while e until

`while` e `until` são conhecidos de qualquer linguagem imperativa. O `while` executa um bloco associado de código enquanto a condição estiver satisfeita:

```
a = [1, 2, 3]

while a.length > 0 do
  puts "Bye bye, #{a.pop}"
end
# Bye bye, 3
# Bye bye, 2
# Bye bye, 1
# => nil
```

O `until` ("até que" em inglês) faz justamente o contrário. Ele executa um bloco associado de código até que a situação se faça verdadeira:

```
a = [1, 2, 3]

until a.empty? do
  puts "Bye bye, #{a.pop}"
end
# Bye bye, 3
# Bye bye, 2
# Bye bye, 1
# => nil
```

CUIDADOS COM `until`

O `until` possui o mesmo problema semântico do `unless` (seção 2.4), ou seja, podemos ter problemas ao pensar em dupla negativa. Dessa forma, recomenda-se usar apenas `while`, negando a expressão quando pertinente.

---

for ... in

O `for ... in` é uma maneira de iterar elementos de uma coleção, ou um objeto "iterável", ou seja, um objeto que seja um `Enumerator`. A estrutura do `for` é simples:

`for` *`variável`* `in` *`coleção`*

Vejamos um exemplo com Arrays, que são “iteráveis":

```
fruits = %w{pera uva maçã}

for fruit in fruits
  puts "Gosto de " + fruit
end
# Gosto de pera
# Gosto de uva
# Gosto de maçã
# => ["pera", "uva", "maçã"]
```

Hashes também são iteráveis, com uma pequena diferença:

```
frequencies = {'hello' => 10, 'world' => 20}

for word, frequency in frequencies
  puts "A frequência da palavra '#{word}' é #{frequency}"
end
# A frequência da palavra 'hello' é 10
# A frequência da palavra 'world' é 20
# => {"hello"=>10, "world"=>20}
```

Blocos

Blocos são estruturas singulares a Ruby, e são extremamente importantes, tanto é que merecem atenção especial. Blocos são trechos de código associados a um método que podem ser executados a qualquer momento por este método.

Este conceito é um pouco estranho para quem vem de linguagens como Java (apesar de que Java possui uma forma de se fazer blocos, com classes anônimas) e C, mas se você já ouviu falar de *lambdas*, está familiarizado com o conceito.

O exemplo mais clássico de blocos é o uso de iteradores:

```
fruits = %w{pera uva maçã}

fruits.each do |fruit|
  puts "Gosto de " + fruit
end
# Gosto de pera
# Gosto de uva
# Gosto de maçã
# => ["pera", "uva", "maçã"]
```

Explicando o exemplo anterior, o método `#each` pega cada valor dentro de sua coleção e repassa para o bloco associado, através da variável `fruit`.

Outro método que usa bloco bastante útil é o `#map`, de Hashes e Arrays, que retorna um novo Array contendo como elementos o resultado devolvido a cada iteração:

```
numbers = [1, 2, 3, 4]

squared_numbers = numbers.map { |number| number * number }
squared_numbers  # [1, 4, 9, 16]
```

Notação para blocos

Como você pôde perceber, foram usadas duas notações distintas nos exemplos anteriores, a notação com `do ... end` e a notação com chaves. Ambas funcionam da mesma maneira. Porém, a comunidade Ruby adotou a seguinte regra:

- Para blocos curtos, de apenas uma linha, adota-se as chaves;
- Para blocos longos, de duas ou mais linhas, adota-se o `do ... end`.

A grande utilidade de blocos é que existem inúmeras APIs que exigem operações antes e depois de serem utilizadas. Com o uso de blocos, este procedimento fica transparente, como é possível observar no próximo exemplo:

```
# Exemplo tradicional
file = File.new('file.txt', 'w')
file.puts "Escrevendo no arquivo"
file.close

# Ao invés de termos que nos preocupar com a abertura e o fechamento do
# arquivo, a própria API faz isso antes e depois do bloco.
File.open('another_file.txt', 'w') do |file|
  file.puts "Escrevendo no arquivo com blocos!"
end
```