

Na célula abaixo apenas estipulamos quais pacotes serão utilizados:

```
• begin
•     using Pkg
•     Pkg.add("Images")
•     using Images
• end
```

O Básico de Arrays

Na aula passada vimos como alguns conceitos da programação seguem uma linha lógica e padronizada, podendo assim abstrair alguns conceitos sobre *arrays*, agora, vamos ver como trabalhar melhor com essa estrutura

Inicializando Arrays

```
v = Int64[1, 2, 3, 4]
```

```
• v = [1, 2, 3, 4] # Vetor
```

```
(4)
```

```
• size(v)
```

Em Julia podemos usar uma sintaxe mais simples e próxima da matemática para definir *arrays*

```
w = 2×3 Array{Int64,2}:
```

```
 1 2 3
 4 5 6
```

```
• w = [1 2 3
      4 5 6]
```

```
(2, 3)
```

```
• size(w)
```

Acessando os elementos

Como em muitas outras linguagens, utiliza-se a estrutura `nome_do_array[i]` para acessar o elemento de endereço(indice) *i*

```
BoundsError: attempt to access 4-element Array{Int64,1} at index [0]
```

1. `getindex` @ `array.jl:809` [inlined]
2. `top-level scope` @ `| Local: 1` [inlined]

- `v[0]`

Se você já programou em outras linguagens irá notar que ocorre um erro quando tentamos acessar o elemento de índice **0** em algum *array* em Julia. Isso ocorre pois em Julia *arrays* começam pelo endereço **1**. Foi decidido assim porque Julia é uma linguagem mais focada em matemática e computação matemática, onde índices geralmente começam no **1**

1

- `v[1]`

1

- `w[1]`

Repare que se o *array* for de duas dimensões, e escrevermos apenas a primeira, será considerada a primeira posição na segunda dimensão (mesma coisa que escrever `w[1,1]`)

4

- `w[2]`

5

- `w[2,2]`

Se formos pegar toda a linha, podemos utiizar o caractere de :

```
Int64[1, 2, 3]
```

- `w[1,:]` # Vetor

Analogamente, pode-se pegar a coluna trocando a posição do :

```
Int64[2, 5]
```

- `w[:,2]` # Vetor

Podemos também dar *slices*(cortar) os *arrays* utilizando o caractere :

```
2×2 Array{Int64,2}:
```

```
2 3
5 6
```

- `w[:,2:3]` # Matriz

Gerando outros tipos de Arrays

A função `rand(intervalo, n, m)` gera uma matriz (n,m) com números aleatórios em um intervalo

```
A1 = 3×4 Array{Int64,2}:
 6 7 1 6
```

```
7  1  7  6
8  7  4  1
```

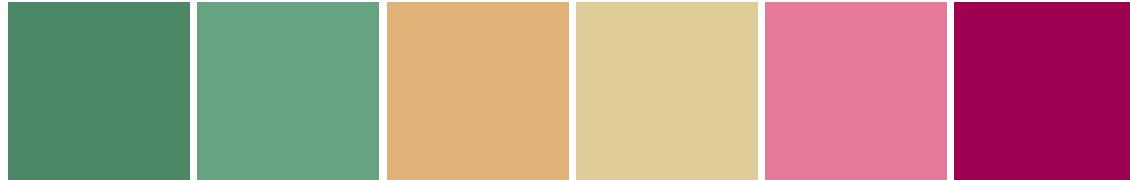
- `A1 = rand(1:9, 3, 4)`

`A2 = 3x4 Array{Int64,2}:`

```
2  7  3  4
8  1  1  5
8  5  7  3
```

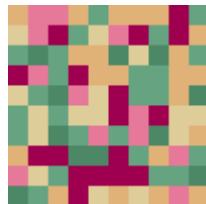
- `A2 = rand(1:9, 3, 4)`

`cores =`



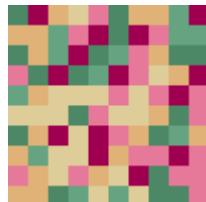
- `cores = [RGB(0.3, 0.53, 0.4), RGB(0.4, 0.64, 0.5), RGB(0.88, 0.7, 0.47), RGB(0.87, 0.8, 0.6), RGB(0.9, 0.47, 0.6), RGB(0.62, 0, 0.31)]`

`A3 =`



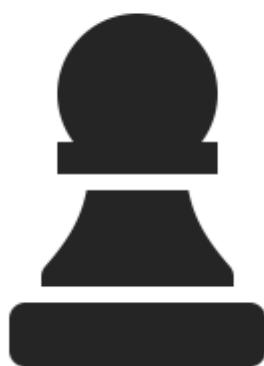
- `A3 = rand(cores, 10, 10)`

`A4 =`



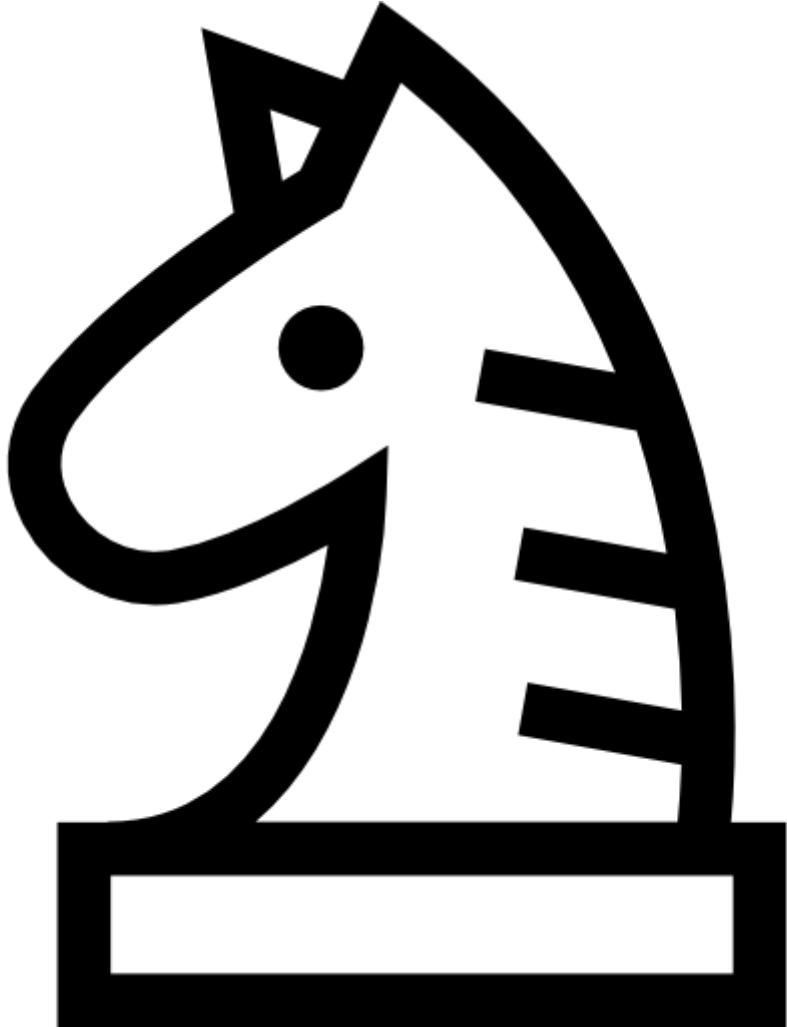
- `A4 = rand(cores, 10, 10)`

`peao =`



- `peao = load(download("https://d1nhio0ox7pgb.cloudfront.net/_img/o_collection_png/green_dark_grey/256x256/plain/chess_piece_pawn.png"))`

```
cavalo =
```



```
• cavalo = load(download("https://image.flaticon.com/icons/png/512/32/32648.png"))
```

```
A5 =
```



```
• A5 = rand([peao, cavalo], 3, 3)
```

Modificando os Arrays

3x4 Array{Int64,2}:

```
10000 7 1 6  
    7 1 7 6  
    8 7 4 1
```

- begin
- B = copy(A1)
- B[1,1] = 10000
- B
- end

C =



- C = fill(peao, 3, 3)



```

• begin
•     D = copy(C)
•     D[2,2] = cavalo
•     D
• end

```

Utilizando o *for loop* em arrays

Podemos utilizar o for loop como em outras linguagens:

4×4 Array{Int64,2}:

```

1 2 3 4
2 4 6 8
3 6 9 12
4 8 12 16

```

```

• begin
•
•     F = fill(0, 4, 4)
•
•     for i=1:4
•         for j=1:4
•             F[i,j] = i*j
•         end
•     end
•
•     F
• end

```

Podemos também fazer um loop duplo:

4×4 Array{Int64,2}:

```

1 2 3 4
2 4 6 8
3 6 9 12
4 8 12 16

```

```

• begin
•
•     F2 = fill(0, 4, 4)
•
•     for i=1:4, j=1:4
•         F2[i,j] = i*j
•     end
•
•     F2
• end

```

Ou melhor ainda, um loop compreendido (*list comprehension*):

F3 = 4×4 Array{Int64,2}:

```

1 2 3 4
2 4 6 8
3 6 9 12
4 8 12 16

```

```

• F3 = [i*j for i=1:4, j=1:4]

```

Operadores e operadores elemento-a-elemento

Como vimos brevemente na primeira aula, podemos usar o caractere `.` para indicar operações elemento-a-elemento

```
4x4 Array{Int64,2}:
 30   60   90  120
 60  120  180  240
 90  180  270  360
120  240  360  480
```

- `F3^2`

```
4x4 Array{Int64,2}:
 1   4    9   16
 4  16   36   64
 9  36   81  144
16  64  144  256
```

- `F3.^2 # Repare no .`

Repare que no primeiro caso multiplicamos a matriz por ela mesma, enquanto no segundo caso, elevamos cada elemento da matriz ao quadrado

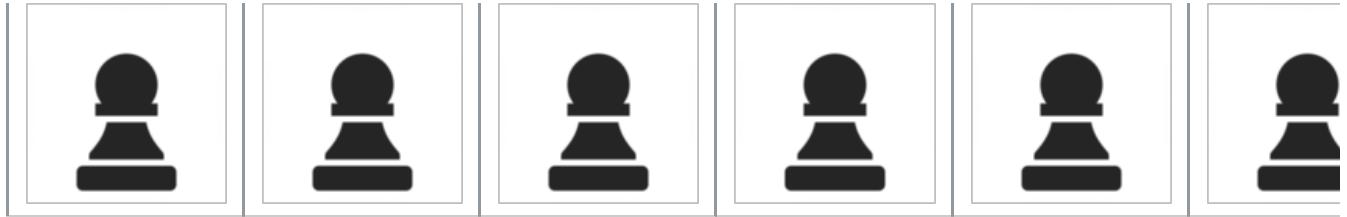
Concatenação de Arrays

Em Julia podemos facilmente concatenar(juntar) arrays os escrevendo como elementos de um array:

```
4x8 Array{Int64,2}:
 1  2    3    4   1   2    3    4
 2  4    6    8   2   4    6    8
 3  6    9   12   3   6    9   12
 4  8   12   16   4   8   12   16
```

- `[F2 F3]`

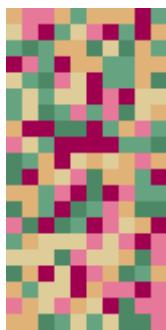




- [C D]



- [A3 A4]



- [A3 ; A4]



- [A3 A3 ; A4 A4]