberar o recurso alocado pelo seu result set. Se você quer ser amigo do seu banco e dos seus recursos, feche eles quando terminar de usá-los. :-)

9.4 Preparando comandos e usando parametros variáveis

Se você utiliza um comando SQL várias vezes no seu programa, onde as únicas coisas que se alteram são os parâmetros do comando, fica melhor se você criar um **prepared statement** uma única vez e alterar apenas os parâmetros enviados ao comando.

Isso pode ser feito utilizando o método **prepare**, onde você especifica onde os parâmetros variáveis vão ficar usando o caracter **?**, e envia os valores a serem substituídos através do método **execute**:

```
require "dbi"

con = DBI.connect("DBI:Mysql:taq:localhost","taq","******")
con.do("drop table if exists pedidos")

# cria tabela
sql = <<FIMSQL
create table pedidos
(numero    int(6) not null,
 cliente   int(6) not null,
 descricao varchar(50) not null,
 valor     decimal(10,2) not null)
FIMSQL
con.do(sql)

# insere alguns valores
con.do("insert into pedidos values (1,1,'CD - Anthrax - Greater of Two Evils',25.00)")
con.do("insert into pedidos values (2,1,'DVD - Slayer - Still Reigning',55.00)")
con.do("insert into pedidos values (3,2,'CD - Judas Priest - Angel of Retribution',30.00)")

# consulta todos os registros
pre = con.prepare("select * from pedidos where numero=?") # prepara a consulta aqui
```

```
for i in (1..10)
   printf "Procurando e listando pedido número %d\n",i
   pre.execute(i)
   pre.fetch do |row|
      printf "Encontrado: pedido:%d cliente:%d produto:%s valor:%.2f\n\n",
      row["numero"],row["cliente"],row["descricao"],row["valor"]
   end
end
pre.finish

con.disconnect
```

Resultado:

```
Procurando e listando pedido número 1
Encontrado: pedido:1 cliente:1 produto:CD - Anthrax - Greater of Two Evils valor:25.00

Procurando e listando pedido número 2
Encontrado: pedido:2 cliente:1 produto:DVD - Slayer - Still Reigning valor:55.00

Procurando e listando pedido número 3
Encontrado: pedido:3 cliente:2 produto:CD - Judas Priest - Angel of Retribution valor:30.00

Procurando e listando pedido número 4
Procurando e listando pedido número 5
Procurando e listando pedido número 6
Procurando e listando pedido número 7
Procurando e listando pedido número 8
Procurando e listando pedido número 9
Procurando e listando pedido número 10
```

Vou alterar agora a inserção inicial de dados para utilizar um prepared statement, ao mesmo tempo que vou utilizar um Array de Arrays para passar os parâmetros esperados pelo insert, usando * para expandir o Array para a lista de parâmetros esperadas por execute:

```
require "dbi"

con = DBI.connect("DBI:Mysql:taq:localhost","taq","******")
con.do("drop table if exists pedidos")

# cria tabela
sql = <<FIMSQL
create table pedidos
(numero    int(6) not null,
 cliente   int(6) not null,
 descricao varchar(50) not null,
 valor     decimal(10,2) not null)
FIMSQL
con.do(sql)

# insere alguns valores
ins = con.prepare("insert into pedidos values (?,?,?,?)")
[[1,1,'CD - Anthrax - Greater of Two Evils',25.00],
 [2,1,'DVD - Slayer - Still Reigning',55.00],
 [3,2,'CD - Judas Priest - Angel of Retribution',30.00]].each {|v| ins.execute(*v)}
ins.finish

# consulta todos os registros
```

```
pre = con.prepare("select * from pedidos where numero=?") # prepara a consulta aqui
for i in (1..10)
   printf "Procurando e listando pedido número %d\n",i
   pre.execute(i)
   pre.fetch do |row|
      printf "Encontrado: pedido:%d cliente:%d produto:%s valor:%.2f\n\n",
      row["numero"],row["cliente"],row["descricao"],row["valor"]
   end
end
pre.finish

con.disconnect
```

O resultado será similar ao apresentado no exemplo anterior.

9.5 Metadados

Você pode acessar os metadados da sua consulta logo após chamar o método execute:

```
require "dbi"

con = DBI.connect("DBI:Mysql:taq:localhost","taq","******")
con.do("drop table if exists pedidos")

# cria tabela
sql = <<FIMSQL
create table pedidos
(numero    int(6) not null,
 cliente   int(6) not null,
 descricao varchar(50) not null,
 valor     decimal(10,2) not null)
FIMSQL
con.do(sql)

# insere alguns valores
ins = con.prepare("insert into pedidos values (?,?,?,?)")
[[1,1,'CD - Anthrax - Greater of Two Evils',25.00],
 [2,1,'DVD - Slayer - Still Reigning',55.00],
 [3,2,'CD - Judas Priest - Angel of Retribution',30.00]].each {|v| ins.execute(*v)}
ins.finish

# consulta todos os registros
rst = con.execute("select * from pedidos")

# pega os metadados
puts "colunas da consulta:"
rst.column_info.each {|c| puts "nome:#{c['name']} precisao:#{c['precision']} escala:#{c['scale']
rst.finish

con.disconnect
```

Resultado:

```
colunas da consulta:
nome:numero precisao:6 escala:0
nome:cliente precisao:6 escala:0
nome:descricao precisao:50 escala:0
nome:valor precisao:10 escala:2
```

Usando um prepared statement, que resulta nos mesmos valores mostrados acima:

```
require "dbi"

con = DBI.connect("DBI:Mysql:taq:localhost","taq","******")
con.do("drop table if exists pedidos")

# cria tabela
sql = <<FIMSQL
create table pedidos
(numero    int(6) not null,
 cliente   int(6) not null,
 descricao varchar(50) not null,
 valor     decimal(10,2) not null)
FIMSQL
con.do(sql)

# insere alguns valores
ins = con.prepare("insert into pedidos values (?,?,?,?)")
[[1,1,'CD - Anthrax - Greater of Two Evils',25.00],
 [2,1,'DVD - Slayer - Still Reigning',55.00],
 [3,2,'CD - Judas Priest - Angel of Retribution',30.00]].each {|v| ins.execute(*v)}
ins.finish

# consulta todos os registros
pre = con.prepare("select * from pedidos")
pre.execute

# pega os metadados
puts "colunas da consulta:"
pre.column_info.each {|c| puts "nome:#{c['name']} precisao:#{c['precision']} escala:#{c['scale']
pre.finish

con.disconnect
```

Nesse caso os valores são disponibilizados após a chamada de **execute**.

9.6 Trabalhando com blocos

Alguns comandos permitem trabalhar com blocos, o que nos dá a conveniência de não nos preocuparmos em liberar seus recursos no final (lembram o que eu disse sobre isso no começo do capítulo?).

Vamos ver nosso exemplo, modificado para isso:

```
require "dbi"

DBI.connect("DBI:Mysql:taq:localhost","taq","******") do |con|
   con.do("drop table if exists pedidos")

   # cria tabela
   sql = <<FIMSQL
create table pedidos
(numero    int(6) not null,
 cliente   int(6) not null,
 descricao varchar(50) not null,
 valor     decimal(10,2) not null)
FIMSQL
```

```
   con.do(sql)

   # insere alguns valores
   con.prepare("insert into pedidos values (?,?,?,?)") do |ins|
      [[1,1,'CD - Anthrax - Greater of Two Evils',25.00],
      [2,1,'DVD - Slayer - Still Reigning',55.00],
      [3,2,'CD - Judas Priest - Angel of Retribution',30.00]].each {|v| ins.execute(*v)}
   end

   # consulta todos os registros
   con.execute("select * from pedidos") do |rst|
      rst.fetch do |row|
         printf "pedido:%d cliente:%d produto:%s valor:%.2f\n",
         row["numero"],row["cliente"],row["descricao"],row["valor"]
      end
   end
end
```

Pronto! Os métodos *connect, prepare* e *execute* já se viram sozinhos, fechando seus recursos quando terminam (end). Fica a dica que, dentro do bloco do *prepare*, você precisa chamar *execute*.

9.7 Output especializado

Temos alguns métodos em DBI::Utils que podem facilitar nossa visualização dos dados.

9.7.1 Tabular

Mostra os resultados como uma tabela:

```
require "dbi"

DBI.connect("DBI:Mysql:taq:localhost","taq","*******") do |con|
   con.do("drop table if exists pedidos")

   # cria tabela
   sql = <<FIMSQL
create table pedidos
(numero    int(6) not null,
 cliente   int(6) not null,
 descricao varchar(50) not null,
 valor     decimal(10,2) not null)
FIMSQL
   con.do(sql)

# insere alguns valores
con.prepare("insert into pedidos values (?,?,?,?)") do |ins|
      [[1,1,'CD - Anthrax - Greater of Two Evils',25.00],
      [2,1,'DVD - Slayer - Still Reigning',55.00],
      [3,2,'CD - Judas Priest - Angel of Retribution',30.00]].each {|v| ins.execute(*v)}
end

# consulta todos os registros
   rows = cols = nil
   con.execute("select * from pedidos") do |rst|
      rows = rst.fetch_all # pega todas as linhas
      cols = rst.column_names
```

```
   end
   DBI::Utils::TableFormatter.ascii(cols,rows)
end
```

Resultado:

```
+--------+---------+----------------------------------------+-------+
| numero | cliente | descricao                              | valor |
+--------+---------+----------------------------------------+-------+
| 1      | 1       | CD - Anthrax - Greater of Two Evils    | 25.00 |
| 2      | 1       | DVD - Slayer - Still Reigning          | 55.00 |
| 3      | 2       | CD - Judas Priest - Angel of Retribution | 30.00 |
+--------+---------+----------------------------------------+-------+
```

9.7.2 XML

Temos também a representação dos dados em XML:

```
require "dbi"

DBI.connect("DBI:Mysql:taq:localhost","taq","******") do |con|
   con.do("drop table if exists pedidos")

   # cria tabela
   sql = <<FIMSQL
create table pedidos
(numero    int(6) not null,
 cliente   int(6) not null,
 descricao varchar(50) not null,
 valor     decimal(10,2) not null)
FIMSQL
   con.do(sql)

   # insere alguns valores
   con.prepare("insert into pedidos values (?,?,?,?)") do |ins|
      [[1,1,'CD - Anthrax - Greater of Two Evils',25.00],
      [2,1,'DVD - Slayer - Still Reigning',55.00],
      [3,2,'CD - Judas Priest - Angel of Retribution',30.00]].each {|v| ins.execute(*v)}
   end

   # consulta todos os registros
   DBI::Utils::XMLFormatter.table(con.select_all("select * from pedidos"))
end
```

Resultado:

```
<?xml version="1.0" encoding="UTF-8" ?>
<rows>
<row>
  <numero>1</numero>
  <cliente>1</cliente>
  <descricao>CD - Anthrax - Greater of Two Evils</descricao>
  <valor>25.00</valor>
</row>
<row>
  <numero>2</numero>
  <cliente>1</cliente>
  <descricao>DVD - Slayer - Still Reigning</descricao>
```

```
  <valor>55.00</valor>
</row>
<row>
  <numero>3</numero>
  <cliente>2</cliente>
  <descricao>CD - Judas Priest - Angel of Retribution</descricao>
  <valor>30.00</valor>
</row>
</rows>
```

Podemos junto com o método table, especificar logo após a consulta SQL como serão chamados os nós de raiz e de cada linha do resultado:

```
DBI::Utils::XMLFormatter.table(con.select_all("select * from pedidos"),"raiz","linha")
```

Capítulo 10

XML

O meio de acessar XML que já vem junto com o pacote padrão do Ruby é através do processador REXML (http://www.germane-software.com/software/rexml/). A única coisa que você precisa fazer para o utilizar é um

```
require ‘‘rexml/document’’
```

no seu código.

10.1 Lendo arquivos XML

Antes de mais nada, vamos criar um arquivo XML para teste, **teste.xml**:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<montadoras>
   <tipo>Listagem</tipo>
   <montadora nome="Volkswagen">
   <localizacao>SP</localizacao>
      <carro modelo="Gol" cor="azul"/>
      <carro modelo="Golf" cor="preto"/>
   </montadora>
   <montadora nome="Ford">
      <localizacao>PR</localizacao>
      <carro modelo="Ka" cor="vermelho"/>
   </montadora>
</montadoras>
```

Agora vamos ler o arquivo de três modos, que nos dão resultados similares:

```
require "rexml/document"

puts "Listando arquivo XML"
xml = REXML::Document.new File.open("teste.xml")
xml.elements.each("montadoras/montadora") do |m|
puts "A montadora #{m.attributes['nome']} (#{m.elements['localizacao'].text}) produziu os seguin
m.elements.each("carro") {|c| puts "#{c.attributes['modelo']} #{c.attributes['cor']}"}
end

# usando XPath
print "\nUsando XPath para descobrir o primeiro carro e as montadoras do arquivo\n"
primeiro = REXML::XPath.first(xml,"//carro")
puts "O primeiro carro da lista é um #{primeiro.attributes['modelo']} #{primeiro.attributes['co
REXML::XPath.each(xml,"//montadora") {|m| puts "Achei a montadora #{m.attributes["nome"]}"}
```

```
# usando Arrays
print "\nListando carros através de um Array\n"
xml.elements.to_a("//carro").each {|e| puts "#{e.attributes['modelo']} #{e.attributes['cor']}"}
```

- O primeiro modo é o "convencional" do REXML, acessando os elementos através de iterators, que são retornados através da abertura do arquivo com o **REXML::Document.new File.open**.
- O segundo é feito através do parser XPath, que além dos métodos **first** e **each** apresentados ali, também tem **match**.
- O terceiro é feito usando a conversão para Array.

10.2 Criando arquivos XML

Vamos criar agora um segundo arquivo, **teste_novo.xml**. Vou deixar o programa bem "verboso" aqui para vocês verem como é feita a coisa:

```
require "rexml/document"

# cria um novo documento XML
doc = REXML::Document.new

# cria a declaração do arquivo XML
xmldecl = REXML::XMLDecl.new("1.0","ISO-8859-1","no")
doc.add xmldecl

# elemento raiz
raiz = REXML::Element.new "montadoras"
doc.add_element raiz

# tipo do arquivo
tipo = REXML::Element.new "tipo"
tipo.text = "Listagem"
raiz.add_element tipo

# primeira montadora
montadora = REXML::Element.new "montadora"
montadora.attributes["nome"] = "Volkswagen"

# nó de texto, localizacao
localizacao = REXML::Element.new "localizacao"
localizacao.text = "SP"
montadora.add_element localizacao

# carro
carro = REXML::Element.new "carro"
carro.attributes["modelo"]="Gol"
carro.attributes["cor"]="azul"
montadora.add_element carro

# carro
carro = REXML::Element.new "carro"
carro.attributes["modelo"]="Golf"
carro.attributes["cor"]="preto"
montadora.add_element carro

raiz.add_element montadora
```

```
# segunda montadora
montadora = REXML::Element.new "montadora"
montadora.attributes["nome"] = "Ford"

# nó de texto, localizacao
localizacao = REXML::Element.new "localizacao"
localizacao.text = "PR"
montadora.add_element localizacao

# carro
carro = REXML::Element.new "carro"
carro.attributes["modelo"]="Ka"
carro.attributes["cor"]="vermelho"
montadora.add_element carro

raiz.add_element montadora

# escreve para o arquivo
doc.write(File.open("teste_novo.xml","w"),0)
```

Vejamos o que aconteceu:

- Criado um novo documento XML através de REXML::Document.new.
- Criada a declaração do XML, especificando a versão, o encoding e o parâmetro standalone.
- Criado um elemento raiz, *montadoras*.
- O elemento raiz foi adicionado no documento.
- Criado um elemento que é um nó de texto, *tipo*, e adicionado ao elemento *montadora*.
- Criado um elemento *carro*, atribuídos os atributos *modelo* e *cor*, e inserido no elemento *montadora*.
- Repetido o passo da criação do elemento *carro* e adicionado o elemento *montadora* no elemento *raiz*.
- O processo de criação de *montadora* e *carro* foi repetido.
- Através de *doc.write*, foram salvos todos os elementos em um arquivo. O primeiro parâmetro é o nome do arquivo, o segundo é o nível de indentação do arquivo (maior ou igual à 0 deixa o arquivo mais "bonitinho").

O conteúdo de **teste_novo.xml** é similar ao conteúdo de **teste.xml**:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<montadoras>
   <tipo>Listagem</tipo>
   <montadora nome="Volkswagen">
   <localizacao>SP</localizacao>
      <carro modelo="Gol" cor="azul"/>
      <carro modelo="Golf" cor="preto"/>
   </montadora>
   <montadora nome="Ford">
      <localizacao>PR</localizacao>
      <carro modelo="Ka" cor="vermelho"/>
   </montadora>
</montadoras>
```

Capítulo 11

YAML

Podemos definir o YAML (http://www.yaml.org/) (YAML Ain't Markup Language - pronuncia-se mais ou menos como "ieimel", fazendo rima com a pronúncia de"camel", em inglês) como uma linguagem de definição ou markup **bem menos verbosa** que o XML. A sua sintaxe é mais limpa, e ele é bem útil para guardar, por exemplo, arquivos de configuração.

11.1 Lendo arquivos YAML

Vamos dar uma olhada em um exemplo simples, vou chamar esse arquivo de **teste.yml**:

```
---
- MySQL
- Oracle
```

No código, vamos usar:

```
require "yaml"

obj = YAML::load(File.open("teste.yml"))
puts "Li:"
puts obj
puts "obj é um objeto do tipo #{obj.class}"
puts "Escrevendo com dump:"
puts YAML::dump(obj)
puts "Escrevendo com y:"
y obj
```

Rodando isso, o resultado é:

```
Li:
MySQL
Oracle
obj é um objeto do tipo Array
Escrevendo com dump:
---
- MySQL
- Oracle
Escrevendo com y:
---
- MySQL
- Oracle
```

O arquivo YAML está representado como uma lista, o que resulta em um Array, olhem o tipo da classe que foi impressa (mais sobre YAML você pode aprender na URL já mencionada e em

http://yaml4r.sourceforge.net/cookbook/).

Podemos ver que tenho dois meios de imprimir o resultado: usando **dump** e **y**, que dão resultados similares. Pelo fato do último método ser mais curto, não vou usar mais usar *dump* nos outros exemplos.

Os métodos *dump* e *y* nos mostram uma representação bem-formatada do resultado lido do arquivo. Se rodarmos o código no irb, usando *puts* para imprimir os resultados, vamos ter algo como

```
irb(main):031:0> p obj
["MySQL", "Oracle"]
```

A sequência de caracteres "—" não aparece pois ela indica o começo de um arquivo YAML.

Podemos ter também Hashes no nosso arquivo YAML:

```
---
username: taq
password: nananinanao
```

Simplifiquei o código para:

```
require "yaml"

obj = YAML::load(File.open("teste.yml"))
puts "obj é um objeto do tipo #{obj.class}"
p obj
y obj
```

Resultado:

```
obj é um objeto do tipo Hash
{"username"=>"taq", "password"=>"nananinanao"}
---
username: taq
password: nananinanao
```

11.2 Gravando arquivos YAML

Vamos modificar um valor da Hash anterior, adicionar mais um valor e gravar os valores alterados no arquivo:

```
require "yaml"

obj = YAML::load(File.open("teste.yml"))
obj["password"] = "semchance"
obj["database"] = "Oracle"
y obj
File.open("teste.yml","w") do |f|
   f << obj.to_yaml
end
```

Conteúdo do arquivo YAML:

```
---
username: taq
database: Oracle
password: semchance
```

Lembrem-se que Hashes não são organizadas na mesma ordem que são definidas. Por isso que *database*, que é novo ali, está antes mesmo de *password*, que já estava lá.

A título de curiosidade, vejam como ficaria nosso exemplo do arquivo *teste.xml*, utilizado no capítulo anterior, em YAML:

```
---
tipo: Listagem
montadoras:
   - nome: Volskwagen
     localizacao: SP
     carros:
     -
        - modelo: Gol
          cor: azul
        - modelo: Golf
          cor: preto
   - nome: Ford
     localizacao: PR
     carros:
     -
        - modelo: Ka
          cor: vermelho
```

Aqui um pouco de código no irb para brincar com seus elementos:

```
[taq@~/code/ruby]irb
irb(main):001:0> require "yaml"
=> true
irb(main):002:0> obj = YAML::load(File.open("montadoras.yml"))
=> {"montadoras"=>[{"localizacao"=>"SP", "carros"=>[[{"cor"=>"azul",
"modelo"=>"Gol"}, {"cor"=>"preto", "modelo"=>"Golf"}]], "nome"=>"Volskwagen"},
{"localizacao"=>"PR", "carros"=>[[
{"cor"=>"vermelho", "modelo"=>"Ka"}]], "nome"=>"Ford"}], "tipo"=>"Listagem"}
irb(main):003:0> obj["montadoras"][0]["nome"]
=> "Volskwagen"
irb(main):004:0> obj["montadoras"][1]["nome"]
=> "Ford"
irb(main):005:0>
```

Capítulo 12

XSLT

XSLT é uma linguagem para transformar documentos XML em outros documentos, sejam eles outros XML, HTML, o tipo que você quiser e puder imaginar. XSLT é desenhado para uso com XSL, que são folhas de estilo para documentos XML. Alguns o acham muito "verboso", mas para o que ele é proposto, é bem útil.

Você pode conhecer mais sobre XSLT na URL oficial do W3C (http://www.w3.org/TR/xslt) e pode aprender alguma coisa sobre transformações de documentos XML em um tutorial *básico* que mantenho em meu site (http://beam.to/taq/xml.php).

O uso de XSLT em Ruby é feito pela classe Ruby/XSLT (http://www.rubyfr.net/ruby-xslt.asp), que permite algumas transformações *básicas*. Baixe os fontes e os instale, é muito simples.

Após isso vamos usar o nosso arquivo *teste.xml* (aquele das montadoras) para mostrar um exemplo de transformação. Para isso vamos precisar de uma folha de estilo XSL:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="html" encoding="iso-8859-1" indent="no"/>

<xsl:template match="/montadoras">
   <html>
      <head>
         <title>Teste de XSLT</title>
      </head>
      <body>
         <xsl:apply-templates/>
      </body>
   </html>
</xsl:template>

<xsl:template match="tipo">
   <h1><xsl:value-of select="."/></h1>
</xsl:template>

<xsl:template match="montadora">
   <h2><xsl:value-of select="@nome"/></h2>
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="localizacao">
   <b>localizada em <xsl:value-of select="."/></b><br/>
   Carros fabricados:<br/>
```

```
</xsl:template>

<xsl:template match="carro">
   <xsl:value-of select="@modelo"/> - <xsl:value-of select="@cor"/><br/>
</xsl:template>

</xsl:stylesheet>
```

O código Ruby para usar XSLT é simples, vamos criar um arquivo chamado *xslt.rb* com o seguinte conteúdo:

```
require "xml/xslt"

xslt = XML::XSLT.new()      # cria o processador XSLT
xslt.xslfile = "teste.xsl"  # informa o arquivo da folha de estilo
xslt.xmlfile = "teste.xml"  # informa o arquivo XML
xslt.save("teste.html")     # salva o resultado em teste.html
print xslt.serve            # mostra o resultado
```

Rodando o programa vamos ter o resultado gravado no arquivo *teste.html* e apresentado na tela. Abrindo o arquivo vamos ver:

Listagem

Volkswagen

localizada em SP
Carros fabricados:
Gol - azul
Golf - preto

Ford

localizada em PR

Carros fabricados: Ka - vermelho

Capítulo 13

Usando Ruby na web

Vou apresentar aqui algumas maneiras de se usar Ruby junto com um servidor web. Você poderá ficar tentado a substituir alguns (ou todos!) scripts que usa em PHP, ASP ou derivados e ganhar o poder de Ruby diretamente no seu browser.

13.1 mod_ruby

O **mod_ruby** é um módulo para rodar Ruby no Apache. Podemos rodar sem o mod_ruby, no esquema de scripts CGI, mas com ele fica mais rápido, pois são executados nativamente.

13.1.1 Instalando o mod_ruby

Podemos dar o download da versão mais nova do mod_ruby em http://www.modruby.net/en/. Após efetuar o download, compile e instale o programa:

```
$ tar zxvf mod_ruby-x.y.z.tar.gz
$ cd mod_ruby-x.y.z/
$ ./configure.rb --with-apxs=/usr/local/apache/bin/apxs
$ make
# make install
```

Pessoalmente, no lugar do *make install*, no Slackware eu uso o *checkinstall* (http://asic-linux.com.mx/~izto/checkinstall/), que permite o gerenciamento dos pacotes instalados de uma forma mais fácil.

Não se esqueçam de adequar o path do Apache para o path correto do seu computador.

13.1.2 Configurando o servidor Apache

Adicione as seguintes linhas no final do seu arquivo *httpd.conf*:

```
LoadModule ruby_module /usr/local/apache2/modules/mod_ruby.so

<IfModule mod_ruby.c>
   RubyRequire apache/ruby-run

   # Excucute files under /ruby as Ruby scripts
   <Location /ruby>
      SetHandler ruby-object
      RubyHandler Apache::RubyRun.instance
      Options ExecCGI
   </Location>

   # Execute *.rbx files as Ruby scripts
```

```
   <Files *.rbx>
      SetHandler ruby-object
      RubyHandler Apache::RubyRun.instance
      Options ExecCGI
   </Files>
</IfModule>
```

Indicamos ali que vamos rodar os arquivos dentro do diretório *ruby* e qualquer um que tenha a extensão **rbx** (e que tenham flag de executável) usando o módulo mod_ruby.

13.1.3 Testando o mod_ruby

Vamos conectar em nosso banco de dados MySQL, e mostrar os resultados no browser. Usaremos o exemplo do capítulo sobre banco de dados, da seção "trabalhanco com blocos", gravando em um arquivo chamado **modruby.rbx**:

```
require "dbi"

DBI.connect("DBI:Mysql:taq:localhost","taq","******") do |con|
   con.do("drop table if exists pedidos")

   # cria tabela
   sql = <<FIMSQL
create table pedidos
(numero    int(6) not null,
 cliente   int(6) not null,
 descricao varchar(50) not null,
 valor     decimal(10,2) not null)
FIMSQL
   con.do(sql)

   # insere alguns valores
   con.prepare("insert into pedidos values (?,?,?,?)") do |ins|
      [[1,1,'CD - Anthrax - Greater of Two Evils',25.00],
      [2,1,'DVD - Slayer - Still Reigning',55.00],
      [3,2,'CD - Judas Priest - Angel of Retribution',30.00]].each {|v| ins.execute(*v)}
   end

   # consulta todos os registros
   con.execute("select * from pedidos") do |rst|
      rst.fetch do |row|
         printf "pedido:%d cliente:%d produto:%s valor:%.2f\n",
         row["numero"],row["cliente"],row["descricao"],row["valor"]
      end
   end
end
```

Abra seu browser e aponte para onde você gravou o arquivo, por exemplo, **http://localhost/modruby.rbx**. O resultado será uma página com os resultados:

```
pedido:1 cliente:1 produto:CD - Anthrax - Greater of Two Evils valor:25.00
pedido:2 cliente:1 produto:DVD - Slayer - Still Reigning valor:55.00
pedido:3 cliente:2 produto:CD - Judas Priest - Angel of Retribution valor:30.00
```

Mas temos um problema aí. Se olharmos o código-fonte da página, veremos que não há marcação HTML alguma ali. Lógico que podemos inserir a marcação dentro do código do Ruby, mas isso ia "poluir" muito seu código final. É aí que entra o **eruby**.

13.2 eruby

Como diz em seu site (http://www.modruby.net/en/index.rbx/eruby/whatis.html), "**eruby** é uma implementação de eRuby em C". eRuby por sua vez é uma linguagem que permite embutir código Ruby em arquivos texto, no nosso caso, arquivos HTML.

13.2.1 Instalando o eruby

Efetue o download da versão mais recente no site oficial (http://www.modruby.net/en/index.rbx/eruby/download.html). Instale da mesma maneira que instalou o **mod_ruby**.

13.2.2 Configurando o servidor Apache

Adicione as seguintes linhas no seu arquivo **httpd.conf**:

```
<IfModule mod_ruby.c>
   RubyRequire apache/eruby-run

   # Handle files under /eruby as eRuby files
   <Location /eruby>
      SetHandler ruby-object
      RubyHandler Apache::ERubyRun.instance
   </Location>

   # Handle *.rhtml files as eRuby files
   <Files *.rhtml>
      SetHandler ruby-object
      RubyHandler Apache::ERubyRun.instance
   </Files>
</IfModule>
```

13.2.3 Testando o eruby

Para diferenciar o conteúdo da página do nosso exemplo anterior, vou chutar o balde agora: vou inserir uma página formatada bonitinha, com CSS e tudo. Vamos ver como ficaria nosso arquivo **eruby.rhtml** (reparem que o código Ruby se encontra entre < % e % >):

```
<html>
   <head>
      <title>Teste do eruby</title>
   </head>
   <style>
      body {
         font-family:verdana,arial,helvetica,sans-serif;
         font-size:12px;
      }
      table {
         border-spacing:0px;
      }
      caption {
         background-color:#EEEEEE;
         font-weight:bold;
      }
      th {
         background-color:#99C9F9;
      }
      th, td, caption {
```

```
            text-align:left;
            border:1px solid #000000;
            border-collapse:collapse;
            margin:0px;
            padding:5px;
         }
      </style>
      <body>
         <h1>Listagem de pedidos</h1>
         <%
         require "dbi"

         con = DBI.connect("DBI:Mysql:taq:localhost","taq","******")
         con.do("drop table if exists pedidos")

         # cria tabela
sql = <<FIMSQL
create table pedidos
(numero    int(6) not null,
 cliente   int(6) not null,
 descricao varchar(50) not null,
 valor     decimal(10,2) not null)
FIMSQL
         con.do(sql)
         # insere alguns valores
         con.prepare("insert into pedidos values (?,?,?,?)") do |ins|
            [[1,1,'CD - Anthrax - Greater of Two Evils',25.00],
            [2,1,'DVD - Slayer - Still Reigning',55.00],
            [3,2,'CD - Judas Priest - Angel of Retribution',30.00]].each {|v| ins.execute(*v)}
         end
         %>
         <table>
            <caption>Pedidos efetuados no mês</caption>
         <%
         # consulta todos os registros
         con.execute("select * from pedidos") do |rst|
            # monta o cabeçalho
            print "<tr>"
            rst.column_info.each {|ci| print "<th>"+ci["name"]+"</th>"}
            print "</tr>\n"

            # mostra os valores
            rst.fetch do |row|
               print "<tr>"+(row.map {|col| "<td>#{col}</td>"}.to_s)+"</tr>\n"
            end
         end
         con.disconnect
         %>
         <table>
      </body>
</html>
```

Aponte seu browser para onde você gravou o arquivo **eruby.rhtml**, o resultado vai ser:

Listagem de pedidos

Pedidos efetuados no mês			
numero	**cliente**	**descricao**	**valor**
1	1	CD - Anthrax - Greater of Two Evils	25.00
2	1	DVD - Slayer - Still Reigning	55.00
3	2	CD - Judas Priest - Angel of Retribution	30.00

Figura 13.1: Teste do eruby

Mais bonito, não? ;-) E melhor para administrar a dupla HTML/Ruby.

13.2.4 Sintaxe do eruby

Um bloco de código Ruby começa com < % e termina com % >. O output do bloco é enviado para o browser. Para mostrar como é essa sintaxe, podemos rodar o eruby da linha de comando também. Crie um arquivo, por exemplo, **erubysintaxe.rhtml** e vamos ir inserindo conteúdo e testando.

Conteúdo de erubysintaxe.rhtml:

```
<%
   puts "Oi!"
%>
```

Rodando:

```
[taq@~/code/ruby]eruby erubysintaxe.rhtml
Oi!
```

Se < % é seguido de =, o valor do bloco vai ser mostrado:

```
<%=
   "Oi!"
%>
```

Rodando (fica a mesma coisa):

```
[taq@~/code/ruby]eruby erubysintaxe.rhtml
Oi!
```

Se < % é seguido de #, o bloco será reconhecido como um comentário:

```
Oi, <%# isso é um comentário! %>mundo!
```

Rodando:

```
[taq@~/code/ruby]eruby erubysintaxe.rhtml
Oi, mundo!
```

Se uma linha começa com %, será avaliada como um programa Ruby e substitui a linha com o seu output:

```
% x = 1 + 1
<%= x %>
```

Rodando:

```
[taq@~/code/ruby]eruby erubysintaxe.rhtml
2
```

13.3 CGI

Podemos usar o módulo **cgi** para lidarmos com formulários, cookies e sessões (tchau,PHP!:-).
O jeito que vou mostrar o módulo cgi aqui é fazendo um mix com o mod_ruby/eruby. Tem outro jeito de fazermos o módulo construir praticamente nossa página (inclusive com forms) inteira, mas acho interessante deixar a coisa mais visível aqui para que possamos ver todo o código do página.

13.3.1 Forms

Vamos testar um formulário simples, vou criar o arquivo *form.rhtml*:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
   <head>
      <title></title>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
      <style>
         body {
            font-family:verdana,arial,helvetica,sans-serif;
            font-size:12px;
        }
   </style>
   </head>
   <body>
      <%
      require "cgi"

      cgi = CGI::new
      if cgi["nome"].length > 0
         puts "<h1>Oi, #{cgi['nome']}!</h1>"
         exit
      end
      %>
      <form name="dados" action="form.rhtml" method="post">
         <label for="nome">Nome</label>
         <input type="text" name="nome"/>
         <br/><br/>
         <input type="submit" value="Enviar"/>
      </form>
   </body>
</html>
```

Esse formulário requisita um nome, e quando enviado através do POST, é verificado que há conteúdo no valor enviado *nome*, apresentado o nome e terminada a execução do script. Abra ele no seu webserver, preencha com um nome e clique o botão "Enviar" para ver o resultado.

Temos aí um problema que a galera do PHP pode identificar como sendo os arrays *$_POST* e *$_GET*. Se no browser o usuário inserir a URL *http://localhost/form.rhtml?nome=Anonimo*, o formulário vai dizer "Oi, Anonimo!", "pulando" o formulário, uma situação que é bem aconselhável prestar atenção.

Para barrarmos esse tipo de coisa, só aceitando os valores que vieram de um *POST* ou de um *GET*, não permitindo valores de um em outro, podemos alterar *form.rhtml* para:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
   <head>
```

```
      <title></title>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
      <style>
         body {
            font-family:verdana,arial,helvetica,sans-serif;
            font-size:12px;
          }
   </style>
   </head>
   <body>
      <%
      require "cgi"

      cgi = CGI::new
      if cgi["nome"].length > 0 and cgi.request_method == "POST"
         puts "<h1>Oi, #{cgi['nome']}!</h1>"
         exit
      end
      %>
      <form name="dados" action="form.rhtml" method="post">
         <label for="nome">Nome</label>
         <input type="text" name="nome"/>
         <br/><br/>
         <input type="submit" value="Enviar"/>
      </form>
   </body>
</html>
```

Verifiquei o método do form com **cgi.request_method**, só vou receber parâmetros através do que foi enviado no form, pelo método POST, se foi o método utilizado.

13.3.2 Cookies

Cookies são facilmente manipulados usando o hash **cgi.cookies**. A única coisa que vocês podem achar estranha no exemplo a seguir é o armazenamento usando o método **header**, perguntando "mas peraí, os headers já não foram enviados?". Até foram, mas é o único jeito (sem usar **out**, que teria que construir a página do jeito que eu mencionei ter evitado no começo do capítulo) para armazenar o cookie. Vamos dar uma olhada no exemplo:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
   <head>
      <title></title>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
      <style>
         body {
            font-family:verdana,arial,helvetica,sans-serif;
            font-size:12px;
         }
      </style>
   </head>
   <body>
      <%
         require "cgi"

         cgi = CGI::new
```

```
            if cgi["nome"].length > 0 and cgi.request_method == "POST"
               puts "<h1>Oi, #{cgi['nome']}!</h1>"
               cookie = CGI::Cookie.new("name"=>"tutrubyform",
                                        "value"=>cgi["nome"],
                                        "expires"=>Time.local(Time.now.year+1,
                                                              Time.now.mon,
                                                              Time.now.day,
                                                              Time.now.hour,
                                                              Time.now.min,
                                                              Time.now.sec));
               cgi.header("cookie"=>[cookie])
               exit
            end
         %>
         <form name="dados" action="form.rhtml" method="post">
            <%
               cookie = cgi.cookies["tutrubyform"]
               if cookie.value.length > 0
                  puts "Bem-vindo de volta, <b>#{cookie.value}</b>!<br/><br/>"
               end
            %>
            <label for="nome">Nome</label>
            <input type="text" name="nome"/>
            <br/><br/>
            <input type="submit" value="Enviar"/>
         </form>
      </body>
   </html>
```

As únicas coisas que mudaram em relação ao exemplo anterior, foram que se há um parâmetro enviado pelo form, é criado um novo cookie, com nome (name) igual "tutrubyform", valor (value) igual o nome informado no form, e data de expiração (expires) igual a data corrente mais um ano.

13.3.3 Sessões

Sessões são um jeito prático de manter dados persistentes entre suas requisições ao webserver. Sessões também se comportam como um Hash, e utilizam (sem você ver) cookies. Vamos ver um exemplo:

session.rhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
   <head>
      <title>Teste de sessão</title>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
      <style>
         body {
            font-family:verdana,arial,helvetica,sans-serif;
            font-size:12px;
         }
      </style>
   </head>
   <body>
      <%
         require "cgi"
         require "cgi/session"
```

```
            cgi = CGI::new
            session = CGI::Session.new(cgi,"prefix"=>"rubysession")
            session["user"]="TaQ"
            print "Usuário da sessão marcado como #{session['user']}"
            session.close
            cgi.header
         %>
         <br/>
         <a href="session2.rhtml">clique aqui para ir para a segunda página</a>
      </body>
   </html>
```

session2.rhtml

```
   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
   <html>
      <head>
         <title>Teste de sessão</title>
         <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
         <style>
            body {
               font-family:verdana,arial,helvetica,sans-serif;
               font-size:12px;
            }
         </style>
      </head>
      <body>
         <%
            require "cgi"
            require "cgi/session"

            cgi = CGI::new
            session = CGI::Session.new(cgi,"prefix"=>"rubysession")
            print "Oi, #{session['user']}"
            session.close
            session.delete
         %>
      </body>
   </html>
```

Vamos analisar o código aqui. Primeiro, *session.rthml*. Além de importar o módulo *cgi*, precisamos também de **cgi/session**. Depois de criar o objeto relativo ao cgi, criamos uma sessão usando **CGI::Session.new**, onde eu passei como parâmetro o objeto cgi (obrigatório) e um hash com o valor **"prefix"=>"rubysession"**.

Uma explicação sobre prefix é valida: para cada sessão, um arquivo é criado no diretório temporário do seu sistema operacional (/tmp no Linux), e usando um prefixo os arquivos serão criados com o nome <**prefixo**>**número**, onde número é um hash do número da sessão, gerado automaticamente.

Fica uma observação importante: se utilizarmos o prefixo em um arquivo para criar uma sessão, devemos usá-lo novamente em outros arquivos onde queremos recuperar essa sessão, caso não o usemos o interpretador vai criar uma outra sessão com o prefixo default. Então, mantenha uma certa padronização ali.

Logo após criar a sessão, armazenei um código de usuário, com o valor "TaQ", usando **session["user"]="TaQ"**, e fechei a sessão com close. Fechar a sessão não significa destruí-la, e sim armazenar o estado da sessão

no seu armazenamento persistente, que no caso corrente é o arquivo de sessão citado acima. Se você quiser atualizar o estado mas sem fechar a sessão, pode usar o método **update**.

Uma dica importante é que para lidarmos com as sessões é necessário enviar suas informações para o navegador. Isso é feito usando novamente **cgi.header** (que, novamente, apesar do nome que pode confundir um pouco, funciona).

Nesse ponto já temos nossa sessão criada e armazenada, e se você verificar seu diretório temporário verá um arquivo com o nome iniciando com o prefixo utilizado (rubysession). Agora vamos clicar no link e ir para *session2.rhtml*.

Em *session2.rhtml*, pedimos para criar a sessão novamente. Como já temos uma criada, ela será reutilizada (se quiséssemos forçar a criação de uma sessão nova, poderíamos utilizar no método new o parametro "new_session"=>"true", isso será visto adiante).

Recuperei o nome do usuário usando **session["user"]**, fechei a sessão e a **apaguei** usando **session.delete**. Quando usamos *delete* o nosso armazenamento persistente (de novo, no nosso caso, o arquivo) é apagado. Se verificarmos nosso diretório temporário, não há mais o arquivo identificado anteriormente.

Para criarmos seguramente uma nova sessão, vai uma dica do próprio Matz: delete alguma sessão já existente e force a criação de uma nova, para evitar que sessões antigas, além de consumir recursos, possam ser "sequestradas" por algum espertinho. O código de *session.rb* ficaria assim:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
   <head>
      <title>Teste de sessão</title>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
      <style>
         body {
            font-family:verdana,arial,helvetica,sans-serif;
            font-size:12px;
         }
      </style>
   </head>
   <body>
      <%
         require "cgi"
         require "cgi/session"

         cgi = CGI::new
         # tenta recuperar uma sessao anterior, se conseguiu, a deleta, se não,
         # gera uma exceção
         begin
            session = CGI::Session.new(cgi,"prefix"=>"rubysession","new_session"=>false)
            session.delete
         rescue ArgumentError
         end
         # força uma sessão nova
         session = CGI::Session.new(cgi,"prefix"=>"rubysession","new_session"=>true)
         session["user"]="TaQ"
         print "Usuário da sessão marcado como #{session['user']}"
         session.close
         cgi.header
      %>
      <br/>
```

```
      <a href="session2.rhtml">clique aqui para ir para a segunda página</a>
   </body>
</html>
```

Uma coisa bem interessante que as sessões do Ruby suportam é que você pode escolher o seu método de persistência, usando o parâmetro **database_manager**. Temos as seguintes opções:

- **CGI::Session::FileStore** o método default. Os dados são gravadas em um arquivo.
- **CGI::Session::MemoryStore** os dados são armazenados na memória, e persistem enquanto o interpretador Ruby está carregado.
- **CGI::Session::PStore** permite o armazenamento de também de objetos (ao contrário dos outros métodos que só permitem strings).

O último método, PStore, é bem interessante. Ele permite que você armazene qualquer objeto do Ruby na sua sessão (uma analogia seria usar os métodos **serialize/unserialize** do PHP para fazer esse tipo de armazenamento). Vamos armazenar uma classe na nossa sessão, alterando *session.rhtml* e *session2.rhtml*:

session.rhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
   <head>
      <title>Teste de sessão</title>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
      <style>
         body {
            font-family:verdana,arial,helvetica,sans-serif;
            font-size:12px;
         }
      </style>
   </head>
   <body>
      <%
         require "cgi"
         require "cgi/session"
         require "cgi/session/pstore"

         cgi = CGI::new
         # tenta recuperar uma sessao anterior, se conseguiu, a deleta, se não,
         # gera uma exceção
         begin
            session = CGI::Session.new(cgi,"prefix"=>"rubysession","new_session"=>false,
                                "database_manager"=>CGI::Session::PStore)
            session.delete
         rescue ArgumentError
         end
         # força uma sessão nova
         session = CGI::Session.new(cgi,"prefix"=>"rubysession","new_session"=>true,
                                "database_manager"=>CGI::Session::PStore)
         session["user"]="TaQ"
         session["time"]=Time::new
         print "Usuário da sessão marcado como #{session['user']}<br/>"
         print "Guardei um objeto Time com o valor #{session['time']}"
         session.close
         cgi.header
```

```
      %>
      <br/>
      <a href="session2.rhtml">clique aqui para ir para a segunda página</a>
   </body>
</html>
```

session2.rhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
   <head>
      <title>Teste de sessão</title>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
      <style>
         body {
            font-family:verdana,arial,helvetica,sans-serif;
            font-size:12px;
         }
      </style>
   </head>
   <body>
      <%
         require "cgi"
         require "cgi/session"

         cgi = CGI::new
         session = CGI::Session.new(cgi,"prefix"=>"rubysession",
                                    "database_manager"=>CGI::Session::PStore)
         time = session["time"]
         print "Oi, #{session['user']}<br/>"
         print "Sua sessão começou dia #{time.day}"
         session.close
         session.delete
      %>
   </body>
</html>
```

No exemplo, armazenamos um objeto do tipo *Time* em session[“time”], e recuperamos o objeto com *time = session[“time”]*, e mostramos o valor do dia com *time.day*, provando que o objeto foi armazenado com todos seus métodos e propriedades.

Capítulo 14

Interface Gráfica

Nesse capítulo vamos ver como fazer programas em Ruby usando uma interface gráfica. As opções são muitas, mas vamos nos concentrar no GTK2 (que me juraram de pé juntos ser mais fácil para usar tanto em Linux como no Windows ;-).

14.1 Obtendo e compilando o módulo GTK2

Podemos baixar o pacote dos fontes do módulo do GTK2 em http://ruby-gnome2.sourceforge.jp/, onde inclusive podemos contar com um ÓTIMO tutorial no endereço http://ruby-gnome2.sourceforge.jp/hiki.cgi?tut-gtk, que inclusive vou usar como base aqui, em uma versão mais resumida.

Há também um tutorial em http://ruby-gnome2.sourceforge.net/tut/toc.htm, mais "avançado", que foi direcionado mais para programadores vindos do C.

Depois de pegar o pacote mais recente do módulo (eu tive problemas compilando a versão 0.11.0, mas a 0.12.0 compilou perfeitamente), o instale com os comandos:

```
ruby extconf.rb
make
make install (ou use o checkinstall ;-)
```

14.2 Hello, GUI world!

Vamos fazer um pequeno programa aqui, lotado de comentários, para vermos como funciona a coisa:

```
require "gtk2" # requisita módulo gtk2

Gtk.init # inicializa a library
window = Gtk::Window.new # cria uma nova janela
label = Gtk::Label.new("Oi, mundo GUI!") # cria uma nova label
window.add(label) # adiciona a label na janela
window.border_width = 10 # adiciona uma borda na janela
window.show_all # mostra o container e todos componentes dentro dele
Gtk.main # chamada para o gtk aguardar eventos
```

Se executarmos esse programa, teremos algo como:

Figura 14.1: Hello, GUI World!

14.3 Eventos

Uma coisa interessante ali, é que se clicarmos no botão para fechar a janela, nada acontece.
Isso por que não temos nenhum evento associado ao fechamento da janela.
Vamos alterar nosso programa um pouquinho:

```
require "gtk2" # requisita módulo gtk2

Gtk.init # inicializa a library
window = Gtk::Window.new # cria uma nova janela
label = Gtk::Label.new("Oi, mundo GUI!") # cria uma nova label
window.add(label) # adiciona a label na janela
window.border_width = 10 # adiciona uma borda na janela
window.show_all # mostra o container e todos componentes dentro dele

window.signal_connect("destroy"){
   puts "Recebi um evento para destruir o componente."
   puts "Nesse caso, a janela principal do programa."
Gtk.main_quit # finaliza o programa
}

Gtk.main # chamada para o gtk aguardar eventos
```

Na alteração, foi interceptado o evento de destruição do objeto, o qual chamou **Gtk.main_quit** para finalizar o programa.

Quando clicamos no botão para fechar a janela, ela fecha e mostra as mensagens no console.

Vamos ver mais alguns exemplos de eventos, inserindo um botão na janela, para que possamos fechá-la. Mas para inserir um botão junto com a nossa label, vamos precisar saber um pouco sobre "packing" de componentes, ou fazendo uma tradução mais explicativa, como dispor vários componentes na tela.

14.4 Packing

O "empacotamento" de componentes é feito através de containers invisíveis chamados "box(es)"(caixas), que podem ser horizontais ou verticais.

Nas caixas horizontais os componentes são incluídos da esquerda para a direita, ou ao inverso, mas dependendo do método utilizado. Nas verticais são inseridos do topo para o fundo, e também podem ter sua ordem de inserção invertida.

Podemos criar as caixas usando **Gtk::HBox.new** para as horizontais e **Gtk::VBox.new** para as verticais.

Os métodos para inserir os componentes nas caixas são:

- **pack_start**, que insere componentes a partir do começo do container.
- **pack_end**, que insere componentes a partir do fim do container.

Vamos alterar nosso programinha outra vez, e de lambuja inserir um evento no botão:

```
require "gtk2" # requisita módulo gtk2

Gtk.init # inicializa a library
window = Gtk::Window.new # cria uma nova janela
vbox = Gtk::VBox.new(false,10) # cria uma caixa vertical
label = Gtk::Label.new("Oi, mundo GUI!") # cria uma nova label
vbox.pack_start(label) # adiciona a label na caixa

button = Gtk::Button.new("Me feche!") # cria um botão
vbox.pack_start(button) # adiciona um botão na caixa

window.add(vbox) # adiciona a caixa na janela

window.border_width = 10 # adiciona uma borda na janela
window.show_all # mostra o container e todos componentes dentro dele

window.signal_connect("destroy"){
puts "Recebi um evento para destruir o componente."
puts "Nesse caso, a janela principal do programa."
Gtk.main_quit # finaliza o programa
}

button.signal_connect("clicked"){
puts "O botão foi clicado! Terminando programa ..."
Gtk.main_quit
}

Gtk.main # chamada para o gtk aguardar eventos
```

Isso nos dá algo como: O que foi feito foi adicionar uma caixa vertical (os parâmetros passados se referem

Figura 14.2: Mais de um componente na janela

respectivamente ao fato dos componentes não terem seu espaço distribuído de forma homogênea (false) e terem 10 pixeis de espaço entre eles), adicionar os componentes na caixa e adicionar a caixa na janela.

Depois foi inserido um evento "clicked" no botão. Agora, quando ele é clicado, o programa é terminado.
Agora vamos adicionar mais um botão, usando uma caixa horizontal, e conectar o sinal "clicked" dele para mostrar um diálogo de mensagem, que vai mostrar um diálogo de mensagem com dois botões com as opções SIM e NÃO.

Uma observação **MUITO** importante é que as strings utilizadas nos componentes tem que ser convertidas em UTF-8. Como aqui utilizo o encoding ISO-8859-1, tenho que utilizar o método **GLib.locale_to_UTF8** para que a acentuação fique correta com o ruby-gtk2.

Vamos ver o programa:

```
require "gtk2" # requisita módulo gtk2

Gtk.init # inicializa a library
window = Gtk::Window.new # cria uma nova janela
vbox = Gtk::VBox.new(false,10) # cria uma caixa vertical
label = Gtk::Label.new("Oi, mundo GUI!") # cria uma nova label
vbox.pack_start(label) # adiciona a label na caixa

button  = Gtk::Button.new("Me feche!") # cria um botão
button2 = Gtk::Button.new("Mensagem") # cria outro botão

hbox = Gtk::HBox.new(false,10) # cria uma caixa horizontal
hbox.pack_start(button) # adiciona um botão na caixa
hbox.pack_start(button2) # adiciona um botão na caixa
vbox.add(hbox) # adiciona a caixa horizontal na vertical

window.add(vbox) # adiciona a caixa na janela

window.border_width = 10 # adiciona uma borda na janela
window.show_all # mostra o container e todos componentes dentro dele

window.signal_connect("destroy"){
puts "Recebi um evento para destruir o componente."
puts "Nesse caso, a janela principal do programa."
Gtk.main_quit # finaliza o programa
}

button.signal_connect("clicked"){
puts "O botão foi clicado! Terminando programa ..."
Gtk.main_quit
}

button2.signal_connect("clicked"){
# criando um diálogo
msg = Gtk::MessageDialog.new(nil,
  Gtk::MessageDialog::DESTROY_WITH_PARENT,
  Gtk::MessageDialog::INFO,
  Gtk::MessageDialog::BUTTONS_YES_NO,
  GLib.locale_to_utf8("Oi! Você escolheu ver a mensagem.\nClique em um dos botões abaixo."))

# conecta o sinal apenas para mostrar o que acontece
# quando clicamos em um botão
msg.signal_connect("response"){
puts "Sinal recebido!"
}

# mostra o diálogo, interceptando a resposta
msg.run do |response|
if response == Gtk::Dialog::RESPONSE_YES
puts "O botão clicado foi SIM."
else
puts "O botão clicado foi NÃO."
end
end
msg.destroy
}
```

```
Gtk.main # chamada para o gtk aguardar eventos
```

Isso nos dá algo como: Podemos gerar a mesma disposição de tela usando uma metodologia conhecida

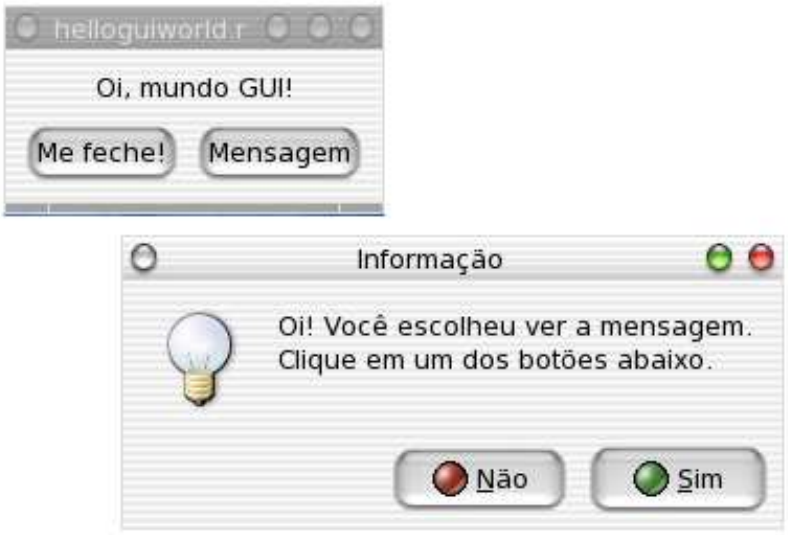

Figura 14.3: Vários botões e um diálog de mensagem

por quem desenvolve em HTML (tremei, adeptos do tableless): tabelas!

14.5 Posicionamento usando tabelas

O programa a seguir produz o mesmo resultado que usar "packing".

Fica a observação em relação ao método **attach_defaults**: temos que especificar as posições *esquerda, direita, topo, fundo*, sendo que esquerda é a coluna esquerda onde o componente *começa na horizontal*, direita é a coluna onde o componente *termina na horizontal*, topo é onde o componente *começa na vertical* e o fundo é onde o componente *termina na vertical*. Importante notar que a coluna onde o componente termina não é utilizada pelo mesmo, mas serve de referência em relação ao seu tamanho.

Assim sendo, *table.attach_defaults(0,2,0,1)* especifica que *o componente vai começar na coluna 0, linha 0, e vai terminar* **encostando** *na coluna 2, linha 1.* Fica meio confuso mas seria mais ou menos como:

```
          coluna 0   coluna 1   coluna 2
        +----------+----------+----------+
linha 0 |//// componente //// |  vazio   |
linha 1 +----------+----------+----------+
```

A margem direita/fundo do componente encosta, mas não transpõe, a margem esquerda/topo das colunas finais de referência.

Aí vai o código:

```
require "gtk2" # requisita módulo gtk2

Gtk.init # inicializa a library
window = Gtk::Window.new # cria uma nova janela
window.title = "Tabelas!" # informa um título para a janela
```

```
table = Gtk::Table.new(2, 2, true) # cria uma tabela de 2x2 com espaço homogêneo
table.row_spacings = 10 # espaçamento de 10 pixeis entre as linhas
table.column_spacings = 10 # espaçamento de 10 pixeis entre as colunas

label = Gtk::Label.new("Oi, mundo GUI!") # cria uma nova label
table.attach_defaults(label,0,2,0,1) # insere na tabela

button  = Gtk::Button.new("Me feche!") # cria um botão
button2 = Gtk::Button.new("Mensagem") # cria outro botão

table.attach_defaults(button,0,1,1,2) # adiciona um botão na tabela
table.attach_defaults(button2,1,2,1,2) # adiciona um botão na tabela

window.add(table) # adiciona a tabela na janela
window.border_width = 10 # adiciona uma borda na janela
window.show_all # mostra o container e todos componentes dentro dele

window.signal_connect("destroy"){
puts "Recebi um evento para destruir o componente."
puts "Nesse caso, a janela principal do programa."
Gtk.main_quit # finaliza o programa
}

button.signal_connect("clicked"){
puts "O botão foi clicado! Terminando programa ..."
Gtk.main_quit
}

button2.signal_connect("clicked"){
# criando um diálogo
msg = Gtk::MessageDialog.new(nil,
  Gtk::MessageDialog::DESTROY_WITH_PARENT,
  Gtk::MessageDialog::INFO,
  Gtk::MessageDialog::BUTTONS_YES_NO,
  GLib.locale_to_utf8("Oi! Você escolheu ver a mensagem.\nClique em um dos botões abaixo."))

# conecta o sinal apenas para mostrar o que acontece
# quando clicamos em um botão
msg.signal_connect("response"){
puts "Sinal recebido!"
}

# mostra o diálogo, interceptando a resposta
msg.run do |response|
if response == Gtk::Dialog::RESPONSE_YES
puts "O botão clicado foi SIM."
else
puts "O botão clicado foi NÃO."
end
end
msg.destroy
}

Gtk.main # chamada para o gtk aguardar eventos
```

14.6 Mais alguns componentes

Vou detalhar mais alguns componentes aqui, mas MUITO mais documentação sobre os componentes pode ser encontrada em http://ruby-gnome2.sourceforge.jp/hiki.cgi?Ruby-GNOME2+API+Reference. Tem MUITA coisa lá, muitos outros componentes úteis.

14.6.1 CheckButton

Código:

```
require "gtk2" # requisita módulo gtk2

Gtk.init # inicializa a library

window = Gtk::Window.new
check = Gtk::CheckButton.new("Marque aqui")

window.add(check) # adiciona o componente na janela
window.border_width = 10 # adiciona uma borda na janela
window.show_all # mostra o container e todos componentes dentro dele

window.signal_connect("destroy"){
puts "O botão "+(check.active? ? "" : "NÃO ")+"está ativo."
Gtk.main_quit # finaliza o programa
}
Gtk.main
```

Resultado:

Figura 14.4: CheckButton

14.6.2 ComboBox

Código:

```
require "gtk2" # requisita módulo gtk2

Gtk.init # inicializa a library

window = Gtk::Window.new
combo = Gtk::ComboBox.new
combo.append_text(GLib.locale_to_utf8("Opção 1"))
combo.append_text(GLib.locale_to_utf8("Opção 2"))
combo.append_text(GLib.locale_to_utf8("Opção 3"))

window.add(combo) # adiciona o componente na janela
window.border_width = 10 # adiciona uma borda na janela
window.show_all # mostra o container e todos componentes dentro dele

window.signal_connect("destroy"){
puts "Item ativo: #{combo.active}"
```

```
Gtk.main_quit # finaliza o programa
}
Gtk.main # chamada para o gtk aguardar eventos
```

Resultado:

Figura 14.5: ComboBox

14.6.3 Campos texto

Código:

```
require "gtk2" # requisita módulo gtk2

Gtk.init # inicializa a library

window = Gtk::Window.new
text = Gtk::Entry.new

window.add(text) # adiciona o componente na janela
window.border_width = 10 # adiciona uma borda na janela
window.show_all # mostra o container e todos componentes dentro dele

window.signal_connect("destroy"){
puts "O texto digitado foi \""+text.text+"\""
Gtk.main_quit # finaliza o programa
}
Gtk.main
```

Resultado:

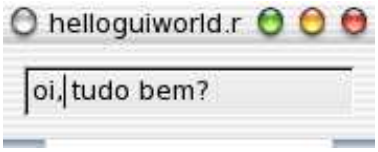

Figura 14.6: Campos texto

14.6.4 Radio buttons

Código:

```
require "gtk2" # requisita módulo gtk2

Gtk.init # inicializa a library
```

```
window = Gtk::Window.new
group  = Gtk::RadioButton.new # o grupo é um radio button vazio!
radio1 = Gtk::RadioButton.new(group,GLib.locale_to_utf8("Opção 1"))
radio2 = Gtk::RadioButton.new(group,GLib.locale_to_utf8("Opção 2"))
radio3 = Gtk::RadioButton.new(group,GLib.locale_to_utf8("Opção 3"))

vbox = Gtk::VBox.new
radio1.active = true
vbox.add(radio1) # adiciona o componente na janela
vbox.add(radio2) # adiciona o componente na janela
vbox.add(radio3) # adiciona o componente na janela

window.add(vbox)
window.border_width = 10 # adiciona uma borda na janela
window.show_all # mostra o container e todos componentes dentro dele

window.signal_connect("destroy"){
puts "O botão 1 "+(radio1.active? ? "":"NÃO ")+"está ativo"
puts "O botão 2 "+(radio2.active? ? "":"NÃO ")+"está ativo"
puts "O botão 3 "+(radio3.active? ? "":"NÃO ")+"está ativo"
Gtk.main_quit # finaliza o programa
}
Gtk.main
```

Resultado:

Figura 14.7: Radio buttons

Capítulo 15

Garbage collector

15.1 O algoritmo utilizado: mark-and-sweep

O algoritmo do *garbage collector* (coletor de lixo de objetos na memória que não estão mais sendo utilizados) que Ruby utiliza é o *mark-and-sweep*.

O algoritmo age em todos os objetos acessíveis direta ou indiretamente pelo programa, sendo que os objetos que são acessados diretamente são chamados de *roots*, e os indiretos são os que são referenciados por esses objetos acessíveis.

O trabalho é feito em duas fases: uma para marcar (*mark*) os objetos que estão acessíveis, e outra para varrer (*sweep*) os objetos que não foram marcados na primeira fase.

Na primeira fase, ele percorre todos os objetos que estão acessíveis no *root*, (e todos os objetos apontados por eles, os acessíveis indiretamente), usando um flag (por exemplo, *marked*), explicitando que aquele objeto está sendo acessado e tem sua importância no programa, não devendo ser destruído.

Na segunda fase, ele percorre a pilha (*heap*) do programa e verifica **todos** os objetos do Ruby, liberando o espaço de memória dos que não foram marcados como acessíveis na primeira fase, e dá um *reset* no flag dos que estão marcados. Por default, todos os objetos são criados como não-marcados.

Dessa maneira, os objetos que não estão sendo mais utilizados tem seu espaço realocado de volta para a memória.

Uma das desvantagens do algoritmo *mark-and-sweep* é que ele permite que o "lixo" (garbage) se acumule na memória até um certo ponto - a limpeza não é feita imediatamente após um objeto ter sua referência destruída, do mesmo modo que os métodos destrutores em C.

E há também o problema da fragmentação de memória : os blocos de memória precisam ser alocados em áreas contíguas, e mesmo se você tiver memória suficiente para uma alocação de, digamos, 200kb, pode ser que esses 200kb estejam fragmentados em um bloco de 100kb, um de 50kb e dois de 25kb, o que leva á áreas não-contíguas de memória, e ao acionamento do garbage collector para liberar os objetos que não estão mais acessíveis, **não livrando do fato que a memória possa continuar fragmentada** após a execução do garbage collector.

O garbage collector entra em ação quando a tentativa de alocação de memória falha (o que pode ser o caso da memória não-contígua comentado acima) ou quando, pelo menos na versão 1.8.2, a soma da alocação já feita atingiu cerca de 7Mb, e nesse momento é feita uma pausa no programa para que o algoritmo entre em ação.

Geralmente essa pausa não chega a ser um problema crítico, mas em casos que o tempo de resposta para o usuário tem que ser mínimo, pode ser que o usuário "sinta" a parada, que de tempos em tempos

interromperá o programa, pois o garbage collector **não roda em uma thread separada**.

15.2 Como funciona, na teoria

Vamos dar uma olhada em uma sequência de exemplo do garbage collector em ação. Vamos criar seis objetos, todos no *root*, onde os objetos 1, 2 e 3 estão conectados, sendo que o objeto 3 está apontando para o objeto 5. Os objetos 4 e 5 estão conectados. O objeto 6 não está conectado com nenhum outro.

Os objetos que estão em vermelho são os que não foram marcados como acessíveis ainda (lembre-se, no momento de criação todos os objetos não são marcados), os verdes são os que foram marcados como acessíveis:

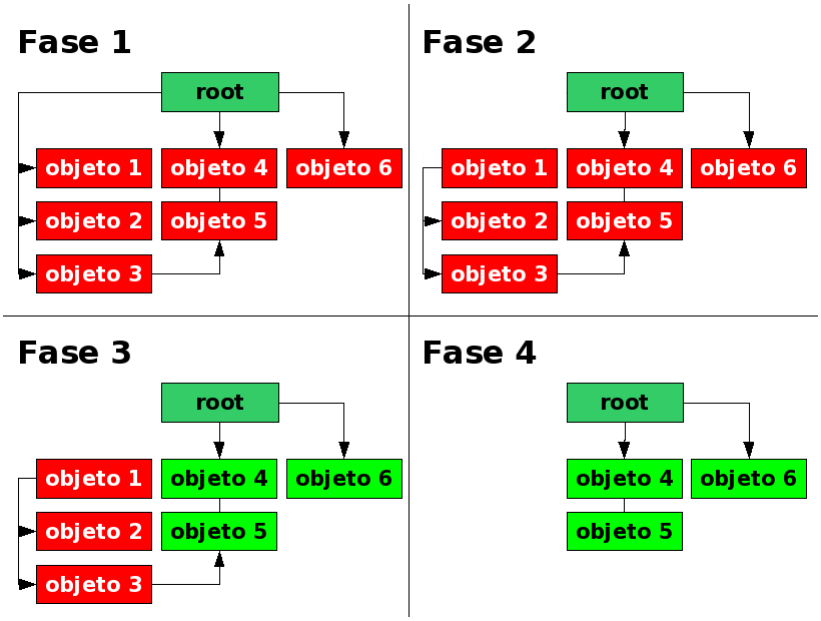

Figura 15.1: Sequência do garbage collector

- Na **Fase 1**, todos os objetos não estão marcados como acessíveis.
- Na **Fase 2**, continuam do mesmo jeito, porém o objeto 1 agora não está disponível no *root*.
- Na **Fase 3**, o algoritmo foi acionado, parando o programa e marcando (*mark*) os objetos que estão acessíveis.
- Na **Fase 4** foi executada a limpeza (*sweep*) dos objetos não-acessíveis, e retirado o flag dos que estavam acessíveis, forçando a sua verificação na próxima vez que o garbage collector rodar.

O garbage collector é controlado pela classe *GC*, onde temos os métodos:

- **enable** - Habilita o garbage collector
- **disable** - Desabilita o garbage collector
- **start** - Inicia a "coleta de lixo"

A não ser que você **saiba muito bem o que está fazendo**, não é bom alterar o modo automático do garbage collector e usar o método **disable** para desabilitá-lo.

15.3 Como funciona, na prática

Agora vamos dar uma olhada em como funciona o garbage collector na prática. Vamos criar uma classe *Carro*, com uma propriedade *cor*, só para exemplificar, e usar o módulo **ObjectSpace** para listarmos nossos objetos na memória:

```
class Carro
   attr_reader :cor
   def initialize(c)
      @cor=c
   end
end

# criando os objetos
c1 = Carro.new("azul")
c2 = Carro.new("vermelho")
c3 = Carro.new("preto")

# verificando todos que são Carro's
c = ObjectSpace.each_object(Carro) {|o| p o}
puts "#{c} carros encontrados"

# invalidando um dos objetos
c2 = nil

# chamando o garbage collector
GC.start

# verificando de novo
c = ObjectSpace.each_object(Carro) {|o| p o}
puts "#{c} carros encontrados"
\section{Otimizando}
```

Rodando o programa, vamos ter:

```
#<Carro:0xb7e4d688 @cor="preto">
#<Carro:0xb7e4d6b0 @cor="vermelho">
#<Carro:0xb7e4d6d8 @cor="azul">
3 carros encontrados
#<Carro:0xb7e4d688 @cor="preto">
#<Carro:0xb7e4d6d8 @cor="azul">
2 carros encontrados
```

Viram o que aconteceu? Eu invalidei *c2* atribuindo **nil** á referência, e chamei o garbage collector. Imediatamente *c2* foi "marcada-e-varrida". Atribuindo **nil** á uma variável que aponta para um qualquer objeto que não é mais necessário faz dele um candidato a ser varrido na próxima vez que o garbage collector for acionado.

15.4 Otimizando

Você pode otimizar seu código para evitar muitas chamadas ao garbage collector. Por exemplo, usando o exemplo do *Carro*, vamos fazer alguns testes:

```
class Carro
   attr_accessor :velocidade
   def initialize(v)
      @velocidade=v
```

```
   end
   def mostra_velocidade
      s = "Carro andando a #{@velocidade}km/h,"
      case @velocidade
      when 0...60
         s += "lento."
      when 60...100
         s += "rápido!"
      else
         s += "muito rápido!"
      end
   end
end

0.step(120,60) do |v|
   carro = Carro.new(v)
   puts carro.mostra_velocidade
end
c = ObjectSpace.each_object(Carro) {|o| p o}
puts "#{c} carros encontrados"
```

Resulta em:

```
Carro andando a 0km/h,lento.
Carro andando a 60km/h,rápido!
Carro andando a 120km/h,muito rápido!
#<Carro:0xb7e4d430 @velocidade=120>
#<Carro:0xb7e4d4a8 @velocidade=60>
#<Carro:0xb7e4d4bc @velocidade=0>
3 carros encontrados
```

Se repararmos, não usamos os objetos criados no loop para mais nada a não ser mostrar a velocidade e um comparativo da mesma, mas não seria melhor se usarmos:

```
class Carro
   attr_accessor :velocidade
   def initialize(v)
      @velocidade=v
   end
   def mostra_velocidade
      s = "Carro andando a #{@velocidade}km/h,"
      case @velocidade
      when 0...60
         s += "lento."
      when 60...100
         s += "rápido!"
      else
         s += "muito rápido!"
      end
   end
end

carro = Carro.new(0)
0.step(120,60) do |v|
   carro.velocidade = v
   puts carro.mostra_velocidade
end
c = ObjectSpace.each_object(Carro) {|o| p o}
```

```
puts "#{c} carros encontrados"
```

Que resulta em:

```
Carro andando a 0km/h,lento.
Carro andando a 60km/h,rápido!
Carro andando a 120km/h,muito rápido!
#<Carro:0xb7e4d55c @velocidade=120>
1 carros encontrados
```

Já que não precisamos dos objectos distintos, podemos criar **um só objeto** de *Carro* e a utilizarmos para fazer o que precisamos. Pode parecer óbvio, mas é bom atentar para esse tipo de coisa.

Nesse exemplo, economizamos apenas dois *Carros*, mas imaginem se esse loop fosse até 300 (vai que é um Fórmula 1, acelera, Ayrton!), seria um belo de um ganho, e com certeza o garbage collector ia ser chamado mais tarde e bem menos vezes.

Capítulo 16

Unit Testing

Em primeiro lugar gostaria de agradecer ao **Guaracy Monteiro** (http://cadafalso.deusexmachina.com.br/) pelos exemplos que ele mandou para esse capítulo, onde eu adicionei mais código e texto. Ele prometeu não me processar e eu tenho uma declaração dele sobre isso. :-)

"Testes unitários" são um meio de testar e depurar pequenas partes do seu código, para verificar se não tem alguma coisa errada acontecendo lá, "modularizando" a checagem de erros. Um sistema é feito de várias "camadas" ou "módulos", e os testes unitários tem que ser rodados nessas camadas.

16.1 A classe a ser testada

Vamos fazer um exemplo simples, uma calculadora, no arquivo *Calculadora.rb*:

```
class Calculadora
   def soma(a,b)
      a+b
   end
   def subtrai(a,b)
      b-a
   end
   def multiplica(a,b)
      a*b
   end
   def divide(a,b)
      a/b
   end
end
```

Uma classe bem básica, com os métodos **soma**,**subtrai**,**multiplica** e **divide**.

16.2 Unidade de testes

Vamos fazer agora a unidade de testes, no arquivo *TC_Calculadora.rb*:

```
require 'test/unit'
require 'Calculadora'

class TC_Calculadora < Test::Unit::TestCase
   def setup
      @clc = Calculadora.new
```

```
    end
    def test_soma
       assert_equal(2 ,@clc.soma(1,1),"1 + 2 = 2")
    end
    def test_subtrai
       assert_equal(3 ,@clc.subtrai(7,4),"7 - 4 = 3")
       assert_equal(-3,@clc.subtrai(4,7),"4 - 7 =-3")
    end
    def test_multiplica
       assert_equal(4 ,@clc.multiplica(2,2),"2 x 2 = 4")
    end
    def test_divide
       assert_equal(2 ,@clc.divide(4,2),"4 / 2 = 2")
    end
    def banana
       puts 'bla'
    end
    def teardown
       @clc = nil
    end
end
```

Vamos dar uma olhada no que acontece ali.

Logo de início, tenho o método **setup**. Esse método é como se fosse, digamos, o "construtor" do teste. Tudo o que precisarmos inicializar para o teste, vai ser ali. No caso, criamos uma variável de classe com um objeto Calculadora.

Do mesmo modo que temos um "construtor", temos um "destrutor", que é **teardown**. Ali fechamos todos os recursos que não precisamos mais.

Logo depois temos alguns métodos, **test_soma**, **test_subtrai**, **test_multiplica**, **test_divide** e . . . **banana**???
Inseri esse método só para mostrar que o módulo de unit testing vai executar todos os métodos **test_***, e não o **banana**. Vamos ver isso daqui a pouco.
Dentro dos métodos temos as asserções que testam uma condição. Utilizamos apenas **assert_equal** , que apenas testa se o resultado indicado como primeiro parâmetro é igual ao resultado do que foi invocado no segundo parâmetro, indicando se houve falha com a mensagem do terceiro parâmetro.

Temos mais várias outras asserções, entre as quais:

- assert_nil
- assert_not_nil
- assert_not_equal
- assert_instance_of
- assert_kind_of
- assert_match
- assert_no_match
- assert_same
- assert_not_same

Agora, vamos rodar o teste:

```
[taq@~]ruby TC_Calculadora.rb
Loaded suite TC_Calculadora
Started
...F
Finished in 0.013689 seconds.

  1) Failure:
test_subtrai(TC_Calculadora) [TC_Calculadora.rb:12]:
7 - 4 = 3.
<3> expected but was
<-3>.

4 tests, 4 assertions, 1 failures, 0 errors
```

Xiii. Conseguimos nosso primeiro erro e, onde esperávamos 7-4=3 resultou em -3.

Antes de mais nada, olhem a contagem dos testes ali: **4 tests**. Se contamos os testes começados com **test_** na classe, temos exatamente 4, então o **banana** está ali para nada. Vamos eliminá-lo da classe.

Agora vamos para a nossa classe e verificamos o método **subtrai**, que foi onde ocorreu a falha.
Após alguns minutos (horas?) analisando, descobrimos que trocamos as bolas, ou melhor, a ordem dos parâmetros.

A resolução é simples, basta trocar o retorno de *b-a* para *a-b*.
Após as alterações executamos o teste novamente e obtemos:

```
[taq@~]ruby TC_Calculadora.rb
Loaded suite TC_Calculadora
Started
....
Finished in 0.000749 seconds.

4 tests, 5 assertions, 0 failures, 0 errors
```

Uau. Nenhum erro ou falha. Podemos considerar o programa como funcional e sem erros.

16.3 Falhas nos testes

Quando um programa passa por todos os testes, não significa que ele não tenha erros.
Apenas significa que passou nos testes que fizemos. No caso da divisão, o denominador não pode ser zero.

Aqui posso tomar diversas decisões. A mais simples seria apenas confirmar se um determinado procedimento gera uma exceção previsível.

Esta exceção pode ser gerada pela linguagem ou por alguma rotina que eu tenha implementado.
Vamos testar a divisão por 0, alterando *TC_Calculadora.rb*:

```
require 'test/unit'
require 'Calculadora'

class TC_Calculadora < Test::Unit::TestCase
   def setup
      @clc = Calculadora.new
   end
   def test_soma
      assert_equal(2 ,@clc.soma(1,1),"1 + 2 = 2")
```

```
   end
   def test_subtrai
      assert_equal(3 ,@clc.subtrai(7,4),"7 - 4 = 3")
      assert_equal(-3,@clc.subtrai(4,7),"4 - 7 =-3")
   end
   def test_multiplica
      assert_equal(4 ,@clc.multiplica(2,2),"2 x 2 = 4")
   end
   def test_divide
      assert_equal(2 ,@clc.divide(4,2),"4 / 2 = 2")
      assert_equal(0 ,@clc.divide(8,0),"8 / 0 = ?")
   end
   def teardown
      @clc = nil
   end
end
```

Rodando:

```
[taq@~]ruby TC_Calculadora.rb
Loaded suite TC_Calculadora
Started
E...
Finished in 0.001188 seconds.

  1) Error:
test_divide(TC_Calculadora):
ZeroDivisionError: divided by 0
    ./Calculadora.rb:12:in `/'
    ./Calculadora.rb:12:in `divide'
    TC_Calculadora.rb:20:in `test_divide'

4 tests, 5 assertions, 0 failures, 1 errors
```

Olha a divisão por 0 lá. Para evitarmos isso, podemos usar:

```
require 'test/unit'
require 'Calculadora'

class TC_Calculadora < Test::Unit::TestCase
   def setup
      @clc = Calculadora.new
   end
   def test_soma
      assert_equal(2 ,@clc.soma(1,1),"1 + 2 = 2")
   end
   def test_subtrai
      assert_equal(3 ,@clc.subtrai(7,4),"7 - 4 = 3")
      assert_equal(-3,@clc.subtrai(4,7),"4 - 7 =-3")
   end
   def test_multiplica
      assert_equal(4 ,@clc.multiplica(2,2),"2 x 2 = 4")
   end
   def test_divide
      assert_equal(2 ,@clc.divide(4,2),"4 / 2 = 2")
      assert_raise(ZeroDivisionError) do
         @clc.divide(8,0)
      end
```

```
    end
    def teardown
       @clc = nil
    end
end
```

Rodando:

```
[taq@~]ruby TC_Calculadora.rb
Loaded suite TC_Calculadora
Started
....
Finished in 0.001071 seconds.

4 tests, 6 assertions, 0 failures, 0 errors
```

Interceptamos a divisão por 0 e o teste funcionou corretamente.

É importante testar se o código está tratando os erros corretamente. Se usarmos **NoMethodError** em vez de **ZeroDivisionError**, o teste irá falhar e, uma mensagem como a que segue seria mostrada:

```
[taq@~]ruby TC_Calculadora.rb
Loaded suite TC_Calculadora
Started
F...
Finished in 0.053883 seconds.

  1) Failure:
test_divide(TC_Calculadora) [TC_Calculadora.rb:20]:
<NoMethodError> exception expected but was
Class: <ZeroDivisionError>
Message: <"divided by 0">
---Backtrace---
./Calculadora.rb:12:in `/'
./Calculadora.rb:12:in `divide'
TC_Calculadora.rb:21:in `test_divide'
TC_Calculadora.rb:20:in `assert_raise'
TC_Calculadora.rb:20:in `test_divide'
---------------

4 tests, 6 assertions, 1 failures, 0 errors
```

16.4 Novas funcionalidades

Até agora, a nossa classe está funcionando corretamente. Mas qual seria o procedimento para incluir novas funcionalidades? Não tem mistério.

É o mesmo utilizado até agora. Incluímos os novos testes, incluímos os procedimentos necessários e executamos o programa para testar.

Aqui vale salientar que todos os testes serão executados novamente e isto é apenas mais um indicador da qualidade do nosso código. Existem situações onde a inclusão de uma nova facilidade pode gerar efeitos colaterais e causar um erro em alguma parte que já foi previamente testada.

Manualmente isto é um sacrifício e induz a que o programador não refaça os testes anteriores.
Com a automatização, localizamos os problemas no momento em que foram introduzidos o que simplifica a correção dos mesmos (é muito mais difícil localizar e corrigir um problema que foi introduzido há meses

ou por outro programador relapso).

O lema é **"Não conviva com janelas quebradas"**. Quando alguma coisa quebrar, conserte imediatamente ou poderá chegar a um ponto onde fica mais fácil e econômico refazer todo o programa que ficar catando e consertando problemas.

Se quisermos incluir um método que retorne o resto de uma divisão, o procedimento seria, primeiro em *Calculadora.rb*:

```
class Calculadora
   def soma(a,b)
      a+b
   end
   def subtrai(a,b)
      a-b
   end
   def multiplica(a,b)
      a*b
   end
   def divide(a,b)
      a/b
   end
   def resto(a,b)
      a%b
   end
end
```

e depois em *TC_Calculadora.rb*:

```
require 'test/unit'
require 'Calculadora'

class TC_Calculadora < Test::Unit::TestCase
   def setup
      @clc = Calculadora.new
   end
   def test_soma
      assert_equal(2 ,@clc.soma(1,1),"1 + 2 = 2")
   end
   def test_subtrai
      assert_equal(3 ,@clc.subtrai(7,4),"7 - 4 = 3")
      assert_equal(-3,@clc.subtrai(4,7),"4 - 7 =-3")
   end
   def test_multiplica
      assert_equal(4 ,@clc.multiplica(2,2),"2 x 2 = 4")
   end
   def test_divide
      assert_equal(2 ,@clc.divide(4,2),"4 / 2 = 2")
   assert_raise(ZeroDivisionError) do
         @clc.divide(8,0)
      end
   end
   def test_resto
      assert_equal(2,@clc.resto(8,3),"8 % 3 = 2")
      assert_raise(ZeroDivisionError) do
         @clc.resto(5,0)
      end
   end
```

```
    def teardown
       @clc = nil
    end
end
```

Ok. Basta executar o programa de testes para ver se está tudo funcionando.

Fica a dica que podemos executar apenas um dos testes se quisermos:

```
[taq@~]ruby TC_Calculadora.rb --name test_resto
Loaded suite TC_Calculadora
Started
.
Finished in 0.000891 seconds.

1 tests, 2 assertions, 0 failures, 0 errors
```

No exemplo, executei apenas **test_resto**, que foi o último teste que inserimos.

Sempre verifique se o programa passou nos testes após cada alteração (quem fará os testes é o computador mesmo, então não existe motivo para preguiça).

Apesar do exemplo ser bem simples, acho que foi possível mostrar um pouco das vantagens da automatização dos testes. Na realidade, outros testes deveriam ser feitos como tipos de parâmetros incorretos *soma("7",23)*, *divide(8,"T")*, etc para testarmos não só os acertos mas também como os procedimentos se comportam com dados inconsistentes.

Capítulo 17

XML-RPC

XML-RPC é, segundo a descrição em seu site (http://www.xmlrpc.com/):

> "É uma especificação e um conjunto de implementações que permitem á softwares rodando em sistemas operacionais diferentes, rodando em diferentes ambientes, fazerem chamadas de procedures pela internet.
> A chamada de procedures remotas é feita usando HTTP como transporte e XML como o encoding. XML-RPC é desenhada para ser o mais simples possível, permitindo estruturas de dados completas serem transmitidas, processadas e retornadas."

Tentando dar uma resumida, você pode escrever métodos em várias linguagens rodando em vários sistemas operacionais e acessar esses métodos através de várias linguagens e vários sistemas operacionais. :-)

Vamos usar como exemplo uma adaptação do exemplo do XML-RPC HowTo (http://xmlrpc-c.sourceforge.net/xmlrpc-howto/xmlrpc-howto.html). Um método que permita somar dois inteiros e retornar a sua diferença, subtraindo-os. Também vamos fazer outro que faz a mesma coisa mas os subtrai na ordem inversa.

Para nosso exemplo em Ruby, vamos ter que instalar o pacote **xmlrpc4r** (http://www.fantasy-coders.de/ruby/xmlrpc4r/).

17.1 Servidor

Antes de mais nada, vamos fazer um servidor para armazenarmos nossos métodos. Vamos criar o arquivo **xmlrpc_server.rb**:

```
require "xmlrpc/server"

s = XMLRPC::Server.new(8081)
s.add_handler("ab.sum_and_difference") do |a,b|
   {"sum" => a+b, "difference" => a-b}
end
s.add_handler("ba.sum_and_difference") do |a,b|
   {"sum" => a+b, "difference" => b-a}
end
s.serve
```

Mais simples, impossível. :-) Vale notar que o servidor está sendo criado "standalone", rodando na porta 8081, ao invés de um server CGI. Para ver como criar um CGI, consulte o link acima.

Logo após criar o servidor, foram inseridos handlers para os métodos, através de blocos de código. Criamos uma procedure chamada *ab.sum_and_difference* e outra chamada *ba.sum_and_difference*. Apesar de parecerem *namespaces*, *ab* e *ba* são apenas sufixos que usamos para diferenciar os métodos, na ordem

da subtração. Nos métodos são retornados *hashes* com o resultado da soma e da subtração.

Para rodar o servidor:

```
[taq@~]ruby xmlrpc_server.rb
[Thu Jul 14 11:25:12 2005] HttpServer 127.0.0.1:8081 start
```

17.2 Cliente

Agora vamos fazer o cliente, **xmlrpc_client.rb**, que também é muito simples:

```
require "xmlrpc/client"

begin
   s = XMLRPC::Client.new("localhost","/RPC2",8081)

   r = s.call("ab.sum_and_difference", 5, 3)
   puts "Soma: #{r['sum']} Diferença: #{r['difference']}"

   r = s.call("ba.sum_and_difference", 5, 3)
   puts "Soma: #{r['sum']} Diferença: #{r['difference']}"
rescue XMLRPC::FaultException => e
   puts "Erro: #{e.faultCode} #{e.faultString}"
end
```

Pedimos para abrir uma conexão com o servidor no host local, na porta 8081 (o "/RPC2" é um parametro default para quando não é utilizado um servidor CGI) e para chamar (*call*) as procedures *ab.sum_and_difference* e *ba.sum_and_difference*, passando como parâmetros para ambas 5 e 3. Rodando nosso programa temos:

```
[taq@~]ruby xmlrpc_client.rb
Soma: 8 Diferença: 2
Soma: 8 Diferença: -2
```

Simples, não?

17.3 Acessando de outras linguagens

Vamos ver o cliente agora em PHP. Para rodar em PHP vamos precisar do **XML-RPC for PHP** , disponível em http://phpxmlrpc.sourceforge.net/:

```
<?php
   ini_set("include_path",ini_get("include_path").":/home/taq/scripts/xmlrpc/");
   include "xmlrpc.inc";
   $server = new xmlrpc_client("/RPC2","localhost",8081);

   $msg = new xmlrpcmsg("ab.sum_and_difference",
                  array(new xmlrpcval(5,"int"),new xmlrpcval(3,"int")));
   $rst = $server->send($msg);

   if($rst && !$rst->faultCode()){
      $struct = $rst->value();
      $sumval = $struct->structmem("sum");
      $sum    = $sumval->scalarval();
      $difval = $struct->structmem("difference");
      $dif    = $difval->scalarval();
      print "Soma: $sum Diferença: $dif\n";
```

```
   }

   $msg = new xmlrpcmsg("ba.sum_and_difference",
                  array(new xmlrpcval(5,"int"),new xmlrpcval(3,"int")));
   $rst = $server->send($msg);

   if($rst && !$rst->faultCode()){
      $struct = $rst->value();
      $sumval = $struct->structmem("sum");
      $sum    = $sumval->scalarval();
      $difval = $struct->structmem("difference");
      $dif    = $difval->scalarval();
      print "Soma: $sum Diferença: $dif\n";
   }
?>
```

Ahn ... ficou maiorzinho né? Mas rodando temos o mesmo resultado:

```
[taq@~]php xmlrpc_client.php
Soma: 8 Diferença: 2
Soma: 8 Diferença: -2
```

Em Python:

```
import xmlrpclib

server = xmlrpclib.Server("http://localhost:8081")

result = server.ab.sum_and_difference(5,3)
print "Soma:",result["sum"]," Diferença:",result["difference"]

result = server.ba.sum_and_difference(5,3)
print "Soma:",result["sum"]," Diferença:",result["difference"]
```

Rodando (ele dá um aviso sobre a xmllib estar obsoleta, mas isso é outra história, para algum tutorial de Python ;-):

```
[taq@~]python xmlrpc_client.py
Soma: 8  Diferença: 2
Soma: 8  Diferença: -2
```

Em Java vamos precisar do Apache XML-RPC (http://ws.apache.org/xmlrpc/). Eu usei a versão 1.2-b1, por que a 2 pede mais pacotes para rodar.
Vamos ver como fica, criando o arquivo **RPCClient.java**:

```
import java.util.Vector;
import java.util.Hashtable;
import org.apache.xmlrpc.*;

public class RPCClient {
   public static void main(String args[]){
      try{
         XmlRpcClient server = new XmlRpcClient("http://localhost:8081/RPC2");

         Vector params = new Vector();
         params.addElement(new Integer(5));
         params.addElement(new Integer(3));

         Hashtable result = (Hashtable) server.execute("ab.sum_and_difference",params);
```

```
            int sum = ((Integer) result.get("sum")).intValue();
            int dif = ((Integer) result.get("difference")).intValue();
            System.out.println("Soma: "+Integer.toString(sum)+
                               " Diferença: "+Integer.toString(dif));

            result = (Hashtable) server.execute("ba.sum_and_difference",params);
            sum = ((Integer) result.get("sum")).intValue();
            dif = ((Integer) result.get("difference")).intValue();
            System.out.println("Soma: "+Integer.toString(sum)+
                               " Diferença: "+Integer.toString(dif));
         }catch(Exception error){
            System.err.println("erro:"+error.getMessage());
         }
      }
   }
```

Rodando:

```
[taq@~]java -classpath .:xmlrpc-1.2-b1.jar: RPCClient
Soma: 8 Diferença: 2
Soma: 8 Diferença: -2
```

Índice Remissivo