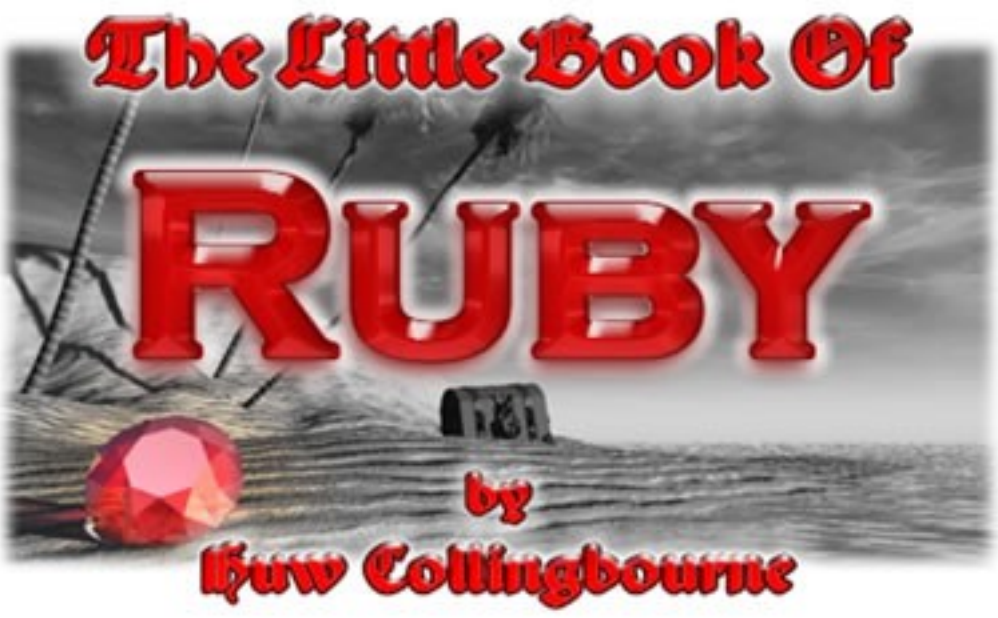
PEQUENO LIVRO DO RUBY



SEU GUIA LIVRE PARA PROGRAMAÇÃO EM RUBY
DE WWW.SAPPHIRESTEEL.COM
EM ASSOCIAÇÃO COM **BITWISE MAGAZINE**

Traduzido para o português do Brasil

por Francisco de Oliveira

SISMICRO Informática Ltda.

www.sismicro.com.br

The Little Book Of Ruby

written by
Huw Collingbourne
Download source code from:
http://www.sapphiresteel.com

First edition: June 2006

The Little Book of Ruby is produced in association with Rosedown Mill Ltd., makers of the Steel IDE for Visual Studio (www.sapphiresteel.com) and Bitwise Magazine – the online magazine for developers (www.bitwisemag.com).

Tradução:
SISMICRO Informática Ltda
www.sismicro.com.br

Junho/2006

Obs: Algumas palavras e, principalmente, os fontes programas foram mantidos na língua original – consideramos que estes casos não prejudicarão o entendimento do texto.

Índice

Bem-vindo ao Pequeno Livro do Ruby

Aprenda Ruby em Dez Capítulos...

1. Strings e Métodos
2. Classes e Objetos
3. Hierarquia de Classes
4. Acessores, Atributos, Variáveis de Classe
5. Arrays
6. Hashes
7. Laços e Iteradores
8. Declarações Condicionais
9. Módulos e Mixins
10. Salvando Arquivos, Avançando ...

O que é Ruby?

Ruby é uma linguagem de programação interpretada multi plataforma que tem muitas características em comum com outras linguagens de script como Perl e Phyton. Contudo a sua versão de orientação a objetos é mais completa que aquelas linguagens e, em muitos aspectos, o Ruby tem mais em comum com o bisavô das linguagens OOP (programação orientada a objetos) 'puras', o Smalltalk. A linguagem Ruby foi criada por Yukihiro Matsumoto (comumente conhecido por ‘Matz’) e foi liberada pela primeira vez em 1995.

O que é Rails?

Atualmente, muito da empolgação em volta do Ruby pode ser atribuída ao framework de desenvolvimento web chamado Rails – popularmente conhecido como ‘Ruby On Rails’. Embora o Rails seja um framework impressionante, ele não é a coisa mais importante do Ruby. A bem da verdade, se você decidir mergulhar fundo no desenvolvimento com Rails sem antes dominar o Ruby, você poderá acabar fazendo uma aplicação que nem você mesmo entende. Ainda que este Pequeno Livro do Ruby não irá cobrir as características especiais do Rails, ele lhe dará a base necessária para você entender o código do Rails e para escrever suas próprias aplicações Rails.

Baixe o Ruby mais um Editor de Código

Você pode baixar a última versão do Ruby em www.ruby-lang.org. Esteja certo de baixar os binários (não somente o código fonte). A forma mais simples de ter o Ruby instalado em um PC rodando o Windows é usar o instalador para Windows: http://rubyinstaller.rubyforge.org/wiki/wiki.pl que inclui o editor de código SciTE. Usuários do Visual Studio 2005 podem baixar gratuitamente nosso editor e depurador Visual Studio Ruby In Steel de: www.sapphiresteel.com.

Pegue o Código Fonte dos Programas Exemplo

Todos os programas de todos os capítulos deste livro estão disponíveis para baixar de www.sapphiresteel.com no formato zipado. Quando você descompactar estes programas você verá que eles estão agrupados em um conjunto de diretórios – um para cada capítulo. Se você estiver usando Ruby In Steel, você poderá carregar todos os programas como uma única solução, com os programas de cada capítulo arranjado na forma de uma árvore no Gerenciador de Projetos.

Rodando Programas Ruby

É útil deixar uma janela de comandos (Command Prompt) aberta no diretório contendo os programas fontes. Assumindo que o seu interpretador Ruby está corretamente configurado no caminho de busca do seu Windows (path), você poderá a rodar os programas digitando ruby <nome do programa> assim:
ruby 1helloworld.rb

Se você está usando Ruby In Steel você pode rodar os seus programas interativamente pressionando CTRL+F5 ou executá-lo no depurador teclando F5.

Como usar este livro

Este livro é um tutorial passo-a-passo para programar em Ruby e você pode segui-lo capítulo a capítulo, lendo o texto e rodando os programas exemplo. Por outro lado, se você preferir encurtar a história, você pode testar alguns programas na ordem que lhe convier, então retornar ao texto para o esclarecimento do código ou da sintaxe usada. Não há nenhuma aplicação monolítica neste livro,

apenas pequenos programas exemplo – então é fácil pular de um capítulo para outro se assim você achar melhor.

Tomando conhecimento do texto

Neste livro, qualquer código fonte Ruby está escrito como este:

```
def saysomething
        puts( "Hello" )
end
```

Quando existir um programa exemplo acompanhando o código, o nome do programa será mostrado em uma pequena caixa como esta. helloname.rb

Notas explicativas (as quais geralmente fornecem alguma sugestão, dica ou uma explicação aprofundada de algum ponto mencionado no texto) são mostrados em uma caixa como esta:

> Esta é uma nota explicativa. Você pode pulá-la se quiser – mas se você o fizer poderá perder algo interessante...!

> Nota do tradutor: Nós optamos por mostrar o código dos programas exemplo diretamente no texto do livro, o que não aconteceu na versão original – ao menos não de forma integra. Esperamos que isso ajude na compreensão. Em alguns casos pode ocorrer a duplicação de código – desde já pedimos desculpas.

Cap. 1 - Strings e métodos

Strings, Números, Métodos e Condicionais

Pelo fato de você estar lendo este livro, eu acho que é seguro deduzir que você quer programar em Ruby – e, se você for como eu, você estará impaciente para por a mão na massa. Ok, então não vamos ficar esperando, vamos ao trabalho. Eu assumirei que você já tem o Ruby instalado. Se não, você precisa fazer isto antes de mais nada, como explicado na Introdução...

Vamos começar a codificar. Abra o seu editor e escreva o seguinte:

puts 'hello world'

Salve o programa como 'helloworld.rb' e agora rode o programa (como explicado em 'Rodando Programas Ruby' visto anteriormente). Se tudo correu bem, o Ruby deveria mostrar “hello world”.

> Se você estiver usando um editor sem uma console interativa, você deverá rodar seus programas a partir do prompt de comandos do sistema operacional. Para isto abra um prompt de comandos (Windows) ou um terminal (Linux) e navegue até o diretório contendo o código fonte e então digite ruby seguido pelo nome do programa, assim: ruby helloworld.rb

uppercase.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

puts( "hello world".upcase )
```

Este deve ser o menor ‘hello world’ programa da história da programação, então vamos avançar modificando-o para pedir que o usuário digite algo ...
O próximo passo, então, é 'solicitar' ao usuário uma string (conjunto de caracteres). O método para fazer isso em Ruby é ***gets***.

helloname.rb

```
# Ruby Sample program from www.sapphiresteel.com /
```

www.bitwisemag.com

```
print('Enter your name: ' )
name = gets()
puts( "Hello #{name}" )
```

O programa helloname.rb pergunta ao usuário o seu nome – vamos supor que é “Fred” - e então mostra uma saudação: “Hello Fred”.

Enquanto isso ainda é muito simples, existem alguns detalhes importantes que precisam ser explicados. Primeiro, note que eu usei ***print*** em vez de ***puts*** para mostrar a pergunta. Isto foi feito porque o puts adiciona uma nova linha no final e o print não; neste exemplo eu quero que o cursor permaneça na mesma linha da pergunta.
Na linha seguinte eu usei o gets() para ler a informação digitada pelo usuário em uma string quando for pressionada a tecla Enter. Esta string é armazenada na variável, name. Eu não pré-declarei esta variável, nem especifiquei seu tipo. Em Ruby você pode criar variáveis como e quando precisar delas e o Ruby deduz o tipo correspondente. Neste caso eu atribui uma string para a variável name, logo o Ruby sabe que o tipo de name deve ser uma string.

Objetos e Métodos

Ruby é uma linguagem altamente OOP (Programação Orientada por Objetos). Tudo desde um inteiro até uma string é considerado um objeto. E cada objeto é constituído de 'métodos' os quais podem fazer muitas coisas. Para usar um método, você precisa colocar um ponto após o objeto, então adicionar o nome do método. Por exemplo, aqui eu estou usando o método ***upcase*** para mostrar a string “hello world” em maiúsculas:

puts("hello world".upcase)

Alguns métodos como o ***puts*** e o ***gets*** estão disponíveis em todo o lugar e não necessitam ser associados a um objeto específico. Tecnicamente falando, estes métodos são fornecidos pelo módulo Kernel do Ruby e eles estão presentes em todos os objetos Ruby. Quando você roda uma aplicação Ruby, um objeto chamado ***main*** é automaticamente criado e este objeto fornece acesso aos métodos do Kernel.

Nota: O Ruby é sensível a maiúsculas e minúsculas (caixa alta/baixa). Uma variável chamada myvar é diferente de outra chamada myVar. Uma variável como 'name' no nosso exemplo deve iniciar com uma letra minúscula (em caixa baixa)

O símbolo de parênteses no final de gets() é opcional assim como os parênteses que envolvem as strings após o print e o puts; o código deveria rodar igualmente se você removesse os parênteses. Contudo, o Ruby está movendo-se gradualmente na direção do uso dos parênteses – particularmente quando passa-se argumentos para métodos. Os parênteses ajudam a evitar possíveis ambigüidades no código e , em alguns casos, o interpretador do Ruby irá avisá-lo se você omiti-los. Enquanto alguns programadores Ruby gostam de omitir os parênteses sempre que possível, eu não sou um deles; você irá, entretanto, verificar o uso liberal de parênteses nos meus programas.

Strings e Avaliação embutida

A última linha no programa helloname.rb é bem interessante:

```
puts( "Hello #{name}" )
```

Aqui a variável name está embutida numa string. Isto é feito colocando a variável dentro do sinal de chaves {} precedido do caracter #, assim: #{variável}. Este tipo de avaliação embutida somente funciona com strings delimitadas por aspas (“”). E não é somente variáveis que podem ser embutidas entre as aspas. Você pode pode, também, embutir caracteres não imprimíveis como nova-linha “\n” e tabs “\t” em strings delimitadas por aspas. Você pode até embutir pequenos códigos de programa e expressões matemáticas. Vamos assumir que você tem um método chamado showname, que retorna a string ‘Fred’. A string seguinte deveria, no processo de avaliação, chamar o método showname e, como resultado, mostrar a string “Hello Fred”:

```
puts "Hello #{showname}"
```

string_eval.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

def showname
   return "Fred"
end

puts "Hello #{showname}"
puts( "\n\t#{(1+2) * 3}" )
```

Veja se você consegue acertar o que seria mostrado pelo código seguinte:

```
puts( "\n\t#{(1 + 2) * 3}" )
```

Execute o programa string_eval.rb para ver se você está certo.

> **Comentários...**
> Linhas que começam com o caracter # são tratadas como comentários (elas são ignoradas pelo interpretador Ruby):
> # This is a comment – Isto é um comentário

Métodos

No exemplo anterior, eu, sem barulho, introduzi um método Ruby sem explicar precisamente o que é e a sintaxe necessária para criá-lo. Vou corrigir essa omissão agora.

object.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

class MyClass
     def saysomething
          puts( "Hello" )
     end
end

ob = MyClass.new
ob.saysomething
```

Um método é assim chamado porque ele fornece um método (isto é, 'uma forma') para um objeto responder a mensagens. Na terminologia OOP, você envia uma mensagem para um objeto pedindo que ele faça algo. Vamos imaginar que você tem um objeto chamado ob o qual possui um método chamado saysomething, esta é a forma que você deveria enviar-lhe uma mensagem saysomething:

```
ob.saysomething
```

Vamos supor que o método saysomething seja o seguinte:

```
def saysomething
```

```
       puts( "Hello" )
end
```

Quando você envia a mensagem saysomething para o objeto ob ele responde com o método saysomething e mostra “Hello”.

OK, esta a é forma ‘OOP pura’ de descrever este processo. Uma forma OOP não tão pura de descrever isso seria dizer que saysomething é como uma função que é ligada ao objeto e pode ser chamada usando a “notação de ponto”:

```
ob.saysomething.
```

method.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

def showstring
     puts( "Hello" )
end

def showname( aName )
     puts( "Hello #{aName}" )
end

def return_name( aFirstName, aSecondName )
     return "Hello #{aFirstName} #{aSecondName}"
end

def return_name2 aFirstName, aSecondName
     return "Hello #{aFirstName} #{aSecondName}"
end

showstring
showname( "Fred" )
puts( return_name( "Mary Mary", "Quite-Contrary"  ) )
puts( return_name( "Little Jack", "Horner"  ) )
```

No Ruby um método é declarado com a palavra-chave ***def*** seguida do nome do método o qual deveria iniciar com uma letra minúscula, como este:

```
def showstring
      puts( "Hello" )
```

```
end
```

Você pode, opcionalmente, colocar um ou mais argumentos, separados por vírgula, após o nome do método:

```
def showname( aName )
      puts( "Hello #{aName}" )
end

def return_name( aFirstName, aSecondName )
      return "Hello #{aFirstName} #{aSecondName}"
end
```

Os parênteses em volta dos argumentos são opcionais. A seguinte sintaxe também é permitida:

```
def return_name2 aFirstName, aSecondName
      return "Hello #{aFirstName} #{aSecondName}"
end
```

Como explicado anteriormente, eu sou mais favorável a usar os parênteses mas você pode omiti-los se quiser.

mainob.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

def showstring
     puts( "Hello" )
end

def showname( aName )
     puts( "Hello #{aName}" )
end

def return_name( aFirstName, aSecondName )
     return "Hello #{aFirstName} #{aSecondName}"
end

def return_name2 aFirstName, aSecondName
     return "Hello #{aFirstName} #{aSecondName}"
end
```

```
# so which object owns these methods, anyhow?
# The following test reveals all...
print( "The 'free standing methods' in this code belong to an
object named: " )
puts( self )
print( "which is an instance of the class: " )
puts( self.class )

#Free-standing methods (like those above which are not defined
within a
#specific class) are methods (strictly speaking, 'private'
methods) of
#the main object which Ruby creates automtically. The following
code
#displays a list of the main object's private methods. Look
carefully and
#you will find showname, return_name and return_name2 in that list
puts( "It contains these private methods: " )
puts( self.private_methods )
```

Se os métodos se ligam aos objetos, qual objeto possui os métodos “independentes” que você escreve no seu código? Como mencionado antes, o Ruby cria automaticamente um objeto chamado main quando você executa um programa e é a esse objeto que os métodos “independentes” se ligam.

Números

Números são tão fáceis de usar quanto as strings. Por exemplo, vamos supor que você quer calcular o preço de venda e o total geral de alguns itens partindo do subtotal e da taxa de imposto.
Para fazer isto você precisaria multiplicar o subtotal pela taxa de imposto e adicionar o valor ao subtotal. Assumindo que o subtotal é $100 e a taxa de imposto é de 17.5%, este código Ruby faz o cálculo e mostra o resultado:

```
subtotal = 100.00
taxrate = 0.175
tax = subtotal * taxrate
puts "Tax on $#{subtotal} is $#{tax}, so grand total is
$#{subtotal+tax}"
```

Obviamente, seria mais útil se pudéssemos efetuar o cálculo de vários subtotais em vez de ficar

fazendo o mesmo cálculo toda vez.!.
Aqui está uma versão simples da "Calculadora de Impostos" que pergunta ao usuário o subtotal:

```
taxrate = 0.175
print "Enter price (ex tax): "
s = gets
subtotal = s.to_f
tax = subtotal * taxrate
puts "Tax on $#{subtotal} is $#{tax}, so grand total is
$#{subtotal+tax}"
```

Aqui **s.to_f** é um método da classe String. Este método tenta converter a string para um número de ponto flutuante. Por exemplo, a string "145.45" seria convertida para o número de ponto flutuante 145.45. Se a string não pode ser convertida, 0.0 é retornado. Assim, por exemplo, "Hello world".to_f retornaria 0.0.

Testando uma Condição: ***if ... then***

O problema com o código da "Calculadora de Impostos" mostrado acima é que ele aceita subtotais negativos e calcula imposto negativo sobre eles – uma coisa que o Governo não vê com bons olhos! Eu, por essa razão, preciso checar os valores negativos e, quando encontrá-los, zerá-los. Esta é minha nova versão do código:

tax_calculator.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

taxrate = 0.175
print "Enter price (ex tax): "
s = gets
subtotal = s.to_f
if (subtotal < 0.0) then
     subtotal = 0.0
end
tax = subtotal * taxrate
puts "Tax on $#{subtotal} is $#{tax}, so grand total is
$#{subtotal+tax}"
```

O teste Ruby **if** é semelhante a um teste **if** em outras linguagens de programação . Note que os

parênteses, mais uma vez, são opcionais, assim como a palavra-chave **then**.
Entretanto, se você for escrever o seguinte, sem quebra de linha após a condição de teste, o **then** é obrigatório:

```
if (subtotal < 0.0) then subtotal = 0.0 end
```

Note que a palavra-chave **end** que termina o bloco if não é opcional. Esqueça de colocá-la e o seu código não irá rodar.

Cap. 2 - Classes e Objetos

Classes e Objetos

Até o momento nós temos usado objetos padrão do Ruby como números e strings. Vamos agora ver como criar novos tipos de objetos. Como na maioria das outras linguagens OOP, um objeto Ruby é definido por uma classe. A classe é como uma "modelo" a partir da qual objetos individuais são construídos. Esta é uma classe bem simples:

```
class MyClass
end
```

E assim é como eu criaria um objeto utilizável a partir dela:

```
ob = MyClass.new
```

Note que eu não posso fazer muita coisa com o meu objeto – pela simples razão que eu não programei nada na minha classe MyClass, da qual o objeto foi criado.

> Na verdade, se você criar uma classe vazia como MyClass, os objetos criados a partir dela não são totalmente inúteis. Todas as classes Ruby automaticamente herdam as características da classe Object. Logo meu objeto ob pode fazer uso dos métodos de Object tal como class (que diz para um objeto mostrar a sua classe):
> `puts ob.class #=> displays: "MyClass"`

Para tornar MyClass um pouco mais útil, eu preciso dar-lhe um método ou dois. Neste exemplo (o qual foi mencionado brevemente no capítulo anterior), eu adicionei um método chamado saysomething:

```
class MyClass
        def saysomething
                puts( "Hello" )
        end
end
```

Agora, quando eu crio um objeto da classe MyClass, eu posso chamar este método para que o objeto diga "Hello":

object_class.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

class MyClass
     def saysomething
          puts( "Hello" )
     end
end

ob = MyClass.new
ob.saysomething
```

Instâncias e Variáveis de Instância

Vamos criar mais alguns objetos úteis. Nenhuma casa (ou programa de computador) deveria ficar sem um cachorro. Então vamos nós mesmos criar um classe Dog:

```
class Dog
        def set_name( aName )
                @myname = aName
        end
end
```

Note que a definição da classe começa com a palavra-chave **class** (toda em minúsculas) e é seguida do nome da classe propriamente, o qual deve iniciar com uma letra maiúscula. Minha classe Dog contém um único método, set_name. Este recebe um argumento de entrada, aName. O corpo do método atribue o valor de aName para uma variável chamada @myname.

Variáveis iniciadas com o caracter @ são 'variáveis de instância' que significa que elas se ligam a objetos individuais – ou 'instâncias' da classe. Não é necessário declarar previamente estas variáveis.

Eu posso criar instâncias da classe Dog (isto é, 'objetos dog') chamando o método new. Aqui eu estou criando dois objetos dog (lembre que o nome da classe inicia com letra maiúscula; nomes de objetos iniciam com letra minúscula):

```
mydog = Dog.new
yourdog = Dog.new
```

Neste momento, estes dois cachorros estão sem nome. Então a próxima coisa a fazer é chamar o método set_name para dar nomes a eles:

```
mydog.set_name( 'Fido' )
yourdog.set_name( 'Bonzo' )
```

Tendo dado nomes aos cachorros, eu preciso de alguma forma para achar os nomes mais tarde. Cada cachorro precisa saber o seu próprio nome, então vamos criar um método get_name:

```
def get_name
  return @myname
end
```

A palavra-chave **return** é opcional. Métodos Ruby sempre retornam a última expressão avaliada. Para maior clareza (e também para evitar resultados inesperados de métodos mais complexos que este!) devemos adquirir o hábito de retornar explicitamente o valor para uso posterior. Finalmente, vamos dar algum comportamento ao cachorro pedindo para ele falar (isto é: latir). Aqui está a definição final da classe:

```
class Dog
  def set_name( aName )
    @myname = aName
  end
  def get_name
    return @myname
  end
  def talk
    return 'woof!'
  end
end
```

Agora, nós podemos criar um cachorro, dar-lhe um nome, mostrar o nome e pedir para ele “latir”:

```
mydog = Dog.new
mydog.set_name( 'Fido' )
puts(mydog.get_name)
puts(mydog.talk)
```

Para maior clareza – e para mostrar eu não estou contra nossos amigos felinos - Eu também adicionei uma classe Cat no meu programa, dogs_and_cats.rb. A classe Cat é semelhante à classe Dog exceto pelo fato que o método talk, naturalmente, retorna um "miaow" em vez de "woof".

Este programa contém um erro. O objeto de nome someotherdog nunca tem um valor atribuido para a sua variável @name. Por sorte, o Ruby não explode quando nós tentamos mostrar o nome do cachorro. Em vez disso o Ruby, simplesmente, imprime 'nil'.
Nós veremos em breve uma forma simples de assegurar que erros como esse não ocorram novamente...

dogs_and_cats.rb

```
# Ruby Sample program from www.sapphiresteel.com / www.bitwisemag.com
# Create classes and use instance variables such as @myname

class Dog
     def set_name( aName )
          @myname = aName
     end

     def get_name
           return @myname
     end

     def talk
          return 'woof!'
     end
end


class Cat
     def set_name( aName )
          @myname = aName
     end

     def get_name
           return @myname
```

```
     end

     def talk
          return 'miaow!'
     end
end

# --- Create instances (objects) of the Dog and Cat classes
mydog = Dog.new
yourdog = Dog.new
mycat = Cat.new
yourcat = Cat.new
someotherdog = Dog.new

# --- Name them
mydog.set_name( 'Fido' )
yourdog.set_name( 'Bonzo' )
mycat.set_name( 'Tiddles' )
yourcat.set_name( 'Flossy' )


# --- Get their names and display them
# Dogs
puts(mydog.get_name)
puts(yourdog.get_name)
# hmmm, but what happens here if the dog has no name?
puts(someotherdog.get_name)
# Cats
puts(mycat.get_name)
puts(yourcat.get_name)

# --- Ask them to talk
puts(mydog.talk)
puts(yourdog.talk)
puts(mycat.talk)
puts(yourcat.talk)
```

Construtores – new e initialize

Agora, vamos olhar um outro exemplo de uma classe definida pelo usuário. Carregue o programa treasure.rb.

treasure.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com
# define a class and create some objects

class Thing
          def set_name( aName )
               @name = aName
          end

          def get_name
               return @name
          end
end

class Treasure
      def initialize( aName, aDescription )
        @name         = aName
        @description  = aDescription
      end

      def to_s # override default to_s method
           "The #{@name} Treasure is #{@description}\n"
      end
end

thing1 = Thing.new
thing1.set_name( "A lovely Thing" )
puts thing1.get_name

t1 = Treasure.new("Sword", "an Elvish weapon forged of gold")
t2 = Treasure.new("Ring", "a magic ring of great power")
puts t1.to_s
puts t2.to_s
# The inspect method lets you look inside an object
puts "Inspecting 1st treasure: #{t1.inspect}"
```

Este é um jogo de aventura em construção. Contém duas classes, Thing e Treasure. A classe Thing é bem similar à classe Dog vista anteriormente – bem, exceto pelo fato que ela não late ('woof'). Já a classe Treasure tem alguns extras interessantes. Primeiramente, ela não tem os métodos get_name e set_name. No lugar , ela contém um método chamado initialize que recebe dois argumentos cujos valores são atribuidos para as variáveis @name e @description:

```
def initialize( aName, aDescription )
  @name               = aName
  @description        = aDescription
end
```

Quando uma classe contém um método de nome ***initialize***, este método é automaticamente chamado quando um objeto é criado usando o método ***new***. É uma boa idéia usar um método initialize para definir os valores das variáveis de instância de um objeto. Isto tem dois benefícios claros sobre definir as variáveis usando métodos como set_name. Primeiro, uma classe complexa poder conter numerosas variáveis de instância. Depois, eu criei um método chamado **to_s** o qual serve para retornar uma string representativa de um objeto da classe Treasure. O nome do método, to_s, não é arbitrário. O mesmo nome de método é usado extensivamente pela hierarquia de classes padrão do Ruby. De fato, o método to_s é definido para a própria classe Object a qual é a última ancestral de todas as outras classes em Ruby. Redefinindo o método to_s, eu adicionei um novo comportamento que é mais apropriado para a classe Treasure que o método padrão. Em outras palavras, nós **sobrescrevemos** o método ***to_s.***

> Nota: O método ***new*** cria um objeto logo você pode vê-lo como um construtor de objetos. Contudo, você não deveria normalmente implementar a sua própria versão do método new (isto é possível mas não é, geralmente, recomendado). Em vez disso, quando você quer executar quaisquer ações de "configuração" - como definir valores iniciais para variáveis internas de um objeto – você deveria fazer isso em um método chamado initialize. O Ruby executa o método initialize imediatamente após um novo obejto ser criado.

Inspecionando Objetos

Note também que eu "olhei internamente" o objeto Treasure, t1, usando o método ***inspect***:

```
t1.inspect
```

Este método inspect é definido para todos os objetos Ruby. Ele retorna uma string contendo uma

representação do objeto legível por nós humanos. No caso em questão, é mostrado algo assim:

```
#<Treasure:0x28962f8 @description="an Elvish weapon forged of
gold", @name="Sword">
```

Começa com o nome da classe, Treasure; seguido de um número, o qual pode ser diferente da mostrada acima – este é o código de identificação interna de um objeto particular do Ruby; então vem os nomes e valores das variáveis do objeto.

O Ruby também fornece o método p que é um atalho para inspecionar e mostrar objetos:
p(anobject)

p.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

class Treasure
      def initialize( aName, aDescription )
        @name         = aName
        @description  = aDescription
      end

      def to_s # override default to_s method
           "The #{@name} Treasure is #{@description}\n"
      end
end


a = "hello"
b = 123
c = Treasure.new( "ring", "a glittery gold thing" )

p( a )
p( b )
p( c )
```

Para ver como **to_s** pode ser usado com uma variedade de objetos e para testar como um objeto Treasure seria convertido para uma string na ausência de uma sobrescrição do método **to_s**, teste o programa to_s.rb .

to_s.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

# Show string representaions of various objects
# using the to_s method

class Treasure
      def initialize( aName, aDescription )
        @name         = aName
        @description  = aDescription
      end
# This time we won't override to_s so the Treasure object
# will use the default to_s method...
end


t = Treasure.new( "Sword", "A lovely Elvish weapon" )
print("Class.to_s: ")
puts(Class.to_s)
print("Object.to_s: ")
puts(Object.to_s)
print("String.to_s: ")
puts(String.to_s)
print("100.to_s: ")
puts(100.to_s)
print("Treasure.to_s: ")
puts(Treasure.to_s)
print("t.to_s: ")
puts(t.to_s)
print("t.inspect: ")
puts(t.inspect)
```

Como você verá, classes como Class, Object, String e Treasure, simplesmente retornam seus nomes quando o método to_s é chamado. Um objeto como o objeto Treasure, t, retorna seu identificador – que é o mesmo identificador retornado pelo método **inspect**.

Olhando para o meu programa treasure.rb verificamos que o código é um pouco repetitivo. Além do que, por que ter uma classe Thing que contém um nome (name) e uma classe Treasure que também contém um nome (a variável de instância @name), cada qual codificada independentemente? Faria mais sentido considerar Treasure com um 'tipo de Thing'. Se eu for desenvolver este programa

dentro de um completo jogo de aventura, outros objetos como Rooms e Weapons podem ser outros ‘tipos de Thing’. É chegada a hora de começar a trabalhar em uma hierarquia de classes apropriada. É o que veremos na próxima lição...

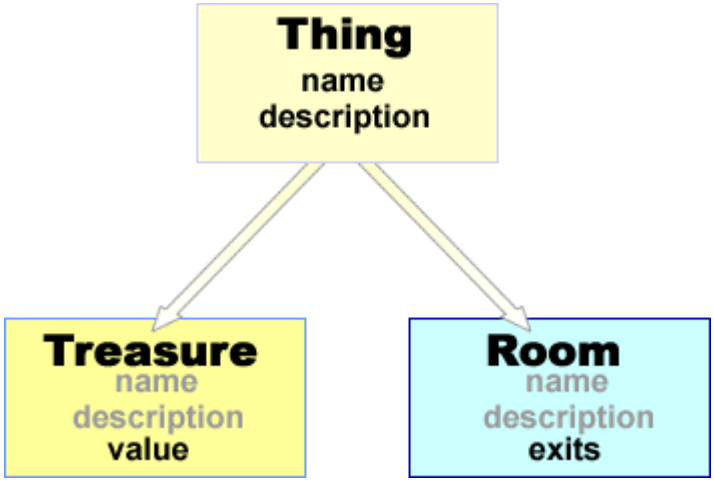
Cap. 3 - Hierarquia de Classes

Hierarquia de classes...

Nós terminamos a lição anterior criando duas novas classes: uma Thing e uma Treasure. Apesar destas duas classes compartilharem algumas características (notadamente ambas têm um 'nome'), não existe conexão entre elas. Agora, estas duas classes são tão triviais que esta pequena repetição não importa muito. Porém, quando você começar a escrever programas reais de alguma complexidade, suas classes irão, freqüentemente, conter numerosas variáveis e métodos; e você não quer ficar recodificando aquele mesmo velho código de novo, outra vez e novamente... Faz sentido, então, criar uma hierarquia de classes na qual uma classe que é um 'tipo especial' de alguma outra classe simplesmente 'herda' as características desta outra classe. No nosso simples jogo de aventura, por exemplo, um Treasure é um tipo especial de Thing então a classe Treasure deve herdar as características da classe Thing.

Hierarquias de Classes – Ascendentes e Descendentes: Neste livro, eu frequentemente falo sobre classes descendentes herdando características de suas classes ascendentes (ou superclasses, ou classes pai). Assim estes termos deliberadamente sugerem um tipo de relacionamento familiar entre as classes relacionadas. Em Ruby, cada classe possui somente um pai.
Uma classe pode, porém, descender de uma longa e distinta árvore familiar com muitas gerações de avós, bisavós e assim por diante…

O comportamento de Things em geral serão codificados na classe Thing. A classe Treasure irá automaticamente 'herdar' todas as características da classe Thing, assim nós não temos que codificar tudo novamente. Adiciona-se, então, algumas características, específicas de Treasures.
Como regra geral, quando criar uma hierarquia de classes, a classe com o comportamento mais generalizado é mais alta na hierarquia que as classes com comportamento mais específico. Assim a classe Thing apenas com um nome (name) e uma descrição (description), seria a superclasse de uma classe Treasure que tem um nome, uma descrição e, adicionalmente, um valor (método value); a classe Thing também poderá ser a superclasse de alguma outra classe especialista como uma Room (sala) que tem um nome, uma descrição e também saídas (exits) – e assim por diante...



Um Pai, muitos Filhos

Este gráfico mostra a classe Thing que tem um name e uma description (em um programa Ruby, estas podem ser variáveis internas como @name e @description mais alguns métodos para acessá-las). As classes Treasure e Room descendem da classe Thing assim elas automaticamente herdam um name e uma description. A classe Treasure adiciona um novo item: value – assim ela agora tem name, description e value; A classe Room adiciona exits – assim tem name, description e exits.

adventure1.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

# illustrates how to create descendent objects

class Thing
      def initialize( aName, aDescription )
        @name         = aName
        @description  = aDescription
      end

      def get_name
```

```
        return @name
    end

    def set_name( aName )
        @name = aName
    end

    def get_description
        return @description
    end

    def set_description( aDescription )
        @description = aDescription
    end
end


class Treasure < Thing      # Treasure descends from Thing
    def initialize( aName, aDescription, aValue )
        super( aName, aDescription )
        @value = aValue
    end

    def get_value
        return @value
    end

    def set_value( aValue )
        @value = aValue
    end
end


 # This is where our program starts...
  t1 = Treasure.new("Sword", "an Elvish weapon forged of
gold",800)
  t2 = Treasure.new("Dragon Horde", "a huge pile of jewels", 550)
  puts "This is treasure1: #{t1.inspect}"
  puts "This is treasure2: #{t2.inspect}"
  puts "t1 name=#{t1.get_name}, description=#{t1.get_description},
value=#{t1.get_value}"
  t1.set_value( 100 )
  t1.set_description(" (now somewhat tarnished)")
  puts "t1 (NOW) name=#{t1.get_name},
```

```
description=#{t1.get_description}, value=#{t1.get_value}"
```

Vamos ver como criar uma classe descendente em Ruby. Carregue o programa adventure1.rb. Este começa com uma definição da classe Thing que tem duas variáveis de instância, @name e @description. A estas variáveis são atribuidos valores no método initialize quando um novo objeto Thing é criado.
Variáveis de instância geralmente não podem (e não devem) ser acessadas diretamente pelo mundo exterior da classe devido ao princípio do encapsulamento.

> **"Encapsulamento"** é um termo que se refere à 'modularidade' de um objeto. De forma simples, significa que somente o próprio objeto pode mexer com o seu estado interno. O mundo externo não. O benefício disso é que o programador poderá mudar a implementação dos métodos sem ter que se preocupar com algum código externo que dependa de algum detalhe específico da implementação anterior.

Para obter o valor de cada variável em um objeto Thing nós precisamos de um método get como get_name; para passar um novo valor para uma variável nós precisamos de um método set como set_name:

```
def get_name
        return @name
end

def set_name( aName )
        @name = aName
end
```

Superclasses e Subclasses

Agora olhe a classe Treasure. Veja como é declarada:

```
class Treasure < Thing
```

O sinal de menor , < , indica que Treasure é uma 'subclasse', ou descendente, de Thing e por isso ela herda os dados (variáveis) e o comportamento (métodos) da classe Thing. Como os métodos

get_name, set_name, get_description e set_description já existem na classe ascendente (Thing) estes métodos não precisam ser recodificados na classe descendente (Treasure). A classe Treasure tem uma peça adicional de dado, seu valor (@value) e eu escrevi acessores get e set para ele. Quando um novo objeto Treasure é criado, seu método initialize é automaticamente chamado. Um objeto Treasure tem três variáveis para inicializar (@name, @description e @value), então seu método initialize recebe três argumentos:

```
def initialize( aName, aDescription, aValue )
```

Os primeiros dois argumentos são passados, usando a palavra-chave **super**, para o método initialize da superclasse (Thing) assim o método initialize da classe Thing pode lidar com elas:

```
super( aName, aDescription )
```

Quando usado dentro de um método, a palavra-chave **super** chama o método com o mesmo nome na classe ascendente ou 'superclasse'.
O método atual da classe Treasure é chamado initialize assim quando o código interno passa os dois argumentos (aName, aDescription) para **super** realmente está passando-as para o método initialize da sua superclasse, Thing.

Se a palavra-chave super for usada, sem qualquer argumento, todos os argumentos enviados ao método atual são passados para o método ascendente.

Cap. 4 - Acessores, Atributos, Variáveis de Classe

Acessores, Atributos e Variáveis de Classes…

Agora, voltando para o pequeno jogo de aventura que eu estava programando... Eu ainda não gosto do fato das classes estarem cheias de código repetido devido a todos os acessores get e set. Deixa ver o que eu posso fazer para remediar isso.

Métodos Acessores

Em vez de acessar o valor da variável de instância @description com dois métodos diferentes, get_description e set_description, como a seguir…

```
puts( t1.get_description )
t1.set_description( “Some description” )
```

… seria muito mais elegante recuperar e atribuir valores da mesma forma que você recupera e atribue valores para variáveis simples, como a seguir:

```
puts( t1.description )
t1.description = “Some description”
```

Para poder fazer isso, eu preciso modificar a definição da classe. Uma forma seria reescrever os métodos acessores para @description como segue:

```
def description
  return @description
end

def description=( aDescription )
  @description = aDescription
end
```

Eu adicionei acessores como os acima no programa accessors.rb (veja o código mais abaixo). Existem duas diferenças da versão anterior. Primeiro, ambos os acessores são chamados description em vez de get_description e set_description; segundo, o acessor set anexa um sinal de igual (=) ao nome do método. Agora é possível atribuir uma nova string para @description assim:

```
t.description = "a bit faded and worn around the edges"
```

E agora você pode recuperar o valor assim:

```
puts( t.description )
```

> Nota: Quando você escreve um acessor set desta forma, você deve anexar o caracter = diretamente no nome do método, não meramente colocá-lo em qualquer lugar entre o nome do método e seus argumentos. Logo, isto é correto:
> **`def name=( aName )`**
> Mas isto é errado:
> **`def name =( aName )`**

accessors.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

# illustrates how to read and write instance variables
# using accessor methods

class Thing

      def initialize( aName, aDescription )
        @name                = aName
        @description         = aDescription
      end

      # get accessor for @name
      def name
          return @name
      end

      # set accessor for @name
      def name=( aName )
          @name = aName
      end

      # get accessor for @description
```

```
        def description
            return @description
        end

        # set accessor for @description
        def description=( aDescription )
            @description = aDescription
        end


end

t = Thing.new("The Thing", "a lovely, glittery wotsit")
print( t.name )
print( " is " )
puts( t.description )
t.name = "A Refurbished Thing"
t.description = "a bit faded and worn around the edges"
print( "It has now changed its name to " )
puts( t.name )
print( "I would describe it as " )
puts( t.description )
```

Readers e Writers de Atributos

De fato, existe uma forma mais simples e curta de obter o mesmo resultado. Tudo que você tem que fazer é usar dois métodos especiais, attr_reader e attr_writer, seguidos de um símbolo como abaixo:

```
attr_reader :description
attr_writer :description
```

Você pode adicionar este código dentro da definição da classe mas fora de quaisquer métodos, assim:

```
class Thing
   attr_reader :description
  attr_writer :description

   # some methods here…
end
```

Símbolos: No Ruby, um símbolo é um nome precedido pelo sinal de dois pontos (:). Symbol é definida na biblioteca de classes do Ruby para representar nomes dentro do interpretador Ruby. Símbolos têm alguns usos especiais. Por exemplo, quando você passa um ou mais símbolos como argumentos para o método attr_reader (pode não ser óbvio, mas attr_reader é, de fato, um método da classe Module), o Ruby cria uma variável de instância e um método accessor get para retornar o valor da variável; ambos a variável de instância e o método acessor terão o mesmo nome que o símbolo especificado.

Chamando attr_reader com um símbolo têm o efeito de criar uma variável de instância com o mesmo nome do símbolo e um acessor get para aquela variável.
Chamando attr_writer, semelhantemente cria uma variável de instância com um acessor set.
Aqui, a variável seria chamada de @description. Variáveis de instância são consideradas como os 'atributos' de um objeto, o que explica porque os métodos attr_reader e attr_writer methods são assim chamados.

accessors2.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

# reading and writing attributes

    class Thing

      attr_reader :description
      attr_writer :description
      attr_writer :name

      def initialize( aName, aDescription )
        @name         = aName
        @description  = aDescription
      end

      # get accessor for @name
      def name
          return @name.capitalize
      end
```

```
# I don't need this name 'set' method since I have an attr_writer
for name
#       def name=( aName )
#                       @name = aName
#       end

     end


    class Treasure < Thing      # Treasure descends from Thing
      attr_accessor :value

      def initialize( aName, aDescription )
          super( aName, aDescription )
      end

    end


 # This is where our program starts...
  t1 = Treasure.new("sword", "an Elvish weapon forged of gold")
  t1.value = 800
  t2 = Treasure.new("dragon horde", "a huge pile of jewels")
  t2.value = 500
  puts "t1 name=#{t1.name}, description=#{t1.description},
value=#{t1.value}"
  t1.value= 100
  t1.description << " (now somewhat tarnished)" # note << appends
specified string to existing string
  puts "t1 name=#{t1.name}, description=#{t1.description},
value=#{t1.value}"
  puts "This is treasure1: #{t1.inspect}"
  puts "This is treasure2: #{t2.inspect}"
```

O programa accessors2.rb contém alguns exemplos de readers e writers de atributos em ação. Perceba que a classe Thing define a forma simplificada do acessor set (usando attr_writer mais um símbolo) para a variável @name:

```
attr_writer :name
```

Mas tem o acessor get na forma longa – um método codificado inteiramente – para a mesma variável:

```
def name
       return @name.capitalize
end
```

A vantagem de escrever um método completo como esse é que dá a você a oportunidade de fazer algum processamento extra em vez de apenas ler e escrever o valor de um atributo. Aqui o acessor get usa o método capitalize da classe String para retornar a string @name com suas letras iniciais em maiúsculas.
O atributo @description não precisa de processamento especial assim sendo eu usei ambos attr_reader e attr_writer para pegar e atribuir o valor da variável @description.

> **Atributos ou Propriedades?** Não seja confundido pela terminologia. No Ruby, um 'atributo' é equivalente ao que muitas outras linguagens de programação chamam de 'propriedade'.

Quando você quer ler e escrever uma variável, o método **attr_accessor** method fornece uma alternativa mais curta que usar ambos attr_reader e attr_writer. Eu usei isso para acessar o valor do atributo value na classe Treasure:

```
attr_accessor :value

This is equivalent to:

attr_reader :value
attr_writer :value
```

Atributos Criam Variáveis

Recentemente eu disse que chamar attr_reader com um símbolo cria uma variável com o mesmo nome do símbolo. O método attr_accessor também faz isso. No código da classe Thing, este comportamento não é óbvio já que a classe tem um método initialize que explicitamente cria as variáveis. A classe Treasure, porém, não faz referência à variável @value no seu método initialize:

```
class Treasure < Thing

    attr_accessor :value
```

```
    def initialize( aName, aDescription )
        super( aName, aDescription )
    end

end
```

A única indicação que uma variável @value existe é a definição do acessor que declara um atributo value:

```
attr_accessor :value
```

Meu código no final do arquivo fonte atribue o valor de cada objeto Treasure:

```
t1.value = 800
```

O atributo value não foi formalmente declarado, a variável @value realmente não existe e nós podemos recuperar o seu valor numérico usando o acessor get:

```
t1.value
```

Para ter certeza que o acessor do atributo realmente criou @value, você pode inspecionar o objeto usando o método inspect. Eu fiz isso no final do código fonte com as duas linhas finais do programa:

```
puts "This is treasure1: #{t1.inspect}"
puts "This is treasure2: #{t2.inspect}"
```

accessors3.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

# more on reading and writing attributes

class Thing
      attr_reader :name, :description
      attr_writer(:name, :description)
      attr_accessor(:value, :id, :owner)
end

t = Thing.new
```

```
t.name = "A Thing"
t.description = "A soft, furry wotsit"
t.value = 100
t.id = "TH100SFW"
t.owner = "Me"
puts("#{t.name} is #{t.description}, it is worth $#(t.value)")
puts("it's id is #{t.id}. It is owned by #{t.owner}.")
```

Acessores de Atributo podem inicializar mais que um atributo por vez se você enviar a eles uma lista de símbolos na forma de argumentos separados por vírgulas, como este exemplo:

```
attr_reader :name, :description
attr_writer(:name, :description)
attr_accessor(:value, :id, :owner)
```

Como sempre, no Ruby, os parênteses em volta dos argumentos são opcionais.

adventure2.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

# illustrates how to creating descendent objects
# reading and writing attributes
# object (instance) variables
# class variables

     # Thing
    class Thing
      @@num_things = 0 # class variable

      attr_reader( :name, :description )
      attr_writer( :description )

      def initialize( aName, aDescription )
        @name         = aName
        @description  = aDescription
        @@num_things +=1  # increment @@num_things
      end

      def to_s # override default to_s method
        return "(Thing.to_s):: The #{@name} Thing is
#{@description}"
```

```
      end

      def show_classvars
        return "There are #{@@num_things} Thing objects in this
game"
      end

    end

     # Room
    class Room < Thing
      # TODO: Add Room-specific behaviour
    end

     # Treasure
    class Treasure < Thing
      attr_reader :value
      attr_writer :value

      def initialize( aName, aDescription, aValue )
          super( aName, aDescription )
          @value = aValue
      end
    end

    # Map
    class Map
       # @rooms will be an array - an ordered list
       # of Room objects
      def initialize( someRooms )
        @rooms = someRooms
      end

      # The to_s method iterates over all the Room objects in
@rooms
      # and prints information on each. We'll come back to look at
the
      # implementation of this method later on
      def to_s
        @rooms.each {
          |a_room|
          puts(a_room)
        }
      end
```

```
    end

  # First create a few objects
  #   i) Treasures
  t1 = Treasure.new("Sword", "an Elvish weapon forged of
gold",800)
  t2 = Treasure.new("Dragon Horde", "a huge pile of jewels", 550)
  #   ii) Rooms
  room1 = Room.new("Crystal Grotto", "A glittery cavern")
  room2 = Room.new("Dark Cave", "A gloomy hole in the rocks")
  room3 = Room.new("Forest Glade", "A verdant clearing filled with
shimmering light")
  #   iii) a Map - which is an array containing the Rooms just
created
  mymap = Map.new([room1,room2,room3])
  # Now let's take a look at what we've got...
  puts "\nLet's inspect the treasures..."
  puts "This is the treasure1: #{t1.inspect}"
  puts "This is the treasure2: #{t2.inspect}"
  puts "\nLet's try out the Thing.to_s method..."
  puts "Yup, treasure 2 is #{t2.to_s}"
  puts "\nNow let's see how our attribute accessors work"
  puts "We'll evaluate this:"
  puts 't1 name=#{t1.name}, description=#{t1.description},
value=#{t1.value}'
  puts "t1 name=#{t1.name}, description=#{t1.description},
value=#{t1.value}"
  puts "\nNow we'll assign 100 to t1.value and alter
t1.description..."
  t1.value = 100
  t1.description << " (now somewhat tarnished)" # note << appends
specified string to existing string
  puts "t1 (NOW) name=#{t1.name}, description=#{t1.description},
value=#{t1.value}"
  puts "\nLet's take a look at room1..."
  puts "room1 name=#{room1.name},
description=#{room1.description}"
  puts "\nAnd the map..."
  puts "mymap = #{mymap.to_s}"
  puts "\nFinally, let's check how many Thing objects we've
created..."
  puts( t1.show_classvars )

 # As an exercise, try adding a class variable to the Map class to
```

```
maintain
 # a count of the total number of rooms that have been created
```

Agora vamos ver como colocar readers e writers de atributos para usar no meu jogo de aventura. Carregue o programa adventure2.rb (listagem acima). Você verá que eu criei dois atributos na classe Thing: name e description. Eu também fiz o atributo description poder ser atribuído pelo programador (**attr_writer)**; porém, eu não planejo mudar os nomes dos objetos criados a partir da classe Thing, por isso o atributo name tem somente o acessor **attr_reader.**

```
attr_reader( :name, :description )
attr_writer( :description )
```

Eu criei um método chamado to_s que retorna a string descrevendo o objeto Treasure. Lembre que toda classe Ruby têm um método to_s padrão. O método to_s na classe Thing sobrescreve (substitui) o padrão. Você pode sobrescrever métodos existentes quando você implementa um novo comportamento apropriado para o tipo específico da classe.

Chamando métodos de uma superclasse

Eu decidi que meu jogo terá duas classes descendendo de Thing. A classe Treasure adiciona um atribute value que pode ser lido e escrito. Note que o método initialize chama sua superclasse para inicializar os atributos name e description antes de inicializar a nova variável @value:

```
super( aName, aDescription )
@value = aValue
```

Aqui, se eu omitisse a chamada para a superclasse, os atributos name e description nunca seriam inicializados. Isto é porque Treasure.initialize sobrescreve Thing.initialize; logo quando um objeto Treasure é criado, o código em Thing.initialize nunca será automaticamente executado.

> Em alguns livros de Ruby, um sinal # pode ser mostrado entre o nome da classe e o nome do método, assim: Treasure#initialize.
> Isto é apenas uma convenção de documentação (a qual eu prefiro ignorar) e não é uma sintaxe real Ruby. Eu acho é somente o caso de "You say tomayto and I say tomahto"; você diz Treasure#initialize e eu digo Treasure.initialize". Não vamos brigar por isso ...!

De outro lado, a classe Room, que também descende de Thing, não tem um método initialize; logo quando um novo objeto Room é criado o Ruby volta-se para para a hierarquia de classes ascendentes para procurar um initalize. O primeiro método initialize que ele encontra está na classe Thing; assim os atributos name e description do objeto Room são inicializados lá.

Variáveis de Classe

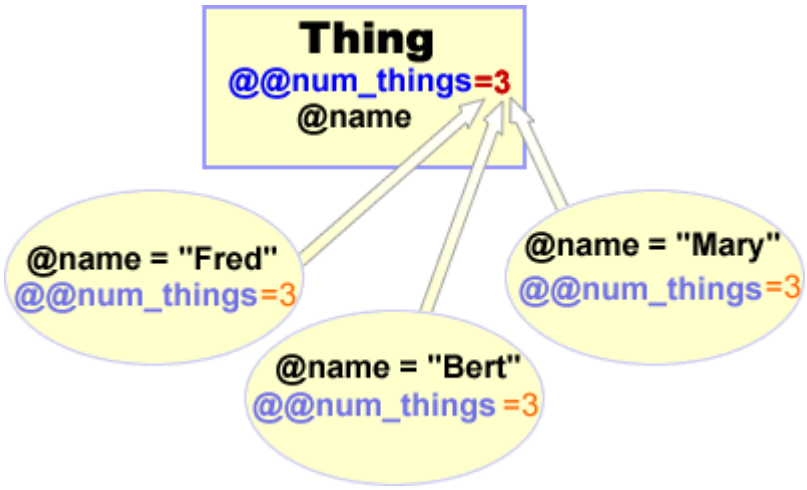
Existem algumas outras coisas interessantes neste programa. No topo da classe Thing você verá isso:

```
@@num_things = 0
```

Os dois caracteres @ no início do nome da variável, @@num_things, definem que essa é uma 'variável de classe'. As variáveis que nós usamos dentro das classes até agora eram 'variáveis de instância', precedidas por um único @, como @name.
Uma vez que um novo objeto (ou 'instância') de uma classe atribue seus próprios valores para suas próprias variáveis de instância, todos os objetos derivados de uma classe específica compartilham as mesmas variáveis da classe. Eu atribui 0 para a variável @@num_things para assegurar que ela tenha um valor significativo para o exterior. Aqui, a variável de classe @@num_things é usada para manter o valor corrente do número de objetos Thing no jogo. Isto é feito pelo incremento da variável de classe (usamos += para adicionar 1 ao número de objetos) no método initialize cada vez que um novo objeto é criado:

```
@@num_things +=1
```

Se você olhar na parte de baixo do meu código, você verá que eu criei uma classe Map que contém uma matriz de salas (objetos Room). Inclui uma versão do método to_s que imprime informação sobre cada sala na matriz. Não se preocupe com a implementação da classe Map; nós veremos arrays e seus métodos brevemente.



O diagrama acima mostra a classe Thing (o retângulo) que contém uma variável de classe, @@num_things e uma variável de instância, @name, As três ovais representam 'objetos Thing' – isto é, 'instâncias' da classe Thing. Quando um destes objetos atribue um valor para sua variável de instância, @name, este valor afeta somente a variável @name no próprio objeto – então aqui, cada objeto têm um valor diferente para @name. Mas quando um objeto atribue um valor para a variável de classe, @@num_things, aquele valor 'vive dentro' da classe Thing e é compartilhado por todas as instâncias desta classe. Aqui @@num_things é igual a 3 e que é igual para todos os objetos Thing.

Ache o código no final do arquivo e rode o programa para ver como nós criamos e inicializamos todos os objetos e usamos a variável de classe, @@num_things, para manter a contagem de todos os objetos Thing que foram criados.

Cap. 5 - Arrays

Matrizes (Arrays)...

Até agora, nós geralmente temos usados objetos um de cada vez. Neste capítulo nós veremos como criar uma lista de objetos. Vamos iniciar vendo o tipo de lista mais comum – uma matriz (array).

O que é um Array?
Um array é uma coleção sequencial de itens na qual cada item pode ser indexado.
No Ruby, (diferente de muitas outras linguagens) um único Array pode conter itens de tipos de dados misturados como strings, inteiros e números de ponto flutuante ou mesmo chamadas de métodos que retornem algum valor:

```
a1 = [1,'two', 3.0, array_length( a0 ) ]
```

O primeiro item de um array têm o índice 0, o que significa que o item final têm um índice igual ao total de itens do array menos 1. Dado o array, a1, mostrado acima, esta é a forma de obter os valores do primeiro e último itens:

```
a1[0]        # returns 1st item (at index 0)
a1[3]        # returns 4th item (at index 3)
```

array0.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

def array_length( anArray )
     return anArray.length
end

a0 = [1,2,3,4,5]
a1 = [1,'two', 3.0, array_length( a0 ) ]

p( a0 )
p( a1 )
puts( "Item index #0 of a1 is #{a1[0]}, item #3 is #{a1[3]}" )
```

Usando arrays

Nós já usamos arrays algumas vezes – por exemplo, no programa adventure2.rb no capítulo 4 nós usamos um array para armazenar um mapa de salas (objetos Room):

```
mymap = Map.new([room1,room2,room3])
```

Criando Arrays

Em comum com muitas outras linguagens de programação, o Ruby usa os sinais de colchetes [] para delimitar um array. Você pode facilmente criar um array, preencha-o com alguns valores separados por vírgula e atribua-os a uma variável:

```
arr = ['one','two','three','four']
```

array1.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

arr = ['a', 'b', 'c']

puts(arr[0])  # shows a
puts(arr[1])  # shows b
puts(arr[2])  # shows c

puts(arr[3]) # nil
```

Como a maioria das outras coisas no Ruby, arrays são objetos. Eles são definidos, como você deve ter adivinhado, por uma classe Array, assim como as strings, os arrays são indexados a partir de 0. Você pode referenciar um item em uma matriz colocando o seu índice entre colchetes []. Se o índice for inválido, nil é retornado:

```
arr = ['a', 'b', 'c']

puts(arr[0]) # shows ‘a’
puts(arr[1]) # shows ‘b’
```

```
puts(arr[2]) # shows 'c'

puts(arr[3]) # nil
```

array2.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

def hello
     return "hello world"
end

x = [1+2, hello,  `dir`]      # array containing an expression, a
methodcall and a string
puts(x.inspect)          # Note: if your are not using Windows, you
may need to
                    # change `dir` to a command that is understood
by your
                    # operating system

y = %w( this is an array of strings )
puts(y.inspect)

a = Array.new
puts("Array.new : " << a.inspect)

a = Array.new(2)
puts("Array.new(2) : " << a.inspect)

a = Array.new(2,"hello world")
puts(a.inspect)

a = Array.new(2)
a[0]= Array.new(2,'hello')
a[1]= Array.new(2,'world')
puts(a.inspect)

a = [     [1,2,3,4],
          [5,6,7,8],
          [9,10,11,12],
          [13,14,15,16] ]
puts(a.inspect)

a = Array.new([1,2,3])
```

```
puts(a.inspect)

# Note: in the above example, if you pass an array to new()
without
# putting it in sound brackets, you must leave a space between
# 'new' and the opening square bracket.
# This works:
#         a = Array.new [1,2,3]
# This doesn't!
#         a = Array.new[1,2,3]
```

É permitido misturar tipos de dados em um array e até mesmo incluir expressões que produzam algum valor. Vamos assumir que você já criou este método:

```
def hello
  return "hello world"
end
```

Você pode agora declarar este array:

```
x = [1+2, hello, `dir`]
```

Aqui, o primeiro elemento é um inteiro, 3 (1+2), e o segundo é uma string“hello world” (retornada pelo método hello criado acima). Se você executar isto no Windows, o terceiro elemento será uma string contendo a listagem do diretório. Isto deve-se ao fato que `dir` é uma string cotada por crase (`) que é executada pelo sistema operacional. O slot final do array é, por essa razão, preenchido com o valor retornado pelo comando dir que monta a string com nomes de arquivos. Se você está usando um sistema operacional diferente você deve substituir o comando dir pelo comando apropriado do seu sistema operacional (por exemplo no Linux o comando é o `ls`).

Criando um Array de Nomes de Arquivos: Algumas classes Ruby possuem métodos que retornam arrays. Por exemplo, a classe Dir, que é usada para executar operações em diretórios de disco, têm o método entries. Passe um nome de diretório para o método e ele retorna uma lista de arquivos em um array:

```
Dir.entries( 'C:\\' )      # returns an array of files in C:\
```

dir_array.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com
```

```
p( Dir.entries( 'C:\\' ) )
```

Se você quer criar um array de strings mas não quer chatear-se digitando todas as aspas e vírgulas, um atalho é colocar texto não cotado separado por espaços entre parênteses precedido por %w como este exemplo:

```
y = %w( this is an array of strings )
```

Você pode também criar arrays com o método usual de construção de objetos, new. Opcionalmente, você pode passar um inteiro para criar um array vazio com um tamanho específico (no qual cada elemento será igual a nil), ou você pode passar dois argumentos – o primeiro para definir o tamanho do array e o segundo para especificar o elemento a colocar em cada índice, assim:

```
a = Array.new                          # an empty array
a = Array.new(2)                       # [nil,nil]
a = Array.new(2,"hello world") # ["hello world","hello world"]
```

Arrays Multi-Dimensionais

Para criar arrays multi-dimensionais, você pode criar um array e então adicionar outros arrays para os 'slots' deste array. Por exemplo, isto cria um array contendo dois elementos, cada qual é também um array de dois elementos:

```
a = Array.new(2)
a[0]= Array.new(2,'hello')
a[1]= Array.new(2,'world')
```

Ou você poderia aninhar arrays dentro de outro usando os colchetes. Isto cria um array de quatro arrays, cada qual contém quatro inteiros:

```
a = [ [1,2,3,4],
        [5,6,7,8],
        [9,10,11,12],
        [13,14,15,16] ]
```

No código acima, eu coloquei quatro ‘sub-arrays’ em linhas separadas. Isto não é obrigatório mas ajuda a tornar clara a estrutura do array multi-dimensional mostrando cada sub-array como se fosse uma linha, semelhante a linhas em uma planilha de cálculo. Quando tratar de arrays dentro de

arrays, é conveniente tratar cada array aninhado como uma linha de um array externo.

array_new.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

# all these are ok
arrayob = Array.new([1,2,3])
arrayob = Array.new( [1,2,3] )
arrayob = Array.new [1,2,3]

# but this one isn't
arrayob = Array.new[1,2,3]

p( arrayob )
```

Você também pode criar um objeto Array passando um array com um argumento para o método new. Seja cuidadoso, contudo. É uma artimanha do Ruby que, mesmo sendo legítimo passar um argumento array com ou sem os parênteses delimitadores, o Ruby considera um erro de sintaxe se você esquecer de deixar um espaço entre o new e o primeiro colchete – outra boa razão para criar o firme hábito de usar parênteses quando passar argumentos!

Para alguns exemplos do uso de arrays multi-dimensionais, carregue o programa multi_array.rb. Ele começa criando um array, multiarr, contendo dois outros arrays. O primeiro destes arrays está no índice 0 de multiarr e o segundo está no índice 1:

```
multiarr = [['one','two','three','four'],[1,2,3,4]]
```

multi_array.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

# create an array with two sub-arrays - i.e. 2 'rows' each with 4
elements
multiarr = [['one','two','three','four'],   # multiarr[0]
 [1,2,3,4]]                  # multiarr[1]
```

```
# iterate over the elements ('rows') of multiarr
puts( "for i in.. (multi-dimensional array)" )
for i in multiarr
     puts(i.inspect)
end

puts( "\nfor a, b, c, d in.. (multi-dimensional array)" )
for (a,b,c,d) in multiarr
     print("a=#{a}, b=#{b}, c=#{c}, d=#{d}\n" )
end
```

Iterando sobre Arrays

Você pode acessar os elementos de uma matriz, iterando sobre eles usando um laço for. O laço irá iterar sobre dois elementos neste exemplo: nomeadamente, os dois sub-arrays nos índices 0 e 1:

```
for i in multiarr
        puts(i.inspect)
end
```

Isto mostra:

```
["one", "two", "three", "four"]
[1, 2, 3, 4]
```

array_index.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

arr = ['h','e','l','l','o',' ','w','o','r','l','d']

# Here we print each char in sequence
print( arr[0,5] )
puts
print( arr[-5,5 ] )
puts
print( arr[0..4] )
puts
print( arr[-5..-1] )
```

```
puts

# Here we inspect the chars. Notice that we can
# index into an array like this...
#    arr[<start index>, <number of items>]
# or specify a range like this...
#    arr[<start index> .. <end index> ]
#
# So these are equivalent:

p( arr[0,5] )
p( arr[0..4] )
```

Iteradores e laços for: O código dentro de um laço for é executado para cada elemento em uma expressão. A sintaxe é resumida assim:

```
for <uma ou mais variáveis> in <expressão> do
      <código para executar>
end
```

Quando mais de uma variável é fornecida, elas são passadas para o código interno do bloco for..end da mesma forma que você passa argumentos para um método. Aqui, por exemplo, você pode ver (a,b,c,d) como quatro argumentos que são inicializados, a cada volta do laço for, pelos quatro valores de uma linha de multiarr:

```
for (a,b,c,d) in multiarr
  print("a=#{a}, b=#{b}, c=#{c}, d=#{d}\n" )
end
```

Indexando Arrays

Você pode indexar a partir do fim de um array usando o sinal de menos (-), onde -1 é o índice do último elemento; e você também pode usar faixas (valores entre um índice inicial e final separados por dois sinais de ponto ..):

```
arr = ['h','e','l','l','o',' ','w','o','r','l','d']
print( arr[0,5] )              #=> 'hello'
print( arr[-5,5 ] )            #=> 'world'
```

```
print( arr[0..4] )              #=> ‘hello’
print( arr[-5..-1] )            #=> ‘world’
```

Note que, como as strings, quando fornecido dois inteiros para retornar um número contíguo de itens de um array, o primeiro inteiro é o índice inicial e o segundo é número de itens (não um índice):

```
arr[0,5]                        # returns 5 chars - ["h", "e", "l",
"l", "o"]
```

array_index.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

arr = []

arr[0] = [0]
arr[1] = ["one"]
arr[3] = ["a", "b", "c"]

p( arr ) # inspect and print arr

arr2 = ['h','e','l','l','o',' ','w','o','r','l','d']

arr2[0] = 'H'
arr2[2,3] = 'L', 'L'
arr2[4..5] = 'O','-','W'
arr2[-4,4] = 'a','l','d','o'

p( arr2 ) # inspect and print arr2
```

Você também pode designar índices dentro de um array. Aqui, por exemplo, Eu primeiro criei um array vazio então coloquei itens nos índices 0, 1 e 3. O slot vazio no índice número 2 será preenchido com um valor nil:

```
arr = []
arr[0] = [0]
arr[1] = ["one"]
arr[3] = ["a", "b", "c"]

# arr now contains:
```

```
# [[0], ["one"], nil, ["a", "b", "c"]]
```

Uma vez mais, você pode usar índices iniciais e finais, faixas e índices negativos:

```
arr2 = ['h','e','l','l','o',' ','w','o','r','l','d']
arr2[0] = 'H'
arr2[2,3] = 'L', 'L'
arr2[4..5] = 'O','-','W'
arr2[-4,4] = 'a','l','d','o'
# arr2 now contains:
# ["H", "e", "L", "L", "O", "-", "W", "a", "l", "d", "o"]
```

Cap. 6 - Hashes

Hashes...

Mesmo que os arrays forneçam uma boa forma de indexar uma coleção de itens por número, há situações que seria mais conveniente indexar de alguma outra forma. Se, por exemplo, você for criar uma coleção de receitas, seria mais significativo ter cada receita indexada pelo nome tal como “Bolo de Chocolate” e “Salada Mista” em vez de números: 23, 87 e assim por diante.
O Ruby têm uma classe que permite você fazer exatamente isso: Ela é chamada de Hash. Isto é o equivalente ao que outras linguagens chamam de “Dicionário (Dictionary)”. Como um dicionário real, as entradas são indexadas por alguma chave única (num dicionário, seria uma palavra) e um valor (num dicionário, seria a definição da palavra).

hash1.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

h1 = {    'room1'=>'The Treasure Room',
     'room2'=>'The Throne Room',
     'loc1'=>'A Forest Glade',
     'loc2'=>'A Mountain Stream' }

class X
     def initialize( aName )
          @name = aName
     end
end

x1 = X.new('my Xobject')

h2 = Hash.new("Some kind of ring")
h2['treasure1'] = 'Silver ring'
h2['treasure2'] = 'Gold ring'
h2['treasure3'] = 'Ruby ring'
h2['treasure4'] = 'Sapphire ring'
h2[x1] = 'Diamond ring'

h3 = {
'treasure3'=>'Ruby ring',
```

```
'treasure1'=>'Silver ring',
'treasure4'=>'Sapphire ring',
'treasure2'=>'Gold ring'
}

p(h2)                                # inspect Hash
puts(h1['room2'])                 # get value using a key ('The Throne
Room')
p(h2['treasure1'])                # get value using a key ('Silver
ring')
p(h1['unknown_room'])             # returns default value (nil)
p(h2['unknown_treasure'])         # returns default value ('Some kind
of ring')
p(h1.default)                     #=> nil
p(h2.default)                     #=> 'Some kind of ring'
h1.default = 'A mysterious place'
puts(h1.default)                  #=> 'A mysterious place'
p(h2[x1])                         # here key is object, x1; value is
'Diamond ring'
```

Criando Hashes

Você pode criar um hash criando uma nova instância da classe Hash:

```
h1 = Hash.new
h2 = Hash.new("Some kind of ring")
```

Ambos os exemplos acima criam um Hash vazio. Um objeto Hash sempre têm um valor padrão – isto é, um valor que é retornado quando nenhum valor específico é encontrado em um dado índice. Nestes dois exemplos, h2 é inicializado com o valor padrão, “Some kind of ring”; h1 não é inicializado com um valor então seu valor padrão será nil.

Tendo criado um objeto Hash, você pode adicionar itens a ele usando uma sintaxe semelhante à dos arrays – isto é, colocando um índice nos colchetes e usando o sinal de igual = para atribuir um valor.
A diferença óbvia aqui é que, com um array, o índice (a chave) deve ser um número inteiro; com um Hash, ele pode ser qualquer item de dado único:

```
h2['treasure1'] = 'Silver ring'
h2['treasure2'] = 'Gold ring'
```

```
h2['treasure3'] = 'Ruby ring'
h2['treasure4'] = 'Sapphire ring'
```

Muitas vezes, a chave pode ser um número ou, como no código acima, uma string. Por princípio, uma chave pode ser qualquer tipo de objeto. Dada alguma classe, X, a seguinte atribuição é perfeitamente legal:

```
x1 = X.new('my Xobject')
h2[x1] = 'Diamond ring'
```

Existe uma forma abreviada de criar Hashes e inicializá-los com pares chave-valor. Adicione a chave seguida por => e o valor associado; cada par chave-valor deve ser separado por uma vírgula e o lote todo colocado entre os sinais de chaves {}:

```
h1 = { 'room1'=>'The Treasure Room',
       'room2'=>'The Throne Room',
       'loc1'=>'A Forest Glade',
       'loc2'=>'A Mountain Stream' }
```

Chaves Únicas? Tome cuidado quando atribuir chaves para Hashes. Se você usar a mesma chave duas vezes em um Hash, você acabará sobrescrevendo o valor original. É a mesma coisa que atribuir duas vezes um valor para o mesmo índice de um array. Considere este exemplo:

```
h2['treasure1'] = 'Silver ring'
h2['treasure2'] = 'Gold ring'
h2['treasure3'] = 'Ruby ring'
h2['treasure1'] = 'Sapphire ring'
```

Aqui a chave 'treasure1' foi usada duas vezes. Como conseqüência, o valor original, 'Silver ring' foi substituído por 'Sapphire ring', resultando neste Hash:

```
{"treasure1"=>"Sapphire ring",
"treasure2"=>"Gold ring",
"treasure3"=>"Ruby ring"}
```

Indexando em um Hash

Para acessar um valor, coloque sua chave entre colchetes:

```
puts(h1['room2'])                    #=> 'The Throne Room'
```

Se você especificar uma chave que não existe, o valor padrão é retornado. Lembre que nós não especificamos um valor padrão para h1 mas o fizemos para h2:

```
p(h1['unknown_room'])                #=> nil
p(h2['unknown_treasure'])            #=> 'Some kind of ring'
```

Use o método ***default*** para pegar o valor padrão e o método ***default=*** para atribui-lo (veja o Capítulo 4 para mais informações sobre métodos get e set):

```
p(h1.default)
h1.default = 'A mysterious place'
```

hash2.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

h1 = {'key1'=>'val1', 'key2'=>'val2', 'key3'=>'val3',
'key4'=>'val4'}
h2 = {'key1'=>'val1', 'KEY_TWO'=>'val2', 'key3'=>'VALUE_3',
'key4'=>'val4'}

p( h1.keys & h2.keys )
p( h1.values & h2.values )
p( h1.keys+h2.keys )
p( h1.values-h2.values )
p( (h1.keys << h2.keys)  )
p( (h1.keys << h2.keys).flatten.reverse  )
```

Operações Hash

Os métodos **keys** e **values** do Hash retornam um array logo você pode usar vários métodos Array methods para manipulá-los. Aqui estão uns poucos exemplos (note, os dados mostrados nos comentários começando por #=> mostram os valores retornados quando cada peça de código é executada):

```
h1 = {'key1'=>'val1', 'key2'=>'val2', 'key3'=>'val3',
'key4'=>'val4'}
h2 = {'key1'=>'val1', 'KEY_TWO'=>'val2', 'key3'=>'VALUE_3',
'key4'=>'val4'}
p( h1.keys & h2.keys )  # set intersection (keys) #=> ["key1",
"key3", "key4"]
p( h1.values & h2.values )                                  # set
intersection (values)
#=> ["val1", "val2", "val4"]
p( h1.keys+h2.keys )                                        #
concatenation
#=> [ "key1", "key2", "key3", "key4", "key1", "key3", "key4",
"KEY_TWO"]
p( h1.values-h2.values )                                    #
difference
#=> ["val3"]
p( (h1.keys << h2.keys) )                                   # append
#=> ["key1", "key2", "key3", "key4", ["key1", "key3", "key4",
"KEY_TWO"] ]
p( (h1.keys << h2.keys).flatten.reverse ) # 'un-nest' arrays and
reverse
#=> ["KEY_TWO", "key4", "key3", "key1", "key4", "key3", "key2",
"key1"]
```

Atenção ao notar a diferença entre concatenar usando **+** para adicionar o valor de um segundo array ao primeiro array e anexar usando **<<** para adicionar o segundo array como o último elemento do primeiro array:

```
a =[1,2,3]
b =[4,5,6]
c=a+b                                  #=> c=[1, 2, 3, 4, 5, 6] a=[1,
2, 3]
a << b                                 #=> a=[1, 2, 3, [4, 5, 6]]
```

Adicionalmente, **<<** modifica o primeiro (o 'recebedor') array mas o **+** retorna um novo array deixando o array recebedor inalterado. Se, após anexar um array com **<<** você decide que gostaria de adicionar os elementos do array anexado ao recebedor em vez de anexar o array propriamente 'aninhado' dentro do recebedor, você pode fazer isto usando o método ***flatten***:

```
a=[1, 2, 3, [4, 5, 6]]
a.flatten                          #=> [1, 2, 3, 4, 5, 6]
```

Cap. 7 - Laços e Iteradores

Laços e iteradores...

A maior parte da atividade de programação está relacionada com repetição. Você pode querer que um programa bipe 10 vezes, leia linhas de um arquivo tantas quantas linhas existirem para serem lidas ou mostrar uma mensagem até que o usuário pressione uma tecla. O Ruby fornece várias formas de executar estes tipos de repetições.

Laços FOR

Em muitas linguagens de programação, quando você quer executar um pedaço de código um certo número de vezes você pode fazê-lo colocando o pedaço de código dentro de um laço ***for.*** Na maioria das linguagens, você tem que dar ao laço ***for*** uma variável com um valor inicial o qual é incrementado de 1 a cada volta do laço até que se atinja um valor final específico. Quando o valor final é atingido, o laço ***for*** termina a execução. Aqui está a versão deste tipo tradicional de laço ***for*** escrito em Pascal:

```
# Este é um código Pascal, não é Ruby!
For i := 1 to 3 do
  writeln( i );
```

for_loop.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

puts( '--- loop #1 ---' )
for i in [1,2,3]  do
     puts( i )
end

puts( '--- loop #3 ---' )
# 'do' is optional when for loop is 'multi-line'
for s in ['one','two','three'] #do
     puts( s )
end
```

```
puts( '--- loop #3 ---' )
# 'do' is obligatory when for loop is on a single line
for x in [1, "two", [3,4,5] ] do puts( x ) end
```

Você pode rever no Capítulo 5 (arrays) que o laço ***for*** do Ruby não trabalha exatamente desta forma! Em vez de fornecer um valor inicial e final, nós fornecemos ao laço ***for*** uma lista de itens e ele itera sobre eles, um a um, atribuindo o valor para a variável do laço até que se chegue ao final da lista.
Por exemplo, aqui está uma laço for que itera sobre seus itens em uma matriz, mostrando o valor a cada volta:

```
# Este é um código Ruby...
for i in [1, 2, 3] do
  puts( i )
end
```

O laço for é mais parecido com o iterador 'for each' existente em algumas outras linguagens. O autor do Ruby descreve o for como “açúcar sintático” para cada método ***each*** o qual é implementado por “tipos coleção” do Ruby como Arrays, Sets, Hashes e Strings (uma String sendo, com efeito, uma coleção de caracteres).

Para efeito de comparação, este é o laço ***for*** mostrado acima reescrito usando o método ***each:***

```
[1, 2, 3].each do | i |
  puts( i )
end
```

Como você pode ver, não existe muita diferença.

each_loop.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

puts( '--- loop #1 ---' )
[1,2,3].each  do |i|
     puts( i )
end

puts( '--- loop #3 ---' )
```

```
['one','two','three'].each do |s|
     puts( s )
end


puts( '--- loop #3 ---' )
[1, "two", [3,4,5] ].each do |x| puts( x ) end
```

Para converter o laço ***for*** em um iterador ***each***, tudo o que fiz foi excluir o ***for*** e o ***in*** e anexar ***.each*** ao array. Então eu coloquei a variável do iterador, ***i***, entre pipes (||).

```
# ii) each
['one', 'two', 'three'].each do |s|
  puts( s )
end

# --- Example 2 ---

# i) for
for x in [1, “two”, [3,4,5]] do puts( x ) end
```

Note que a palavra-chave ***do*** é opcional em um laço ***for*** que se extende por múltiplas linhas mas é obrigatório quando ele é escrito em uma única linha:

```
# Aqui a palavra-chave 'do' pode ser omitida
for s in ['one', 'two', 'three']
  puts( s )
end

# Mas aqui é obrigatório
for s in ['one', 'two', 'three'] do puts( s ) end
```

Como escrever um laço *for* 'normal'...

Se você sentir saudades do tipo tradicional do laço ***for***, você sempre pode 'imitá-lo' em Ruby usando um laço ***for*** para iterar sobre os valores de uma faixa. Por exemplo, veja como usar uma variável para o laço ***for*** para contar de 1 até 10, mostrando o valor a cada volta do laço:

```
for i in (1..10) do
  puts( i )
end
```

O qual pode ser reescrito usando each:

```
(1..10).each do | i |
  puts( i )
end
```

Note que a expressão da faixa, por exemplo 1..3, deve ser escrita entre parênteses, "()", quando é usada com o método ***each***, senão o Ruby assume que você está tentando usar o ***each*** como um método do número final da faixa (um FixNum) em vez da expressão inteira (uma faixa). Os parênteses são opcionais quando uma faixa é usada no laço ***for***.

Quando iteramos sobre os itens usando ***each***, o bloco de código entre o ***do*** e o ***end*** é chamado (obviamente) de 'bloco iterador'.

Parâmetros de Bloco: No Ruby quaisquer variáveis declaradas no topo do bloco são chamadas de "parâmetros do bloco". Desta forma, um bloco funciona como uma função e os parâmetros do bloco funcionam como a lista de argumentos da função. O método each executa o código interno do bloco e passa para ele os argumentos fornecidos por uma coleção (por exemplo, uma matriz).

Blocos

O Ruby tem uma sintaxe alternativa para delimitar blocos. Em vez de usar **do..end**, como abaixo..

```
# do..end
[[1,2,3],[3,4,5],[6,7,8]].each do
  | a, b, c |
    puts( "#{a}, #{b}, #{c}" )
end
```

... você pode usar chaves {..} assim:

```
# chaves {..}
[[1,2,3],[3,4,5],[6,7,8]].each {
  | a, b, c |
    puts( "#{a}, #{b}, #{c}" )
}
```

Não importa qual delimitador você use, você deve assegurar-se que o delimitador de abertura do bloco, **'{' ou 'do'**, seja colocado na mesma linha do método *each*. Colocar uma quebra de linha entre o método ***each*** e o delimitador de abertura do bloco é um erro de sintaxe.

Laços While

Ruby tem algumas outras formas de laços. Esta é como fazer um laço ***while***:

```
while tired
  sleep
end
```

Ou, colocado de outra forma:

```
sleep while tired
```

Embora a sintaxe destes dois exemplos sejam diferentes, eles desempenham a mesma função. No primeiro exemplo, o código entre ***while*** e ***end*** (uma chamada para o método ***sleep***) executa enquanto a condição lógica (a qual, neste caso, é o valor retornado pela chamada do método ***tired***) for verdadeira. Como nos laços ***for*** a palavra-chave ***do*** pode, opcionalmente, ser colocada entre a

condição de teste e o código a ser executado estiver em linhas separadas; a palavra-chave ***do*** é obrigatória quando a condição de teste e o código a executar forem colocados na mesma linha.

Modificadores While

No segunda versão do laço (***sleep while tired***), o código a ser executado (***sleep***) precede a condição de teste (***while tired***). Esta sintaxe é chamada de um 'modificador while'. Quando você quer executar muitas expressões usando esta sintaxe, você pode colocá-las entre as palavras-chave ***begin*** e ***end***.

```
begin
  sleep
  snore
end while tired
```

Este é um exemplo mostrando as várias alternativas de sintaxe:

```
$hours_asleep = 0

def tired
  if $hours_asleep >= 8 then
    $hours_asleep = 0
    return false
  else
    $hours_asleep += 1
    return true
  end
end

def snore
  puts( 'snore...' )
end
def sleep
  puts( “z” * $hours_asleep )
end

while tired do sleep end # laço while em uma única linha

while tired                    # laço while em múltiplas linhas
  sleep
end
```

```
sleep while tired              # modificador while em uma única
linha

begin                        # modificador while em múltiplas linhas
  sleep
  snore
end while tired
```

O último exemplo acima (modificador while em múltiplas linhas) merece uma consideração maior já que introduz um novo comportamento. Quando um bloco de código delimitado por begin e end precede um teste while, o código sempre executa pelo menos uma vez. Nos outros tipos de teste while, o código pode nunca executar se a condição lógica inicialmente for falsa.

Assegurando que um laço execute pelo menos uma vez

Usualmente, um laço ***while*** executa 0 ou mais vezes desde que o teste lógico seja avaliado antes que o laço execute; se o teste retorna falso no início, o código interno do laço não será executado.
Porém, quando o teste ***while*** vem depois de um bloco de código delimitado por ***begin*** e ***end***, o laço executará 1 ou mais vezes já que a condição de teste é avaliada somente após o código interno do laço executar.
Para verificar as diferenças de comportamento destes dois tipos de laço ***while***, rode o programa ***while2.rb***. Estes exemplos devem ajudar no entendimento:

```
x = 100

# o código neste laço nunca executa
while ( x < 100 ) do puts('x < 100') end

# o código neste laço nunca executa
puts('x < 100') while ( x < 100 )

# mas o código neste laço executa uma vez
begin puts('x < 100') end while ( x < 100 )
```

Laços Until

O Ruby também tem um laço ***until*** o qual pode ser entendido como uma laço '***while not***'. Sua sintaxe e opções são as mesmas aplicadas ao ***while*** – isto é, o teste e o código a ser executado podem ser colocados em uma única linha (neste caso a palavra-chave ***do*** é obrigatória) ou eles podem ser colocados em linhas separadas (neste caso o ***do*** é opcional). Existe também um modificador ***until*** o qual permite você colocar o código antes da condição de teste; e existe, também, a opção de colocar o código entre um ***begin*** e ***end*** para assegurar que código execute pelo menos uma vez.

Aqui estão alguns exemplos simples de laços ***until***:

```
i = 10

until i == 10 do puts( i ) end          # nunca executa

until i == 10                       # nunca executa
  puts( i )
  i += 1
end

puts( i ) until i == 10             # nunca executa

begin                           # executa uma vez
  puts(i)
end until i == 10
```

Ambos os laços while e until podem, assim como o laço for, ser usados para iterar sobre matrizes (arrays) e outras coleções. Por exemplo, o código abaixo mostra como iterar sobre todos os elementos de uma matriz:

```
while i < arr.length
  puts( arr[i] )
  i += 1
end

until i == arr.length
  puts( arr[i] )
  i += 1
end
```

Cap. 8 – Declarações Condicionais

Declarações Condicionais...

Programas de computadores, como a vida real, estão cheios de decisões difíceis aguardando para serem tomadas. Coisas como: Se eu ficar na cama eu vou dormir um pouco mais, senão eu devo ir para o trabalho; se eu for para o trabalho e vou ganhar algum dinheiro, senão eu vou perder o meu emprego – e assim por diante...

Nós já fizemos alguns testes ***if*** nos programas anteriores neste livro. Peguemos um simples exemplo, o calculador de impostos do capítulo 1:

```
if (subtotal < 0.0 ) then
  subtotal = 0.0
end
```

Neste programa, o usuário foi solicitado a digitar um valor, **subtotal**, o qual foi então usado para calcular o imposto devido sobre o valor. O pequeno teste acima assegura que o **subtotal** nunca será um valor negativo. Se o usuário, num momento de loucura, informar um valor menor que 0, o teste ***if*** detecta isso já que a condição (**subtotal < 0.0**) é avaliada como verdadeira, o que causa a execução do código entre o ***if*** e o ***end*** colocando o valor 0 na variável **subtotal**.

> **Igual uma vez = ou igual duas vezes == ?**
>
> Em comum com muitas outras linguagens de programação, o Ruby usa um sinal de igual para atribuir um valor '**=**' e usa dois para testar um valor '**==**' .

If.. Then.. Else..

Um teste ***if*** simples tem apenas um de dois resultados possíveis. Ou um pedaço de código é executado ou não, dependendo se o teste retorna verdadeiro ou falso.

Muitas vezes, você precisará ter mais que duas saídas possíveis. Vamos supor, por exemplo, que seu programa necessita seguir um curso de ação se o dia da semana é uma quarta-feira e um curso

de ação diferente se é fim de semana. Você pode testar estas condições adicionando uma seção ***else*** após a seção ***if***, como no exemplo a seguir:

```
if aDay == 'Saturday' or aDay == 'Sunday'
  daytype = 'weekend'
else
  daytype = 'weekday'
end
```

A condição ***if*** aqui é direta. Ela testa 2 condições possíveis:

1. Se o valor da variável, aDay, é igual a 'Saturday' (sábado) ou ..
2. Se o valor de aDay é igual a 'Sunday' (domingo).

Se uma destas condições for verdadeira então o código da linha seguinte será executado:

```
daytype = 'weekend'
```

Em todos os demais casos, o código depois do ***else*** será executado:

```
daytype = 'weekday'
```

Quando um teste ***if*** e o código a ser executado são colocados em linhas separadas, a palavra-chave ***then*** é opcional. Quando o teste e o código são colocados na mesma linha, a palavra-chave ***then*** (ou se você preferir resumir o código, o símbolo de dois pontos ':'), é obrigatória.

```
if x == 1 then puts( 'ok' ) end       # com 'then'

if x == 1 : puts( 'ok' ) end              # com :

if x == 1  puts( 'ok' ) end               # com erro de
sintaxe!
```

Um teste if não está restrito a avaliar somente duas condições. Vamos supor, por exemplo, que seu código necessite saber se um determinado dia é um dia útil ou um feriado. Todos os dias da semana são dias úteis (dia de trabalho); todos os sábados (Saturday) são dias de folga mas os domingos (Sunday) somente são dias de folga se você não estiver fazendo horas extras.
Esta é a minha primeira tentativa de escrever um teste para avaliar todas estas condições:

```
working_overtime = true
if aDay == 'Saturday' or aDay == 'Sunday' and not working_overtime
  daytype = 'holiday'
  puts( “Hurrah!” )
else
  daytype = 'working day'
end
```

Infelizmente, isto não teve o efeito esperado. Lembre-se que Saturday (sábado) é sempre dia de folga. Mas este código insiste que 'Saturday' é um dia útil. Isto ocorre porque o Ruby entende o teste como: *“Se o dia é Sábado e eu não estou fazendo horas extras, ou se o dia é Domingo e eu não estou fazendo horas extras”* onde eu realmente queria dizer *“Se o dia é Sábado; ou se o dia é Domingo e eu não estou fazendo horas extras”*.

A forma mais simples de resolver esta ambigüidade é colocar parênteses em volta de qualquer código que deva ser avaliado como uma única expressão, assim:

```
if aDay == 'Saturday' or (aDay == 'Sunday' and not
working_overtime)
```

And.. Or.. Not

O Ruby tem duas sintaxes diferentes para testar condições lógicas (verdadeiro/falso – true/false).

No exemplo acima, E usei os operadores da língua inglesa: **and**, **or** and **not**. Se você preferir você pode usar operadores alternativos similares aos usados em outras linguagens de programação, a saber: **&&** (and), || (or) and **!** (not).

Seja cuidadoso, os dois conjuntos de operadores não são completamente intercambiáveis. Eles tem diferente precedência o que significa que quando múltiplos operadores são usados em um único teste, as partes do teste podem ser avaliadas em ordem diferente dependendo de quais operadores você usa.

If.. Elsif

Não há dúvidas que haverá ocasiões que você precisará tomar múltiplas ações baseadas em múltiplas condições alternativas. Uma forma de fazer isso é avaliar uma condição ***if*** seguida de uma

série de outros testes colocados após a palavra-chave ***elseif***. O lote todo então é terminado usando a palavra-chave ***end***.

Por exemplo, aqui eu estou repetidamente solicitando informação ao usuário dentro de um laço ***while***; uma condição ***if*** testa se o usuário digitou 'q' (eu usei o método chomp() para remover o 'retorno de carro' do dado digitado); se 'q' não foi informado o primeiro ***elsif*** testa se foi digitado um número inteiro (**input.to_i**) maior que 800; se o teste falhar a próxima condição **elsif** testa se o número é menor ou igual a 800:

```
while input != 'q' do
  puts( “Informe um número entre 1 e 1000 ( ou 'q' para sair)” )
  print( “?- “ )
  input = gets().chomp()
  if input == 'q'
    puts( “Tchau...” )
  elsif input.to_i > 800
    puts( “Este é um valor muito alto!” )
  elsif input.to_i <= 800
    puts( “Eu não posso gastar isso.” )
  end
end
```

Este código tem um bug. Ele pede por um número entre 1 e 1000 mas ele aceita outros números, até mesmo palavras e caracteres diversos. Veja se você pode a escrever os testes para consertar o código acima.

O Ruby também tem uma forma reduzida de notação para o **if..then..else** no qual um ponto de interrogação **?** Substitui a parte **if..then** e o sinal de dois pontos **:** atua como o **else**...

```
<Condição de teste> ? <se verdadeira faça isto> : <senão faça isto>
```

Por exemplo:

```
x == 10 ? puts( “Igual a 10” ) : puts( “Algum outro valor” )
```

Quando a condição de teste é complexa (se usa **and**s and **or**s) você deve envolvê-la por parênteses.

Se os testes e os códigos extendem-se por várias linhas a **?** deve ser colocada na mesma linha da condição precedente e o **:** deve ser colocado na mesma linha do código seguinte à **?.**

Em outras palavras, se você colocar uma nova linha antes da **?** Ou do **:** isto irá gerar um erro de sintaxe. Veja abaixo um exemplo de um bloco de código multi-linhas válido:

```
(aDay == 'Saturday' or aDay == 'Sunday') ?
  daytype = 'weekend' :
  daytype = 'weekday'
```

Unless

Ruby também pode executar testes ***unless***, o qual é o oposto do teste ***if***:

```
unless aDay == 'Saturday' or aDay == 'Sunday'
  daytype = 'weekday'
else
  daytype = 'weekend'
end
```

Pense no unless com sendo uma forma alternativa de expressar 'if not'. O exemplo abaixo é equivalente ao código acima:

```
if !(aDay == 'Saturday' or aDay == 'Sunday')
  daytype = 'weekday'
```

```
else
  daytype = 'weekend'
end
```

Modificadores If e Unless

Você pode rever a sintaxe alternativa para os laços ***while*** no Capítulo 7. Em vez de escrever isso...

```
while tired do sleep end
```

... você pode escrever assim:

```
sleep while tired
```

Esta sintaxe alternativa, na qual a palavra-chave ***while*** é colocada entre o código a executar e a condição de teste é chamado de 'modificador ***while***'. Dessa mesma forma, Ruby também possui modificadores ***if*** e ***unless***. Aqui estão alguns exemplos:

```
sleep if tired

begin
  sleep
  snore
end if tired

sleep unless not tired

begin
  sleep
  snore
end unless not tired
```

A forma resumida desta sintaxe é útil quando, por exemplo, você repetidamente precisa tomar alguma ação bem definida se uma certa condição é verdadeira.
Aqui está como você poderia criticar o seu código através de depuração se uma constante DEBUG é verdadeira:

```
puts(“somevar = #{somevar}” ) if DEBUG
```

Declarações Case

Quando você precisa tomar uma variedade de diferentes ações baseado no valor de uma única variável, usar múltiplos testes if..elsif é verboso e repetitivo demais. Uma alternativa elegante é fornecida pela declaração **case**. Começa-se com a palavra case seguido do nome da variável a testar. Então vem uma série de seções when, cada qual especifica um valor 'gatilho' seguido de algum código. Este código executa somente quando o valor da variável for igual ao valor 'gatilho':

```
case( i )
  when 1 : puts( “É segunda” )
  when 2 : puts( “É terça” )
  when 3 : puts( “É quarta” )
  when 4 : puts( “É quinta” )
  when 5 : puts( “É sexta” )
  when (6..7) : puts( “Oba! É fim de semana!” )
  else puts( “Não é um dia válido” )
end
```

No exemplo acima, E usei : para separar cada teste when do código a executar, alternativamente você poderá usar a palavra-chave then.

```
When 1 then puts( “É segunda” )
```

O sinal de dois pontos : ou o ***then*** pode ser omitido se o teste e o código a executar estão em linhas separadas. Diferentemente das declarações ***case*** em linguagens semelhantes ao C, não é necessário usar a palavra chave ***break*** numa seção cujo teste deu verdadeiro para evitar a execução das seções restantes.
Em Ruby, uma vez que um teste ***case*** deu verdadeiro a seção correspondente é executada e então a declaração ***case*** *é encerrada*.

```
case( i )
 when 5 : puts(“É sexta” )
    puts( “... o fim de semana está chegando!” )
  wen 6 : puts( “É sábado” )
    # O código seguinte nunca executa
 when 5 : puts(“É sexta novamente” )
end
```

Você pode incluir muitas linhas de código entre cada condição ***when*** e você pode incluir múltiplos valores separados por vírgula para disparar um único bloco ***when***, como este:

```
  when (6, 7) : puts( “Oba! É fim de semana!” )
```

A condição em uma declaração ***case*** não está obrigada a ser uma variável simples; pode ser uma expressão como esta:

```
case (i + 1)
```

Você também pode usar tipos não inteiros como uma string.

Se múltiplos valores gatilhos são especificados em uma seção ***when***, eles podem ser de tipos variados – por exemplo, ambos string e inteiros:

```
when 1, 'Monday', 'Mon' : puts( “Epa, '#{i}' é segunda-feira” )
```

Aqui está um exemplo maior, ilustrando alguns dos elementos sintáticos mencionados recentemente:

```
case( i )
  when 1 : puts( “É segunda” )
  when 2 : puts( “É terça” )
  when 3 : puts( “É quarta” )
  when 4 : puts( “É quinta” )
  when 5 : puts( “É sexta” )
    puts( “... o fim de semana está chegando!” )
  when 6, 7)
    puts( “É sábado” ) if i == 6
    puts( “É domingo” ) if i == 7
     puts( “Oba! É fim de semana!” )
   # O código seguinte nunca executa
   when 5 : puts(“É sexta novamente” )
 else puts( “Não é um dia válido” )
end
```

Cap. 9 – Módulos e Mixins

Módulos e Mixins...

Como mencionado em um capítulo anterior, cada classe Ruby têm somente um 'pai' imediato, embora cada classe pai possa ter vários 'filhos'.
Restringindo a hierarquia de classes em um única linha de descendência, o Ruby evita alguns dos problemas que podem ocorrer nas linguagens (como C++) que permitem múltiplas linhas de descendência.
Quando classes têm muitos pais assim como muitos filhos e seus pais, e filhos, também podem ter muitos pais e filhos, você corre o risco de terminar em uma rede impenetrável em vez de uma hierarquia limpa e bem ordenada que é o desejável.
Não obstante, existem ocasiões que é útil para uma classe poder implementar características de mais de uma outra classe pré-existente. Por exemplo, uma Espada (Sword) pode ser um tipo de Arma (Weapon) mas também um tipo de Tesouro (Treasure); uma Casa pode ser um tipo de Edifício mas também um tipo de Investimento e assim por diante.

Um Módulo é Como Uma Classe...

A solução do Ruby para este problema é fornecida pelos 'Modules' (módulos). À primeira vista, um módulo parece muito com uma classe. Igual a uma classe ele pode conter constantes, métodos e classes.
Aqui está um módulo simples:

```
module MyModule

 GOODMOOD = "happy"

 BADMOOD = "grumpy"

 def greet
   return "I'm #{GOODMOOD}. How are you?"
 end

end
```

Como você vê, este módulo possui uma constante, GOODMOOD e um 'método de instância', greet.

Para tornar isto em uma classe você só precisaria substituir a palavra module, na sua definição, pela palavra classe.

Métodos de Módulo

Em adição aos métodos de instância um módulo também pode ter métodos de módulo que são precedidos pelo nome do módulo:

```
def MyModule.greet
  return "I'm #{BADMOOD}. How are you?"
end
```

Apesar de suas similaridades, existem duas características principais que as classes possuem mas os módulos não: instâncias e herança. Classes podem ter instâncias (objetos), superclasses (pais) e subclasses (filhos); módulos não podem ter nada disso. O que leva-nos à próxima questão: se você não pode criar um objeto a partir de um módulo, para que servem os módulos?
Esta é outra questão que pode ser respondida em duas palavras: namespaces e mixins. Os mixins do Ruby fornecem-nos uma forma para lidar com o "pequeno" problema da herança múltipla que eu mencionei anteriormente. Nós voltaremos aos 'mixins' em breve. Antes disso, vamos dar uma olhada nos namespaces.

Módulos como Namespaces

Você pode pensar em um módulo como um certo "empacotador" para um conjunto de métodos, constantes e classes. Os vários bits de código dentro do módulo compartilham o mesmo "espaço de nomes" - 'namespace' - que significa que eles são todos visíveis para cada um deles mas não são visíveis para o código exterior ao módulo. A biblioteca de classes do Ruby define vários módulos tais como Math e Kernel. O módulo Math contém métodos matemáticos como sqrt para retornar a raiz quadrada e constantes como PI. O módulo Kernel contém muitos dos métodos que nós temos usado desde o início como print, puts e gets.

Constantes
Constantes são como variáveis exceto que seus valores não mudam (ou não deveriam mudar). De fato, é (bizarramente!) possível mudar o valor de uma constante no Ruby mas isto não é nada recomendável e o Ruby irá avisá-lo se você o fizer. Observe que as constantes começam com uma letra maiúscula.

modules1.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

module MyModule
   GOODMOOD = "happy"
   BADMOOD = "grumpy"

     def greet
          return "I'm #{GOODMOOD}. How are you?"
     end

     def MyModule.greet
          return "I'm #{BADMOOD}. How are you?"
     end
end


puts("  MyModule::GOODMOOD")
puts(MyModule::GOODMOOD)
puts( "  MyModule.greet" )
puts( MyModule.greet )
```

Vamos assumir que nós temos este módulo:

```
module MyModule
  GOODMOOD = "happy"
  BADMOOD = "grumpy"

  def greet
     return "I'm #{GOODMOOD}. How are you?"
  end

  def MyModule.greet
     return "I'm #{BADMOOD}. How are you?"
  end
end
```

Nós podemos acessar as constantes usando os sinais :: assim:

```
puts(MyModule::GOODMOOD)
```

Nós podemos, similarmente, acessar métodos do módulo usando a notação de ponto – isto é, especificando o nome do módulo seguido do ponto "." e do nome do método. O seguinte deveria imprimir "I'm grumpy. How are you?" (Eu estou irritado. Como você está?):

```
puts( MyModule.greet )
```

"Métodos de Instância" de Módulos

Isto nos deixa com o problema de como acessar o método de instância, greet. Como os módulos definem um espaço de nomes fechado, o código externo ao módulo não poderá "ver" o método greet, então isto não funciona:

puts(greet)

Se esta fosse uma classe em vez de um módulo, é claro, criaríamos objetos da classe usando o método new – e cada objeto separado (cada "instância" da classe), poderia ter acesso aos métodos de instância. Mas, como eu disse antes, você não pode criar instâncias de módulos. Então como nós podemos usar os seus métodos de instância? Aí é que aqueles misteriosos mixins entram em cena…

modules2.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

module MyModule
   GOODMOOD = "happy"
   BADMOOD = "grumpy"

     def greet
          return "I'm #{GOODMOOD}. How are you?"
     end

     def MyModule.greet
          return "I'm #{BADMOOD}. How are you?"
     end
end
```

```
include MyModule
puts( greet )
puts( MyModule.greet )
```

Módulos inclusos ou 'Mixins'

Um objeto pode acessar os métodos de instância de um módulo bastando incluir o módulo usando o método **include**. Se você for incluir MyModule em seu programa, tudo no interior do módulo será colocado dentro do escopo corrente. Assim o método greet de MyModule será agora acessível:

```
include MyModule

puts( greet )
```

O processo de incluir um módulo em uma classe é também chamado de 'misturar' o módulo – o que explica porque os módulos incluídos são freqüentemente chamados de 'mixins'.

Quando você inclui objetos em uma classe, quaisquer objetos criados a partir da classe poderão usar os métodos de instância do módulo incluso como se tivessem sido definidos na própria classe.

modules3.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com


module MyModule
   GOODMOOD = "happy"
   BADMOOD = "grumpy"

     def greet
          return "I'm #{GOODMOOD}. How are you?"
     end

     def MyModule.greet
          return "I'm #{BADMOOD}. How are you?"
     end
end


class MyClass
   include MyModule
```

```
     def sayHi
          puts( greet )
     end

     def sayHiAgain
          puts( MyModule.greet )
     end

end

ob = MyClass.new
ob.sayHi
ob.sayHiAgain
puts(ob.greet)
```

Não somente os métodos desta classe acessam o método greet de MyModule, mas também quaisquer objetos filhos da classe podem acessá-lo, como em:

```
ob = MyClass.new
ob.sayHi
ob.sayHiAgain
puts(ob.greet)
```

Em suma, então, os módulos podem ser usados com um meio de agrupar, de juntar, métodos, constantes e classes relacionadas dentro de um mesmo escopo. Assim, podemos ver os módulos como unidades discretas de código que pode simplificar a criação de bibliotecas de código reusável.

modules4.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

module MagicThing
     def m_power
          return @power
     end

     def m_power=(aPower)
          @m_power=aPower
     end
end
```

```
module Treasure
     attr_accessor :value
     attr_accessor :insurance_cost
end

class Weapon
     attr_accessor :deadliness
     attr_accessor :power
end

class Sword < Weapon
  include Treasure
  include MagicThing

  attr_accessor :name
end

s = Sword.new
s.name = "Excalibur"
s.deadliness = 10
s.power = 20
s.m_power = "Glows when Orcs Appear"

puts(s.name)
puts(s.deadliness)
puts(s.power)
puts(s.m_power)
```

Por outro lado, você pode estar mais interessado em usar módulos como uma alternativa para herança múltipla. Retornando ao exemplo que eu mencionei no início do capítulo, vamos assumir que você tem uma classe Sword (espada) que não é somente um tipo de Weapon (arma) mas também um tipo de Treasure (tesouro). Pode ser Sword uma descendente da classe Weapon (logo ela herda seus métodos tais como letalidade e potência), mas ela também dever ter métodos da classe Treasure (tais como valor e custo de seguro). Se você define estes métodos dentro de um módulo Treasure em vez de uma classe Treasure, a classe Sword poderia incluir o módulo Treasure para adicionar (misturar - 'mix in') os métodos de Treasure aos próprios métodos da classe Sword.

Note que quaisquer variáveis que são locais para o módulo não podem ser acessadas de fora do módulo. Este é o caso mesmo se um método dentro do módulo tentasse acessar uma variável local e este método fosse chamado pelo código de fora do módulo – por exemplo, quando o módulo é misturado através de inclusão. O programa **mod_vars.rb** ilustra isso.

mod_vars.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

# local variables declared inside a module are not
# accessible outside the module - even when the module
# is mixed in.

x = 1                    # local to this program

module Foo
     x = 50              # local to module Foo

     # This can be mixed in but the variable x won't then be
visible
     def no_bar
       return x
     end

     def bar
     @x = 1000      # You can mix in methods with instance
variables, however!
          return  @x
     end
     puts( "In Foo: x = #{x}" )   # this can access its local x
end

include Foo

 puts(x)
# puts( no_bar )    # This can't access the module-local variable x
needed by
                         # the no_bar method
 puts(bar)
```

Incluindo Módulos de Arquivos

Até agora, nós temos misturados módulos que foram todos definidos dentro de um único arquivo fonte. Freqüentemente é mais útil definir os módulos em arquivos separados e inclui-los quando necessário. A primeira coisa que você tem que fazer para usar o código de outro arquivo é carregar

o arquivo usando o método ***require***, desta forma:

requiremodule.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

require( "testmod.rb")

puts( "  MyModule.greet" )
puts( MyModule.greet )
puts("  MyModule::GOODMOOD")
puts(MyModule::GOODMOOD)
# puts( greet ) #=> This won't work!

include MyModule #mix in MyModule
puts( "  greet" )
puts( greet )   #=> But now this does work!
puts( "  sayHi" )
puts( sayHi )
puts( "  sayHiAgain" )
puts( sayHiAgain )
sing
puts(12346)
```

testmod.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

module MyModule
   GOODMOOD = "happy"
   BADMOOD = "grumpy"

     def greet
          return "I'm #{GOODMOOD}. How are you?"
     end

     def MyModule.greet
          return "I'm #{BADMOOD}. How are you?"
     end

     def sayHi
```

```
          return greet
     end

     def sayHiAgain
          return MyModule.greet
     end
end

def sing
     puts( "Tra-la-la-la-la....")
end

puts( "module loaded")
sing
```

O arquivo requerido deve estar no diretório corrente, no caminho de procura do sistema operacional (path) ou em uma pasta listada na variável de array predefinida $:. Você pode adicionar um diretório a este array usando o método << desta forma:

```
$: << "C:/mydir"
```

O método ***require*** retorna **true** se o arquivo especificado é carregado com sucesso; senão ele retorna **false**. Em caso de dúvida, você pode simplesmente mostrar o resultado:

```
puts(require( "testmod.rb" ))
```

Módulo Predefinidos

Os seguintes módulos estão embutidos no interpretador do Ruby: Comparable, Enumerable, FileTest, GC, Kernel, Math, ObjectSpace, Precision, Process, Signal

O mais importante dos módulos predefinidos é o Kernel que, como já mencionei, fornece muitos dos métodos padrão do Ruby tais como **gets, puts, print e require**. Em comum com muitas das classes do Ruby, o módulo Kernel é escrito na linguagem C. Mesmo o Kernel sendo, de fato, 'embutido' no interpretador Ruby, conceitualmente ele pode ser considerado como um módulo misturado que, como um mixin normal do Ruby, torna seus métodos diretamente disponíveis para quaisquer classes que os requeiram; já que ele é misturado na classe Object, da qual todas as outras classes do Ruby descendem, os métodos do módulo Kernel são universalmente acessíveis.

Cap. 10 – Salvando Arquivos, Avançando...

Indo Além...

Nós cobrimos um pedação de chão nos nove capítulos anteriores mas, mesmo assim, nós apenas começamos a explorar as possibilidade de programação com o Ruby. Uma das áreas que nós nem mesmo chegamos a tocar é o desenvolvimento de aplicações para a web usando o framework Rails (popularmente conhecido como 'Ruby On Rails'). A boa notícia é que, desenvolver com Rails será muito mais fácil agora que você tem um entendimento fundamental da programação em Ruby. Enquanto o Rails possui todos os tipos de ferramentas para colocar uma aplicação simples em uso, tentar programar em Rails sem entender Ruby seria como tentar escrever um romance sem conhecer bem a língua que se usará para escrevê-lo!
Nós também não vimos as características do Ruby relacionadas a um sistema operacional específico. Existem, por exemplo, muitos projetos em desenvolvimento destinados a fazer o Ruby rodar na plataforma Microsoft .NET. Existem também bibliotecas e ferramentas as quais ajudam você criar interfaces gráficas para seus programas em Ruby.

Gravando dados

Agora é hora de finalizar este Pequeno Livro do Ruby. Vamos fazer isto vendo mais um exemplo – um pequeno banco de dados de Cds que permite você criar novos objetos (um objeto para cada disco em sua coleção de Cds), adicioná-los a uma matriz e os armazená-los em disco.

Para gravar os dados no disco eu usei a biblioteca YAML do Ruby:

```
# saves data to disk in YAML format
def saveDB
   File.open( $fn, 'w' ) {
      |f|
      f.write($cd_arr.to_yaml)
   }
end
```

YAML

YAML descreve o formato para gravar dados como texto legível por nós humanos. Os dados podem ser depois recarregados do disco para construir uma matriz de objetos CD na memória do computador:

```
def loadDB
   input_data = File.read( $fn )
   $cd_arr = YAML::load( input_data )
end
```

Muito da codificação deste pequeno programa deve ser familiar – são oriundos de nossos exemplos anteriores. Algumas coisas devem ser ressaltadas, porém. Primeiro, as variáveis iniciando com um sinal $ são globais logo são usadas por todo o código do programa (lembre-se das variáveis de instância, que iniciam com @, que são usadas somente por um objeto específico; enquanto variáveis locais, que iniciam com uma letra minúscula, somente são usadas dentro de um escopo bem definido como um método específico).

Arquivos

Também note que nós usamos a classe File para verificar se um arquivo existe:

```
if File.exist?( $fn )
```

Aqui, **exist?** é um 'método de classe' – isto é, está ligado à classe File e não a uma instância da classe File. O que explica como nós invocamos o método diretamente de File em vez de chamá-lo de um novo objeto criado de File. Isto deve tê-lo feito se lembrar dos métodos de módulos discutidos no Capítulo Nove – outro exemplo de semelhanças entre módulos e classes.

data_save.rb

```
# Ruby Sample program from www.sapphiresteel.com /
www.bitwisemag.com

require 'yaml'

#array for cd objects
$cd_arr = Array.new

# file name for saving/loading
```

```
$fn = 'cd_db.yml'

# ancestor generic CD class
class CD

    # initialize variables parsed from array argument
    # ( arr )
    def initialize( arr )
        @name          = arr[0]
        @artist        = arr[1]
        @numtracks     = arr[2]
    end

    # return an array containing all instance variables
    # of a CD object
    def getdetails
        return[@name, @artist, @numtracks]
    end
end

# PopCD is a 'child' class of CD
class PopCD < CD

    # call super to pass array, arr, to parent (CD) class
    # This initializes three instance variables. This
    # method the initializes one more: @genre
    def initialize( arr )
        super( arr  )
        @genre = arr[3]
    end

    # calls sup to get arra from CD class getdetails
    # method. Then adds @genre to the end of the array
    # and returns the entier 4-item array
    def getdetails
        return( super << @genre )
    end
end

class ClassicalCD < CD
    def initialize( arr )
        super( arr )
        @conductor  = arr[3]
        @composer   = arr[4]
```

```
    end

    def getdetails
        return( super << @conductor << @composer )
    end
end


# some methods to get data from the user
def otherCD
    print( "Enter CD name: " )
    cd_name = gets().chomp()
    print( "Enter artist's name: " )
    a_name = gets().chomp()
    print( "Enter number of tracks: " )
    num_tracks = gets().chomp().to_i
    return [cd_name, a_name, num_tracks ]
end

def classicCD
    cdInfo= otherCD
    print( "Enter conductor's name: " )
    con_name= gets().chomp()
    print( "Enter composer's name: " )
    comp_name = gets().chomp()
    cdInfo << con_name << comp_name
    return cdInfo
end

def popCD
    cdInfo= otherCD
    print( "Enter genre of music: " )
    genre = gets().chomp()
    cdInfo << genre
    return cdInfo
end

# adds a CD object to the array variable, $cd_arr
def addCD( aCD )
    $cd_arr << aCD
end

# saves data to disk in YAML format
# the to_yaml method converts to YAML format
```

```
def saveDB
    File.open( $fn, 'w' ) {
        |f|
        f.write($cd_arr.to_yaml)
    }
end

# loads data from disk and recreates the array of
# cd objects, $cd_arr, from the data
def loadDB
    input_data = File.read( $fn )
    $cd_arr = YAML::load( input_data )
end

# prints the data from the array to screen in human-
# readable (YAML) format
def showData
    puts( $cd_arr.to_yaml )
end


# Start of Program

if File.exist?( $fn ) then
    loadDB
    showData
else
    puts( "The file #{$fn} cannot be found.")
end


# 'main' loop
ans = ''
until ans == 'q' do
  puts( "Create (P)op CD (C)lassical CD, (O)ther CD - (S)ave or
(Q)uit?" )
  print( "> " )
  ans = gets[0].chr().downcase()
  case ans
     when 'p' then addCD( PopCD.new( popCD() ) )
     when 'c' then addCD( ClassicalCD.new( classicCD() ) )
     when 'o' then addCD( CD.new( otherCD() ) )
    when 's' then saveDB
  end
```

```
  showData
end
```

Avançando...

A comunidade Ruby é muito ativa atualmente e novos projetos estão surgindo constantemente. Para manter-se atualizado, nós sugerimos que você visite o site Sapphire In Steel (www.sapphiresteel.com) para encontrar os links de alguns dos mais úteis recursos para os programadores Ruby. Nós também devemos adicionar mais tutoriais e projetos exemplo ao site para continuar nossa exploração da programação em Ruby.

Concluindo, eu espero que você tenha aproveitado esta pequena introdução à linguagem Ruby e que possa ser apenas o início de muitos anos de um prazeroso e produtivo desenvolvimento em Ruby.