Aula 11

Arrays

Prof. Jalerson Lima

Apresentação

Nessa aula iremos estudar as estruturas de dados conhecidas como *arrays*. Aprenderemos o que são, como funcionam e como manipular dados armazenados em *arrays* através de métodos e operadores auxiliares, bem como exercitar nossos conhecimentos através de atividades práticas.

Objetivos

1. Compreender o que são e como funcionam os *arrays*;
2. Aprender como criar, armazenar e ler dados armazenados em *arrays*;
3. Aprender como manipular *arrays* utilizando métodos e operadores auxiliares;
4. Exercitar os conhecimentos através de atividades práticas.

1. Introdução

Até a presente aula, a única forma que temos de armazenar dados na memória é através de variáveis simples, conforme aprendemos na Aula 03. Nessa aula iremos aprender a trabalhar com *arrays*, que são estruturas capazes de armazenar dados organizamos numa sequência, e que nos fornece métodos para que possamos manipular esses dados.

Observe a Figura 1 que mostra uma representação gráfica de uma variável de nome "idade" e que armazena o valor 9 na memória do computador. Se for necessário armazenar outro valor, 10 por exemplo, será preciso criar uma outra variável para armazená-lo, pois se tentarmos utilizar a mesma variável, perderemos o valor que está guardado nela.

Figura 1 – Representação de uma variável na memória

Para armazenar vários valores, o indicado é utilizar um *array*, pois ele é uma estrutura capaz de armazenar um conjunto de valores sem a necessidade de criar várias variáveis. Observe a Figura 2, que ilustra uma representação gráfica de um *array* na memória do computador. Observe que o *array* é capaz de guardar vários valores com apenas uma referência para a memória (idades).

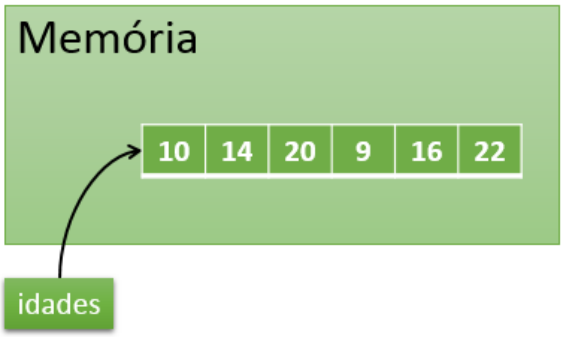

Figura 2 – Representação gráfica de um *array* na memória

Para melhor ilustrar a estrutura de um *array* e como eles funcionam, observe a Figura 3.

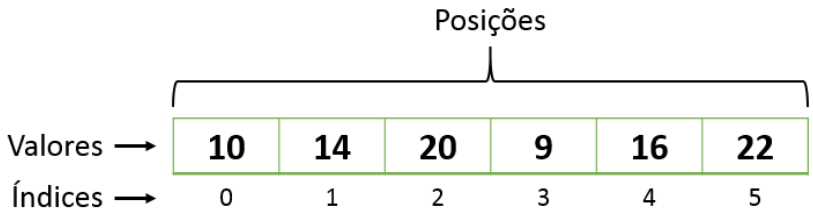

Figura 3 – Estrutura de um *array*

Conforme já explicado anteriormente, o *array* é uma estrutura capaz de guardar um conjunto de valores. Cada valor é guardado numa posição do *array* e cada posição é endereçada (também chamamos de indexada) por um índice, que é um número inteiro que marca cada posição no *array*. Portanto, conforme ilustrado na Figura 3, o valor da posição 0 é 10, o valor da posição 1 é 14, o valor da posição 2 é 20, etc. O primeiro índice sempre será 0 (zero), portanto num *array* com X posições, o último índice será sempre X – 1.

2. Arrays em Ruby

Em Ruby os *arrays* são dinâmicos, portanto você não precisa definir seu tamanho inicial, mas se preferir definir você também pode. Caso tenha criado um *array* com 10 posições e precise guardar um novo dado no *array*, você poderá fazê-lo sem problemas pois o *array* será expandido automaticamente conforme a necessidade.

Diferentemente de outras linguagens, em Ruby os *arrays* também podem guardar tipos de dados distintos, portanto você pode guardar uma *String* na primeira posição, um número inteiro na segunda, um número real na terceira, um objeto na quarta, etc.

2.1 Arrays em Ruby

2.1.1 Criando um array

Primeiramente vamos aprender algumas formas de criar um *array* em Ruby. Para isso, abra o IRB, digite a primeira linha do Exemplo de código 1 e observe a resposta dada pelo IRB (linha 2).

```
> meu_array = Array.new
=> []
```

Exemplo de código 1 – Criando um *array* em Ruby

Digitando `meu_array = Array.new` (atenção para o A maiúsculo de `Array`), estamos criando um novo *array* vazio e guardado ele na variável `meu_array`. Sim, são as variáveis que guardam os *arrays*, que por sua vez são capazes de guardar conjuntos de valores. Vale salientar que o nome da variável é escolhido por você e deve obedecer às regras de nomenclatura de variáveis apresentadas na Aula 3. O `[]` dado como resposta na linha 2 do Exemplo de código 3 indica que um *array* vazio foi criado.

Atividade 11.1

Abra o IRB e crie um novo *array* vazio, conforme ilustrado no Exemplo de código 1.

O Exemplo de código 3 apresenta como criar um *array* com 10 posições utilizando `Array.new`.

```
> meu_array = Array.new(10)
=> [nil, nil, nil, nil, nil, nil, nil, nil, nil, nil]
```

Exemplo de código 2 – Criando um *array* em Ruby

Observe que a resposta dada pelo IRB indica que foi criado um novo *array* com 10 posições, na qual cada uma possui o valor `nil` (nulo).

Atividade 11.2

Abra o IRB e crie um novo *array* vazio com cinco posições usando `Array.new`, conforme ilustrado no Exemplo de código 2.

Outra forma de criar um *array* é ilustrada no Exemplo de código 3.

```
> meu_array = []
=> []
```

Exemplo de código 3 – Criando um *array* em Ruby

Atribuindo `[]` a uma variável estamos criando um novo *array* vazio.

Atividade 11.3

Abra o IRB e crie um novo *array* vazio usando `[]`, conforme ilustrado no Exemplo de código 3.

Utilizando essa mesma sintaxe podemos criar um *array* com valores iniciais, conforme apresenta o Exemplo de código 4.

```
> meu_array = [10, 20, 30, 40, 50]
=> [10, 20, 30, 40, 50]
```

Exemplo de código 4 – Criando um *array* em Ruby

Observe, na resposta do IRB (linha 2), que foi criado um novo *array* com os valores definidos na primeira linha.

Atividade 11.4

Abra o IRB e crie um *array* com valores iniciais quaisquer, conforme ilustrado no Exemplo de código 4.

Conforme explicado anteriormente, os *arrays* em Ruby permitem guardar diferentes tipos de dados. Confira essa situação no Exemplo de código 6.

```
> meu_array = [10, "vinte", true, nil, 3.14]
=> [10, "vinte", true, nil, 3.14]
```

Exemplo de código 5 – Criando um *array* em Ruby

No exemplo anterior, criamos um *array* com cinco posições: a primeira contém um número inteiro, a segunda contém uma *String*, a terceira contém um valor booleano, a quarta contém um nulo e a quinta contém um número real.

Atividade 11.5

Abra o IRB e crie um novo array com valores iniciais quaisquer, mas sendo de tipos diferentes, conforme ilustra o Exemplo de código 5.

2.1.2 Guardando valores no array

Agora vamos aprender como guardar valores e acessar valores de um *array*. Existem pelo menos duas formas de se adicionar valores num *array*: você pode especificar uma posição (índice) do *array* para guardar o novo valor ou usar a próxima posição disponível. Inicialmente vamos aprender como guardar um valor no *array* usando a próxima posição disponível.

Para melhor ilustrar, vamos utilizar como exemplo o *array* apresentado no Exemplo de código 4, que é apresentado graficamente na Figura 4.

Figura 4 – Adicionando novos valores ao *array*

O *array* que estamos usando como exemplo possui cinco posições, e todas elas já estão ocupadas com valores. Portanto, qual é a próxima posição disponível nesse *array*? A posição com índice 5! Lembre-se que os *arrays* em Ruby são automaticamente expansíveis, portanto você pode adicionar novos valores mesmo que não haja posições vazias.

Observe o Exemplo de código 6, que ilustra como adicionar um novo valor ao *array*.

```
> meu_array << 60
=> [10, 20, 30, 40, 50, 60]
```

Exemplo de código 6 – Adicionando um novo elemento na próxima posição disponível

Conforme ilustrado na linha 1 do Exemplo de código 6, o operador `<<` permite a adição de um novo valor ao *array*. Esse novo valor será adicionado na próxima posição disponível, que nesse caso é a sexta posição (índice 5). Observe a resposta do IRB na linha 2, que mostra que o novo valor foi adicionado no final do *array* (índice 5).

Atividade 11.6

Abra o IRB, crie um *array* com valores iniciais quaisquer e adicione um novo elemento usando o operador `<<`, conforme ilustrado no Exemplo de código 6.

A segunda forma que iremos apresentar para adicionar novos elementos num *array* é aquela em que o novo elemento é colocado numa posição (índice) específica. Observe como fazer isso no Exemplo de código 7.

```
> meu_array
=> [10, 20, 30, 40, 50, 60]
> meu_array[0] = "dez"
=> ["dez", 20, 30, 40, 50, 60]
```

Exemplo de código 7 – Adicionando um novo elemento numa posição específica

Observe, no Exemplo de código 7, que estamos usando o mesmo *array* produzido no exemplo anterior. Para adicionar um elemento numa posição específica do *array*, basta definir

o índice entre [e], conforme ilustrado na linha 3. Nesse exemplo, adicionamos "dez" na primeira posição (índice 0) do *array*, substituindo o valor que havia nessa posição.

Atividade 11.7

Abra o IRB, crie um *array* com valores iniciais quaisquer e adicione um novo elemento numa posição específica, conforme ilustrado no Exemplo de código 7.

2.1.3 Acessando valores do array

Para acessar valores armazenados em um *array*, basta usar o nome da variável que guarda o *array*, seguido do índice da posição que se deseja acessar entre [e]. Observe, no Exemplo de código 8, como acessar o valor armazenado na segunda posição (índice 1).

```
> meu_array
=> ["dez", 20, 30, 40, 50, 60]
> meu_array[1]
20
```

Exemplo de código 8 – Acessando valores armazenados no *array*

Caso você acesse uma posição que não guarda valor algum, o resultado será nil.

Atividade 11.8

Abra o IRB, crie um *array* com valores iniciais quaisquer e depois acesse uma posição qualquer do *array*, conforme ilustrado no Exemplo de código 8.

2.1.4 Métodos e operadores auxiliares

Existem alguns métodos e operadores que auxiliam a manipulação de dados em *arrays*. Nessa seção iremos apresentar apenas alguns deles, mas você pode obter maiores informações sobre eles e sobre outros métodos e operadores na página oficial da documentação do Ruby: http://docs.ruby-lang.org/en/2.0.0/Array.html (em inglês).

Para ilustrar os métodos e operadores auxiliares, iremos utilizar os dois arrays apresentados no Exemplo de código 9.

```
> array1 = [1, 2, 3]
=> [1, 2, 3]
> array2 = [3, 4, 5]
=> [3, 4, 5]
```

Exemplo de código 9 – *Arrays* que serão usados como exemplo

O operador `&`, ilustrado no Exemplo de código 10, retorna (dá como resultado) um novo *array* contendo os elementos em comum nos dois *arrays* (`array1` e `array2`), sem duplicatas.

```
> array1 & array2
=> [3]
```

Exemplo de código 10 – Operador &

O operador `+`, ilustrado Exemplo de código 11, retorna um novo *array* produzido pela concatenação (união) dos dois arrays (`array1` e `array2`).

```
> array1 + array2
=> [1, 2, 3, 3, 4, 5]
```

Exemplo de código 11 – Operador +

O operador `-`, ilustrado Exemplo de código 12, retorna uma cópia do `array1` removendo os elementos que também constam no `array2`.

```
> array1 - array2
=> [1, 2]
```

Exemplo de código 12 – Operador -

O operador `==`, ilustrado no Exemplo de código 13, irá retornar verdadeiro caso o `array1` e o `array2` tenham a mesma quantidade de elementos e se cada elemento de `array1` for igual ao seu correspondente (mesma posição) no `array2`.

```
> array1 == array2
=> false
```

Exemplo de código 13 – Operador ==

O método `clear`, ilustrado no Exemplo de código 14, remove todos os elementos do `array1`.

```
> array1.clear
=> []
```

Exemplo de código 14 – Método *clear*

O método `delete`, ilustrado no Exemplo de código 15, recebe um elemento como parâmetro e o remove do `array1`. O método irá retornar `nil` caso o elemento passado como parâmetro não seja encontrado no *array*.

```
> array1.delete(1)
=> 1
```

Exemplo de código 15 – Método *delete*

O método `delete_at`, ilustrado no Exemplo de código 16, recebe um índice (número inteiro) como parâmetro e remove o elemento armazenado naquela posição. O método irá retornar `nil` caso o índice passado como parâmetro esteja fora das dimensões do *array*.

```
> array2.delete_at(1)
=> 4
```

Exemplo de código 16 – Método *delete_at*

O método `empty?`, ilustrado no Exemplo de código 17, irá retornar verdadeiro caso o `array1` esteja vazio e falso caso contrário.

```
> array1.empty?
=> false
```

Exemplo de código 17 – Método *empty?*

O método `include?`, ilustrado no Exemplo de código 18, recebe um elemento como parâmetro e retorna verdadeiro caso esse elemento esteja presente dentro do `array1`, e falso caso contrário.

```
> array1.include?(1)
=> true
```

Exemplo de código 18 – Método *include?*

O método `size`, ilustrado no Exemplo de código 19, retorna o número de elementos armazenados no `array2`.

```
> array2.size
=> 3
```

Exemplo de código 19 – Método *size*

2.1.5 Iterando em arrays

É muito comum, em diversas situações, precisarmos iterar, ou seja, verificar cada um dos elementos de um *array*, e fazer algum processamento se for necessário. Nessa seção, iremos apresentar como iterar entre os elementos de um *array* usando alguns dos laços já apresentados em aulas anteriores.

Confira o Exemplo de código 20 que ilustra como iterar entre os elementos de um *array* usando o *each*.

```
1  meu_array = [10, 20, 30, 40, 50]
2
3  meu_array.each do |elemento|
4    puts elemento
5  end
```

Exemplo de código 20 – Iterando entre os elementos do *array* com *each*

Atividade 11.7

Crie um *script* em Ruby com o código apresentado no Exemplo de código 20, execute-o e observe o resultado.

Na linha 1, criamos um *array* com cinco elementos numéricos quaisquer. Na linha 3, usamos o método `each` no *array* para que possamos iterar entre seus elementos. A cada iteração do laço, um dos valores armazenados no *array* será atribuído à variável `elemento` (definida entre `|` e `|`). Portanto, nesse caso, na primeira iteração o valor de `elemento` será 10; na segunda iteração, o valor de `elemento` será 20; na terceira iteração, o valor de `elemento` será 30, e assim sucessivamente, até que a variável `elemento` tenha assumido o valor de cada posição do `meu_array`. Vale salientar que, o nome da variável `elemento` é definida pelo próprio programador, obedecendo as normas de nomenclatura de variáveis.

Outra forma de iterar entre os elementos de um *array* com o *each* é usando a notação de chaves.

```
1  meu_array = [10, 20, 30, 40, 50]
2
3  meu_array.each { |elemento|
4    puts elemento
5  }
```

Exemplo de código 21 – Iterando entre os elementos do *array* com *each* e chaves

Atividade 11.8

Crie um *script* em Ruby com o código apresentado no Exemplo de código 21, execute-o e observe o resultado.

Novamente o nome da variável auxiliar é definido entre `|` e `|`, contudo, ao invés de usar o `do` e o `end` como marcadores de início e fim do bloco de código, estamos usando as chaves (`{` e `}`). A variável `elemento` irá assumir cada valor armazenado no `meu_array`. Outra forma de iterar entre os elementos de um *array* é usando a instrução *for*, conforme ilustra o Exemplo de código 22.

```
1  meu_array = [10, 20, 30, 40, 50]
2
3  for elemento in meu_array
4    puts elemento
5  end
```

Exemplo de código 22 – Iterando entre os elementos do *array* com *for*

Atividade 11.9

Crie um *script* em Ruby com o código apresentado no Exemplo de código 22, execute-o e observe o resultado.

Usando o *for*, o nome da variável auxiliar é definido logo após o `for`, e o nome da variável que guarda o *array* é colocado após o `in`. No Exemplo de código 22, a variável `elemento` irá assumir cada um dos valores armazenados no `meu_array`.

Atividade 11.10

a) Crie um *script* em Ruby que leia 10 nomes e armazene-os num *array*. Imprimir os nomes numa lista numerada;
b) Crie um *script* em Ruby que leia 15 números inteiros e guarde-os em um *array*. Depois, imprimir cada um dos números e dizendo se ele é par ou ímpar;
c) Crie um *script* em Ruby que leia e armazene 8 números inteiros em um *array* e imprima todos os números. Ao final, imprimir o total de números múltiplos de 6;
d) Crie um *script* em Ruby que leia e armazene 10 números inteiros em um *array*. Para cada valor guardado numa posição "p", verificar se o valor contido na posição "p-1" é divisor do valor guardado na posição "p";
e) Crie um *script* em Ruby que leia e armazene os nomes e os salários de 20 pessoas. Calcular e armazenar o novo salário, sabendo-se que houve um reajuste de 8%. Imprimir uma listagem com os nomes e o novo salário de cada funcionário;
f) Crie um *script* em Ruby que leia e armazene os nomes e as idades de 10 pessoas. Depois, o algoritmo deve imprimir o nome e a idade da pessoa mais nova, e o nome e a idade da pessoa mais velha;
g) Crie um *script* em Ruby que leia 5 números inteiros para o conjunto A, e 5 números inteiros para o conjunto B. Depois o *script* deve imprimir o conjunto intersecção entre A e B, ou seja, imprimir os valores que estão em A e em B;
h) Crie um *script* em Ruby que leia 10 números inteiros e armazene-os num *array*. Depois, o *script* deve calcular o fatorial de cada um desses 10 números, e armazenar os resultados num outro *array*. Depois, imprimir os valores contidos nesse segundo *array*;
i) Crie um *script* em Ruby que leia 10 números inteiros e armazene-os num *array*. Depois, o *script* deve ordenar esses números em ordem crescente e imprimi-los;
j) Crie um *script* em Ruby que leia 10 números inteiros e armazene-os num *array*. Depois, o *script* deve ordenar esses números em ordem decrescente e imprimi-los.

Resumindo

Essa aula apresentou o que são, como funcionam e como manipular dados armazenados em *arrays* utilizando a linguagem Ruby. Também estudamos como manipular dados e arrays utilizando métodos e operadores auxiliares, bem como trabalhamos os conhecimentos adquiridos através de exercícios práticos.

Referências

POINT, T. Ruby Tutorial. **Tutorials Point**, 2015. Disponivel em: <http://www.tutorialspoint.com/ruby/>. Acesso em: 12 nov. 2015.

RANGEL, E. **Conhecendo Ruby**. [S.l.]: Leanpub, 2014.

RUBY LANG. Class Array. **Ruby Documentation**, 2015. Disponivel em: <http://docs.ruby-lang.org/en/2.0.0/Array.html>. Acesso em: 04 fev. 2016.

SOUZA, L. **Ruby - Aprenda a programar na linguagem mais divertida**. 1ª. ed. São Paulo: Casa do Código, v. I, 2012.