# ÖGOR Summer-Workshop for PhD-candidates and Post-Docs

## An introduction to Julia and JuMP for Operations Research

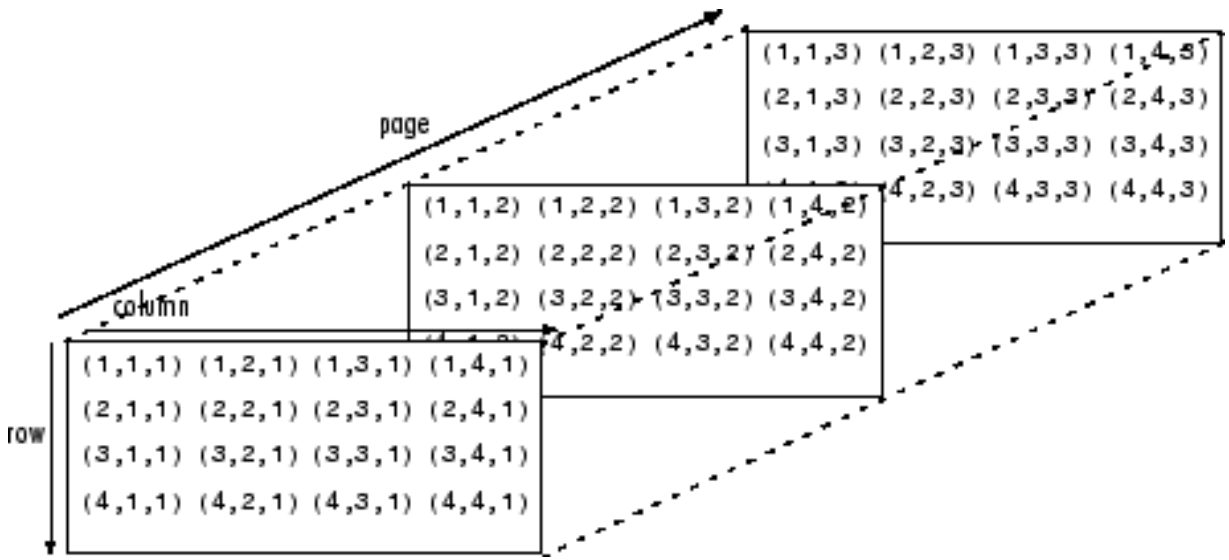
Prof. Dr. Xavier Gandibleux

Nantes Université – France
Département Informatique – Faculté des Sciences et Technique

Topic 6

Data structures (part 1)

# Arrays (vectors, matrices, lists)



# Constructing a vector (1/3)

**A vector is an array with 1 dimension**

Arrays contain ordered collections

Definition:

Let a collection of $n$ items: $item_1$, $item_2$, ..., $item_n$

$$[item_1, item_2, \ldots, item_n]$$

Julia is 1-based indexing

Examples:

```
julia> ["Austria", "France", "Belgium"]
```

```
julia> primeNumbers = [2, 3, 5, 7, 11]
```

```
julia> primeNumbers = [2, 3, 5, "seven", 11]
```

# Constructing a vector (1/3)

**A vector is an array with 1 dimension**

Arrays contain ordered collections

Definition:

Let a collection of $n$ items: $item_1$, $item_2$, ..., $item_n$

$$[item_1, item_2, \ldots, item_n]$$

Julia is 1-based indexing

Examples:

```
julia> ["Austria", "France", "Belgium"]
```

```
julia> primeNumbers = [2, 3, 5, 7, 11]
```

```
julia> primeNumbers = [2, 3, 5, "seven", 11]
```

# Constructing a vector (2/3)

Typed but non initialized vector:

```
Array{type}(undef,size)
```

```
Vector{type}(undef,size)
```

Examples:

```
julia> v = Array{Int64}(undef,5)
```

```
julia> v = Vector{Int64}(undef,5)
```

# Constructing a vector (2/3)

Typed but non initialized vector:

```
Array{type}(undef,size)
```

```
Vector{type}(undef,size)
```

Examples:

```
julia> v = Array{Int64}(undef,5)
```

```
julia> v = Vector{Int64}(undef,5)
```

# Constructing a vector (3/3)

Typed and initialized vector:

```
zeros(type,size)
```

```
ones(type,size)
```

See also `collect` and `fill` functions

Examples:

```
julia> v0 = zeros(Int64,5)
```

```
julia> v1 = ones(Int64,5)
```

```
julia> v3to6 = collect(3:6)
```

```
julia> v10 = fill(10,5)
```

# Constructing a vector (3/3)

Typed and initialized vector:

```
zeros(type,size)
```

```
ones(type,size)
```

See also `collect` and `fill` functions

Examples:

```
julia> v0 = zeros(Int64,5)
```

```
julia> v1 = ones(Int64,5)
```

```
julia> v3to6 = collect(3:6)
```

```
julia> v10 = fill(10,5)
```

# Accessing and editing a vector (1/3)

Access to individual data inside an array by indexing into the array:

```
nameArray[index]
```

```
julia> primeNumbers[3]
```

Access to the last data inside an array:

```
nameArray[end]
```

```
julia> primeNumbers[end]
```

Access to multiple consecutive data inside an array:

```
nameArray[indexStart:indexEnd]
```

```
julia> primeNumbers[3:4]
```

# Accessing and editing a vector (1/3)

Access to individual data inside an array by indexing into the array:

```
nameArray[index]
```

```
julia> primeNumbers[3]
```

Access to the last data inside an array:

```
nameArray[end]
```

```
julia> primeNumbers[end]
```

Access to multiple consecutive data inside an array:

```
nameArray[indexStart:indexEnd]
```

```
julia> primeNumbers[3:4]
```

# Accessing and editing a vector (1/3)

Access to individual data inside an array by indexing into the array:

```
nameArray[index]
```

```
julia> primeNumbers[3]
```

Access to the last data inside an array:

```
nameArray[end]
```

```
julia> primeNumbers[end]
```

Access to multiple consecutive data inside an array:

```
nameArray[indexStart:indexEnd]
```

```
julia> primeNumbers[3:4]
```

# Accessing and editing a vector (1/3)

Access to individual data inside an array by indexing into the array:

```
nameArray[index]
```

```
julia> primeNumbers[3]
```

Access to the last data inside an array:

```
nameArray[end]
```

```
julia> primeNumbers[end]
```

Access to multiple consecutive data inside an array:

```
nameArray[indexStart:indexEnd]
```

```
julia> primeNumbers[3:4]
```

# Accessing and editing a vector (2/3)

Edit an existing element of an array (arrays are mutable):

```
nameArray[index] = item
```

```
julia> primeNumbers[2] = "three"
```

Edit multiple consecutive element of an array:

```
nameArray[indexStart:indexEnd] = [item_1,...,item_n]
```

```
julia> v[2:3] = [3, 7]
```

# Accessing and editing a vector (2/3)

Edit an existing element of an array (arrays are mutable):

```
nameArray[index] = item
```

```
julia> primeNumbers[2] = "three"
```

Edit multiple consecutive element of an array:

```
nameArray[indexStart:indexEnd] = [item_1,...,item_n]
```

```
julia> v[2:3] = [3, 7]
```

# Accessing and editing a vector (2/3)

Edit an existing element of an array (arrays are mutable):

```
nameArray[index] = item
```

```
julia> primeNumbers[2] = "three"
```

Edit multiple consecutive element of an array:

```
nameArray[indexStart:indexEnd] = [item_1,...,item_n]
```

```
julia> v[2:3] = [3, 7]
```

# Accessing and editing a vector (3/3)

`push!` function adds an element to the end of an array

```
push!(nameArray,item)
```

```
julia> push!(primeNumbers,13)
```

`pop!` function removes the last element of an array

```
pop!(nameArray)
```

```
julia> pop!(primeNumbers)
```

# Accessing and editing a vector (3/3)

`push!` function adds an element to the end of an array

```
push!(nameArray,item)
```

```
julia> push!(primeNumbers,13)
```

`pop!` function removes the last element of an array

```
pop!(nameArray)
```

```
julia> pop!(primeNumbers)
```

# Accessing and editing a vector (3/3)

`push!` function adds an element to the end of an array

```
push!(nameArray,item)
```

```
julia> push!(primeNumbers,13)
```

`pop!` function removes the last element of an array

```
pop!(nameArray)
```

```
julia> pop!(primeNumbers)
```

# Constructing a matrix (1/4)

## A matrix as multi-dimensional array

Definition:

Let a collection of items stored in a (e.g.) $n \times m$ grid (2D array),

$$[t_{1,1}\ \ t_{1,2} \dots t_{1,m}\ ;\ t_{2,1}\ \ t_{2,2} \dots t_{2,m}\ ;\ \dots\ ;\ t_{n,1}\ \ t_{n,2}\ \dots t_{n,m}]$$

Examples:

```
julia> [1 2 ; 3 4]
```

```
julia> T = ['a' 'b' 'c' ; 'x' 'y' 'z']
```

# Constructing a matrix (1/4)

**A matrix as multi-dimensional array**

Definition:

Let a collection of items stored in a (e.g.) $n \times m$ grid (2D array),

$$[t_{1,1} \;\; t_{1,2} \ldots t_{1,m} \;\; ; \;\; t_{2,1} \;\; t_{2,2} \ldots t_{2,m} \;\; ; \;\; \ldots \;\; ; \;\; t_{n,1} \;\; t_{n,2} \;\; \ldots t_{n,m}]$$

Examples:

```
julia> [1 2 ; 3 4]
```

```
julia> T = ['a' 'b' 'c' ; 'x' 'y' 'z']
```

# Constructing a matrix (1/4)

**A matrix as multi-dimensional array**

Definition:

Let a collection of items stored in a (e.g.) $n \times m$ grid (2D array),

$$[t_{1,1} \;\; t_{1,2} \ldots t_{1,m} \;\; ; \;\; t_{2,1} \;\; t_{2,2} \ldots t_{2,m} \;\; ; \;\; \ldots \;\; ; \;\; t_{n,1} \;\; t_{n,2} \;\; \ldots t_{n,m}]$$

Examples:

```
julia> [1 2 ; 3 4]
```

```
julia> T = ['a' 'b' 'c' ; 'x' 'y' 'z']
```

# Constructing a matrix (2/4)

## A matrix as an array of arrays

Definition:

Let a collection of items stored in (e.g.) a $n \times m$ grid (2D array),

```
[ [t1_1 t1_2...t1_m], [t2_1 t2_2...t2_m], [...], [t3_1 t3_2 ...t3_m] ]
```

Examples:

```
julia> [ [1 2], [3 4] ]
```

```
julia> A = [ [1 2], [3 4 6], [1] ]
```

# Constructing a matrix (2/4)

**A matrix as an array of arrays**

Definition:

Let a collection of items stored in (e.g.) a $n \times m$ grid (2D array),

```
[ [t1_1  t1_2...t1_m], [t2_1  t2_2...t2_m], [...], [t3_1  t3_2  ...t3_m] ]
```

Examples:

```
julia> [ [1 2], [3 4] ]
```

```
julia> A = [ [1 2], [3 4 6], [1] ]
```

# Constructing a matrix (2/4)

**A matrix as an array of arrays**

Definition:

Let a collection of items stored in (e.g.) a $n \times m$ grid (2D array),

$$[\ [t1_1\ \ t1_2 \ldots t1_m],\ [t2_1\ \ t2_2 \ldots t2_m],\ [\ldots],\ [t3_1\ \ t3_2\ \ \ldots t3_m]\ ]$$

Examples:

```
julia> [ [1 2], [3 4] ]
```

```
julia> A = [ [1 2], [3 4 6], [1] ]
```

# Constructing a matrix (3/4)

Typed but non initialized matrix:

```
Array{type}(undef,dim_1,dim_2,...,dim_n)
```

```
Matrix{type}(undef,dim_1,dim_2,...,dim_n)
```

Examples:

```
julia> m = Array{Int64}(undef,2,3)
```

```
julia> m = Matrix{Int64}(undef,2,3)
```

# Constructing a matrix (3/4)

Typed but non initialized matrix:

```
Array{type}(undef, dim_1, dim_2, ..., dim_n)
```

```
Matrix{type}(undef, dim_1, dim_2, ..., dim_n)
```

Examples:

```
julia> m = Array{Int64}(undef,2,3)
```

```
julia> m = Matrix{Int64}(undef,2,3)
```

# Constructing a matrix (4/4)

Typed and initialized matrix:

`zeros(`$type, dim_1, dim_2, \ldots, dim_n$`)`

`ones(`$type, dim_1, dim_2, \ldots, dim_n$`)`

See also `fill` function

Examples:

```
julia> m0 = zeros(Float64,2,3,4)
```

```
julia> m1 = ones(Float64,2,3,4)
```

```
julia> fill(π,2,3)
```

# Constructing a matrix (4/4)

Typed and initialized matrix:

```
zeros(type, dim_1, dim_2, ..., dim_n)
```

```
ones(type, dim_1, dim_2, ..., dim_n)
```

See also `fill` function

Examples:

```
julia> m0 = zeros(Float64,2,3,4)
```

```
julia> m1 = ones(Float64,2,3,4)
```

```
julia> fill(π,2,3)
```

# Accessing a matrix

General syntax for indexing into an n-dimensional array:

$$nameArray[index_1, index_2, \ldots, index_n]$$

```
julia> m[2,1]
```

General syntax for indexing an array of arrays:

$$nameArray[index_1][index_2] \ldots [index_n]$$

```
julia> A[2][3]
```

All indices within an entire dimension or across the entire array:

```
:
```

```
julia> T[:,2]
```

# Accessing a matrix

General syntax for indexing into an n-dimensional array:

$$nameArray[index_1, index_2, \ldots, index_n]$$

```
julia> m[2,1]
```

General syntax for indexing an array of arrays:

$$nameArray[index_1][index_2] \ldots [index_n]$$

```
julia> A[2][3]
```

All indices within an entire dimension or across the entire array:

```
:
```

```
julia> T[:,2]
```

# Accessing a matrix

General syntax for indexing into an n-dimensional array:

$$nameArray[index_1, index_2, \ldots, index_n]$$

```
julia> m[2,1]
```

General syntax for indexing an array of arrays:

$$nameArray[index_1][index_2] \ldots [index_n]$$

```
julia> A[2][3]
```

All indices within an entire dimension or across the entire array:

```
:
```

```
julia> T[:,2]
```

# Dimensions of an array

```
size(array, [dim])
```

```
julia> size(T)
julia> size(T,2)
```

```
length(array)
```

```
julia> length(T)
```

# Dimensions of an array

```
size(array, [dim])
```

```
julia> size(T)
julia> size(T,2)
```

```
length(array)
```

```
julia> length(T)
```

# Warning 1:

Observe this:

```
julia> [2, 3, 5, 7]
julia> [2 3 5 7]
```

Return an array with the same data as `A`, but with different dimension sizes or number of dimensions

```
reshape(array,dims)
```

```
julia> reshape([2 3 5 7], 4)
```

```
julia> reshape([2 3 5 7], (2,2))
```

# Warning 1:

Observe this:

```
julia> [2, 3, 5, 7]
julia> [2 3 5 7]
```

Return an array with the same data as `A`, but with different dimension sizes or number of dimensions

```
reshape(array, dims)
```

```
julia> reshape([2 3 5 7], 4)
```

```
julia> reshape([2 3 5 7], (2,2))
```

# Warning 2 (1/2)

Observe this:

```
julia> g = [2 3 4 5]
julia> h = g

julia> println(g)
julia> println(h)

julia> g[2] = 0

julia> println(h)
```

# Warning 2 (2/2)

```
copy(array)
```

```
julia> copy(T)
```

```
deepcopy(array)
```

```
julia> deepcopy(T)
```

```
similar(array)
```

```
julia> similar(T)
```

# Comprehension to construct arrays

[ fct($var_1$,$var_2$,...) for $var_1$=$val_1$, $var_2$=$val_1$,...]

```
julia> [i for i in 1:5]
```

```
julia> [i+j for i in 1:2 for j in 1:4]
```

```
julia> [i for i in 1:2, j in 1:4]
```

```
julia> [i for i in 1:10 if i % 2 == 1]
```

```
julia> [exp(i) for i in 1:3]
```

# Array and operations on a list (1/2)

```
julia> L=[1,2,3,4,5]
```

**Adding elements**

- at the end

```
julia> push!(L,10)
```

- at the front

```
pushfirst!(array,item)
```

```
julia> pushfirst!(L,0)
```

- (replacing) at the given index

```
splice!(array,position(s),item(s))
```

```
julia> splice!(L,3,[L[3] 7 4])
```

# Array and operations on a list (1/2)

```
julia> L=[1,2,3,4,5]
```

**Adding elements**

- at the end

```
julia> push!(L,10)
```

- at the front

```
pushfirst!(array,item)
```

```
julia> pushfirst!(L,0)
```

- (replacing) at the given index

```
splice!(array,position(s),item(s))
```

```
julia> splice!(L,3,[L[3] 7 4])
```

# Array and operations on a list (2/2)

```
julia> L=[1,2,3,4,5]
```

**Deleting elements**

- at the end

```
julia> pop!(L)
```

- at the front

```
popfirst!(array,item)
```

```
julia> popfirst!(L)
```

- at the given index

```
splice!(array,position(s))
```

```
julia> splice!(L,2)
```

# Array and operations on a list (2/2)

```
julia> L=[1,2,3,4,5]
```

**Deleting elements**

- at the end

```
julia> pop!(L)
```

- at the front

```
popfirst!(array,item)
```

```
julia> popfirst!(L)
```

- at the given index

```
splice!(array,position(s))
```

```
julia> splice!(L,2)
```

# Review and exercises

(notebook)