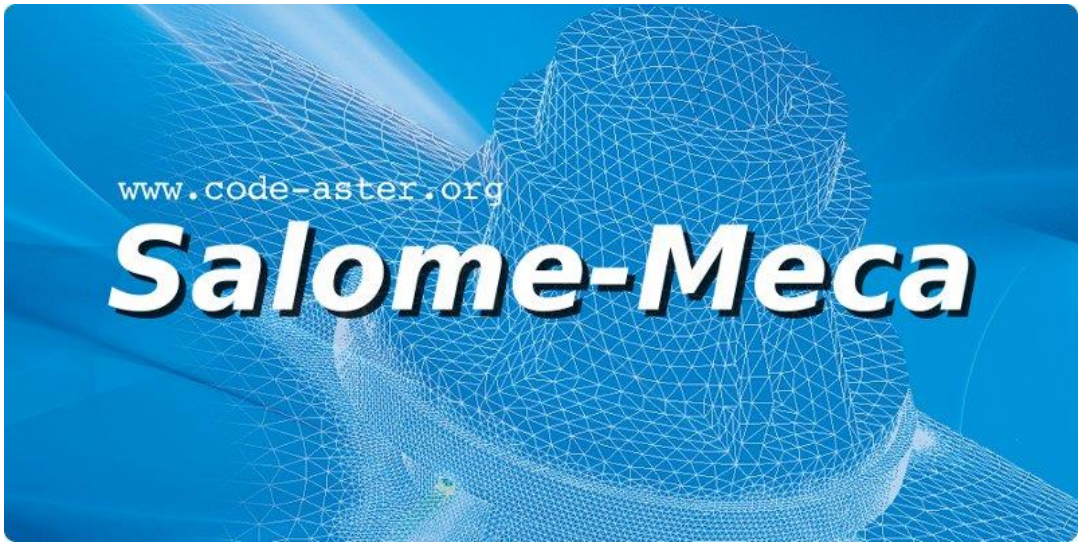
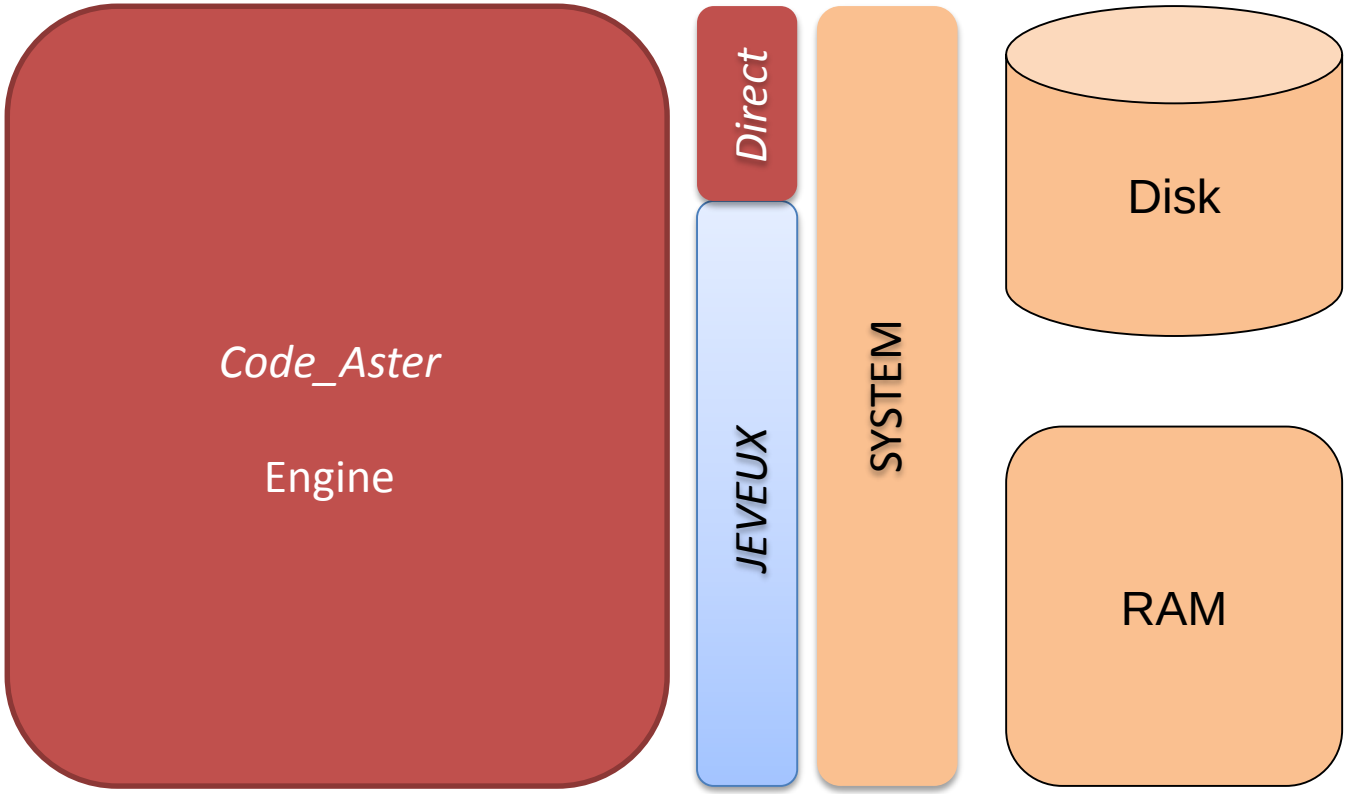
Development in code_aster
Memory management (JEVEUX)



Code_Aster, Salome-Meca course material

Memory management



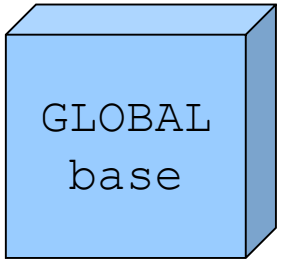
MAIN PRINCIPLES

Main principles (1)

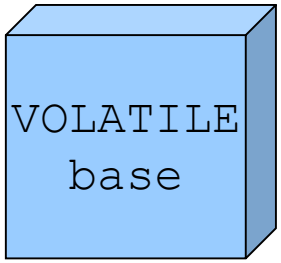
Lifetime of objects:

Questions: what is the lifetime of objects ? Are they saved somewhere ?

- Objects restricted to a single command reside on the **VOLATILE** basis.
- Objects that persist throughout the sequence of commands reside on the **GLOBAL** basis



All the **GLOBAL** objects are saved in output base (FIN)



All the **VOLATILE** objects are erased at end of command that created them

Main principles (2)

- Management with conventional FORTRAN 90
  - Static tables (predefined size)
  - Dynamic allocation: `allocate/deallocate`
- Management with JEVEUX memory utilities
  - Dynamic allocation
  - Structured objects management
  - Out-of-core mechanism (RAM/disk)
  - Native format for output bases (can been converted in HDF format)

Main principles (3)

- When use memory mechanisms ?
  - Static size: only for small objects
  - `allocate/deallocate`: for large objects but no saving (VOLATILE objects)
    - For non-structured objects (real, integers, …): `AS_ALLOCATE/AS_DEALLOCATE`
    - For structured objects (derived data type): `allocate/deallocate`
  - JEVEUX memory management:
    - For structured objects and large objects with save (GLOBAL objects)

DYNAMIC ALLOCATION

Dynamic allocation (1)

- Is allowed by Fortran90 standard
  - `allocate/deallocate`
  - May only be used for local object (equivalent to `VOLATILE` basis)
- Wrappers are available in Code_Aster, use them whenever possible ! Syntax is slightly different:
  - `AS_ALLOCATE(vi=ptr, size=N)`
    instead of
  - `ALLOCATE(ptr(N))`

  - `AS_DEALLOCATE(vi=ptr)`
    instead of
  - `DEALLOCATE(ptr)`
- These wrappers rely on JEVEUX : they are useful to keep the memory evaluations precise (for the out-of-core mechanism) and to avoid memory leaks

Dynamic allocation (2)

- Allocate/deallocate: should be used only when as_allocate is not relevant :
  - allocation of a 2D array
  - of a vector of derived type ....
- Don’t forget to deallocate, otherwise **you create a memory leak!**

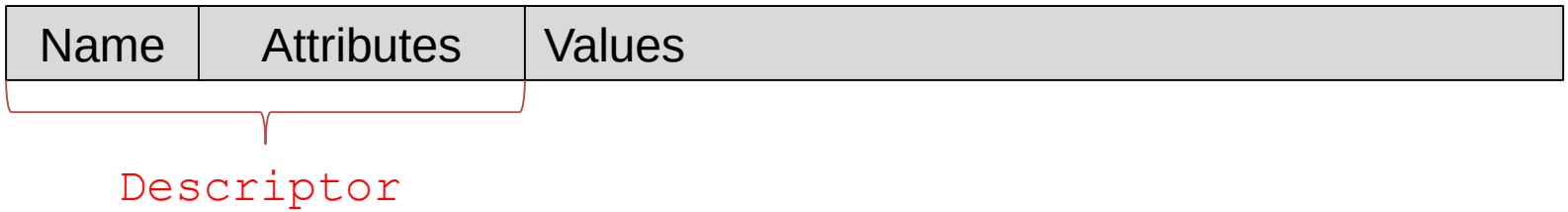
THE JEVEUX UTILITY

JEVEUX , Code_Aster native memory management utility

- Is a high level utility for memory management
- Allows the creation, dynamic allocation and destruction of *objects* in a (at least originally) Fortran77 context
- Provides the building blocks (= *objects*) for Code_Aster datastructures
- Offers two abstract memory zones : *GLOBAL/VOLATILE*.
  - *Global objects* persist throughout the sequence of commands
  - *Volatile objects* are cleared at the end of each command
- Is a memory scheduler : manages the use of RAM/disk ressources. *Objects* may reside in RAM or disk, depending on the available ressources.

The JEVEUX object

- A JEVEUX object is made of :
  - A descriptor : Name + Attributes
  - A vector of values



- A JEVEUX object is referred to by its name

JEVEUX object : the name

- The JEVEUX object name:

- Maximum length: 24
- Use only alphanumeric characters : A-Z,0-9," ","_",".","&"
- Naming conventions depend on the GLOBAL/VOLATILE basis choice
  - GLOBAL objects result from a command. Their names shall begin with the name of this result (defined in the command file)

```
STEEL = DEFI_MATERIAU(ELAS=_F(E=205000.E6,NU=0.3 ))
```

Name of JEVEUX objects begins by obj_name = 'STEEL '

  - VOLATILE objects are temporary objects created within a command (and destroyed at the end of this command). Usually prefixed with && but this is not mandatory

JEVEUX object: the attributes (1)

Name	Attributes	Values

Attribute name	Description	Values	Access status
CLAS	Basis: volatile/global	(V|G)	creation
GENR	Kind of object: Element, vector, names directory	(E|V|N)	creation
LONMAX	Length of the vector of values	Integer	creation
TYPE	Type of values	(IS|I|R|C|L|K8|K16|K24|K32|K80)	creation
LONUTI	Number of used components of an object of type V	Integer	Any time
IADM, IADD, USAGE	Internal use		Never
And many more attributes : see D6.02.01 for an exhaustive description			

JEVEUX object: the attributes (2)

- The JEVEUX type:

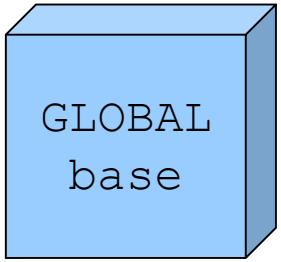
Value Type	COMMON	JEVEUX	FORTRAN
Short integer	ZI4	IS	INTEGER(kind=4)
Integer	ZI	I	INTEGER(kind=8)
Real	ZR	R	REAL(kind=8)
Complex	ZC	C	COMPLEX(kind=8)
Boolean	ZL	L	LOGICAL(kind=1)
String (length=8)	ZK8	K8	CHARACTER(len=8)
String (length=16)	ZK16	K16	CHARACTER(len=16)
String (length=24)	ZK24	K24	CHARACTER(len=24)
String (length=32)	ZK32	K32	CHARACTER(len=32)
String (length=80)	ZK80	K80	CHARACTER(len=80)

JEVEUX attribute: the base

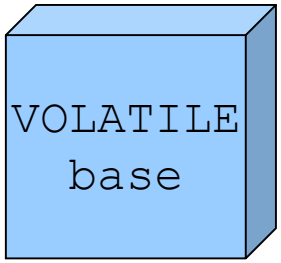
The JEVEUX object basis:

Questions: what is the lifetime of JEVEUX objects ? Are they saved somewhere ?

- Objects restricted to a single command reside on the VOLATILE basis.
- Objects that persist throughout the sequence of commands reside on the GLOBAL basis

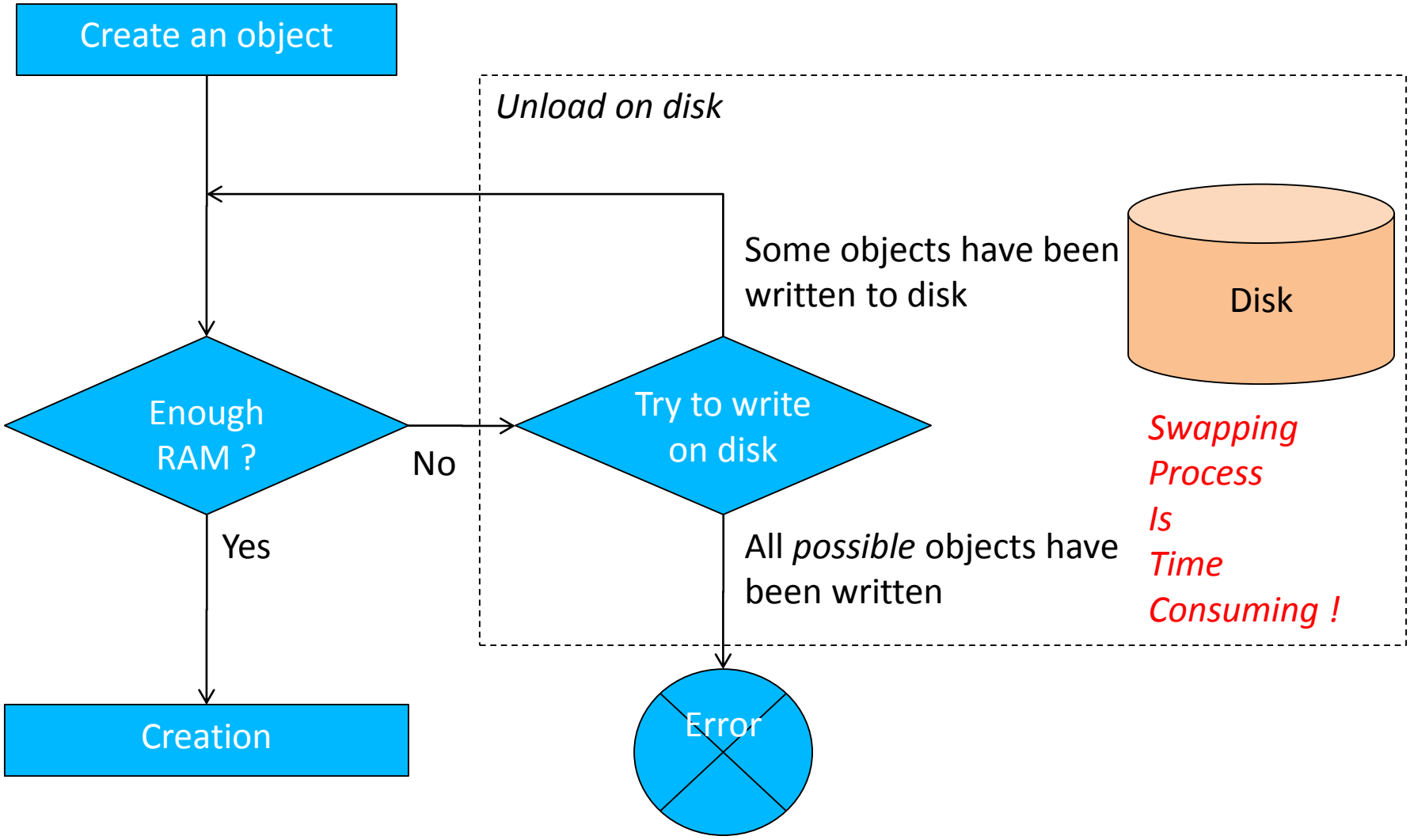


All the GLOBAL objects are saved in output base (FIN)



All the VOLATILE objects are erased at end of command that created them

Create an object (1)



Create an object (2)

Create in three steps:

- Create descriptor

```
call jecreo(obj_name,type)
```

- Set attributes

```
call jeecra(obj_name,'LONMAX',ival = size)
call jeecra(obj_name,'LONUTI',ival = size)
```

- Create values

```
call jeveuo(obj_name,'E',addr)
```

Create an object (3)

- Name of JEVEUX object: `obj_name`
- Type of object: `type = 'V V itype'`

  First: JEVEUX base (V or G)

  Second: kind V for vector and N for names' repository (see collection)

  Third: type of values `itype`
- Size of object: `size`
- Address of object: `addr`
- JEVEUX objects are **always initialized** (`ZERO,' ', .FALSE.`) at creation

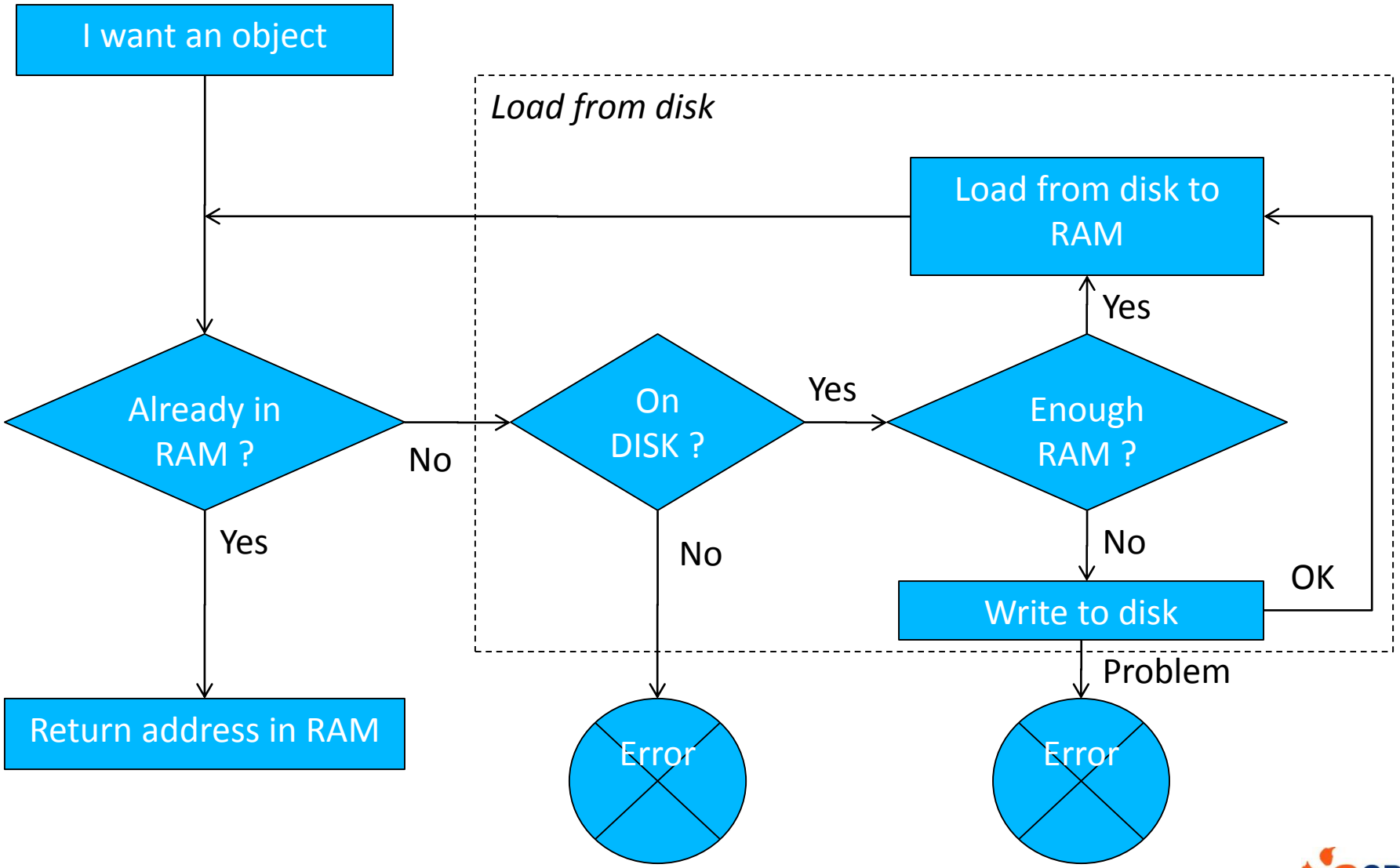
Create an object (4)

```
implicit none
#include 'asterfort/jecreo.h'
#include 'asterfort/jeecra.h'
#include 'asterfort/jeveuo.h'

integer :: adrs, size
character(len=24) :: obj_name

obj_name = '&&TOTO.EXAMPLE'
size = 5
call jecreo(obj_name, 'V V K16')
call jeecra(obj_name, 'LONMAX', size)
call jeecra(obj_name, 'LONUTI', size)
call jeveuo(obj_name, 'E', adrs)
```

All in one !

```
#include 'asterfort/wkvect.h'
call wkvect(obj_name, 'V V K16', size, adrs)
```

Access to an object (1)



Access to an object (2)

Get address of JEVEUX object:

```
call jeveuo(obj_name, access, adrs)
```

Name of JEVEUX object: `obj_name`

Type of access: `access= 'L' or 'E'`

Address of object: `adrs`

Get pointer to JEVEUX object:

```
call jeveuo(obj_name, access, v{type}=ptr)
```

Pointer to object: `ptr`

`v{type}` is one of `vi, vr, vk8…`

Access to an object (3)

Access by address: you need access to JEVEUX common

```
#include 'jeveux.h'
call jeveuo(obj_name, 'E', adrs)
zr(adrs-1+3) = 0.d0
```

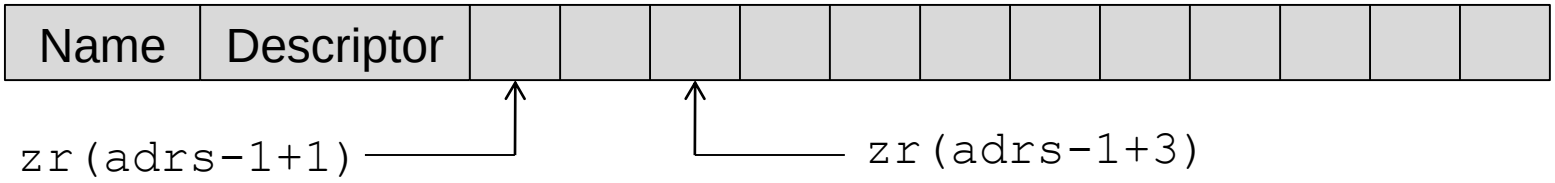
You must know the type to use correct common object:

`zr, zi, zk8, zk16, zk32,` etc...

Value Type	IS	I	R	C	L	K8	K16	K24	K32	K80
Name of the array	zi4	zi	zr	zc	zl	zk8	zk16	zk24	Zk32	zk80

You must know the size of the vector (and the index of the entry you want to access)

You must respect access: read or read/write



Access to an object (4)

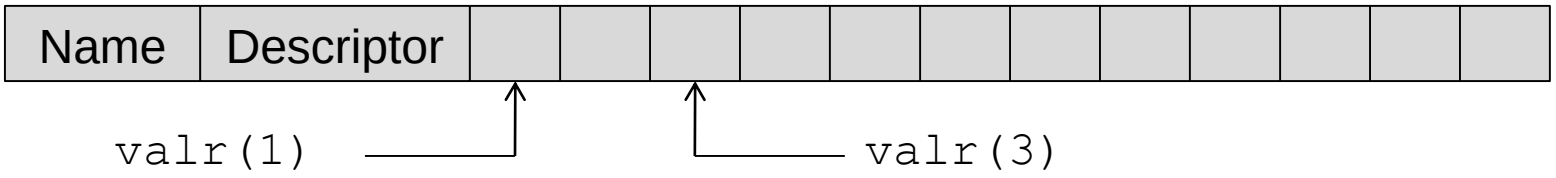
Access by pointer (F90)

```
real(kind=8), pointer :: valr(:)=>null()
call jeveuo(obj_name, 'E', vr=valr)
valr(3) = 0.d0
```

You must know the type to use correct common pointer

You must know the size to access correct object in vector

You must respect access: read or read/write



Access to an object (5)

How long does the address of an object remains valid ?

Until JEVEUX writes the object to the disk

Why does JEVEUX write an object to the disk ? To get RAM !

Out-of-core mechanism

How does the JEVEUX select objects to write on disk ?

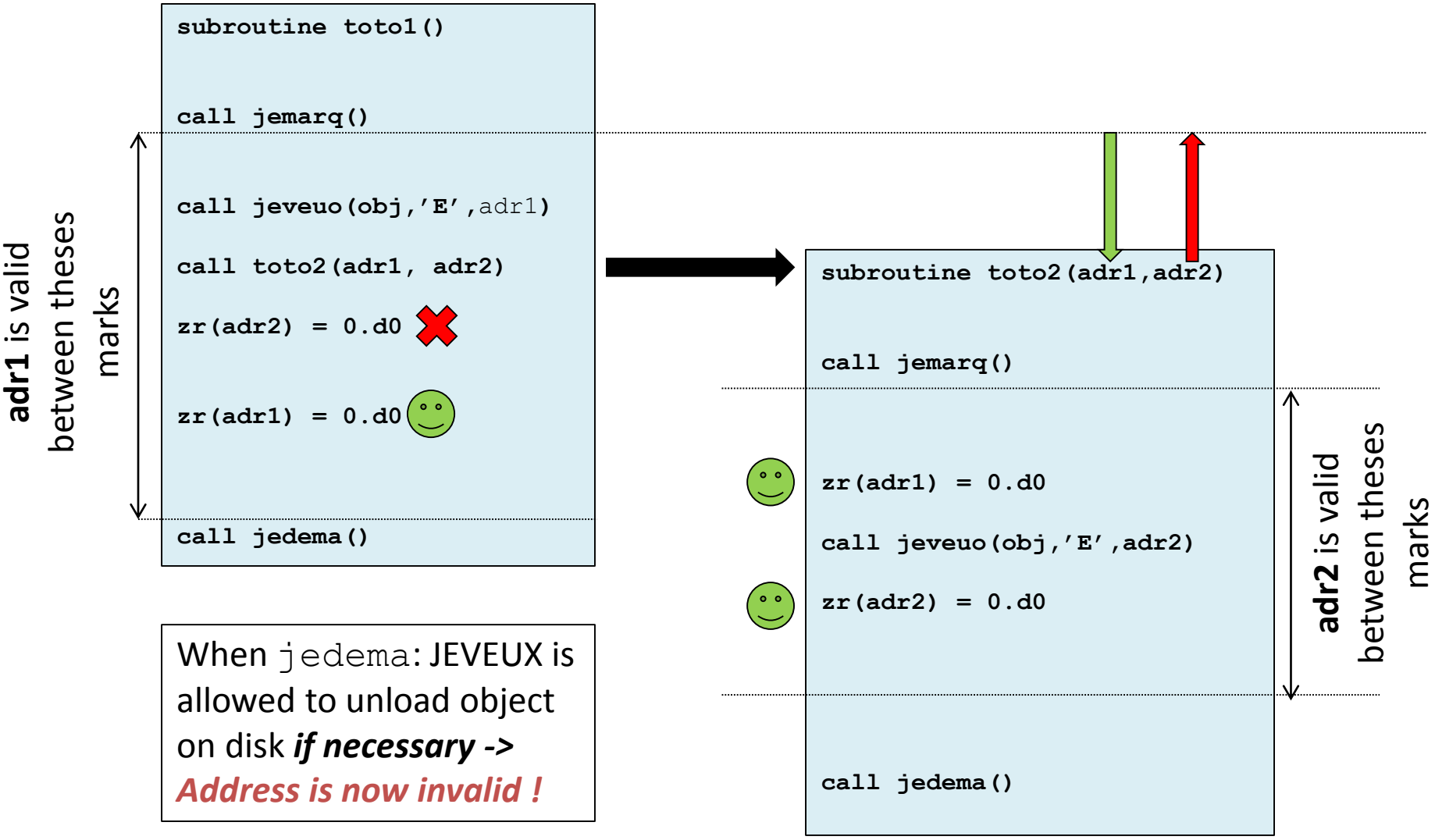
- The developer explicitly indicates that the object may be written on the disk, if necessary.

  `call jelibe(obj_name)`
- Based on a system of “marks”. On entry of a routine,
  - `call jemarq` : the current mark is increased (+1)
  - `call jedema` : object with the current mark may be written on disk, then the current mark is decreased (-1) .

When the JEVEUX objects are wrote on disk ? ?

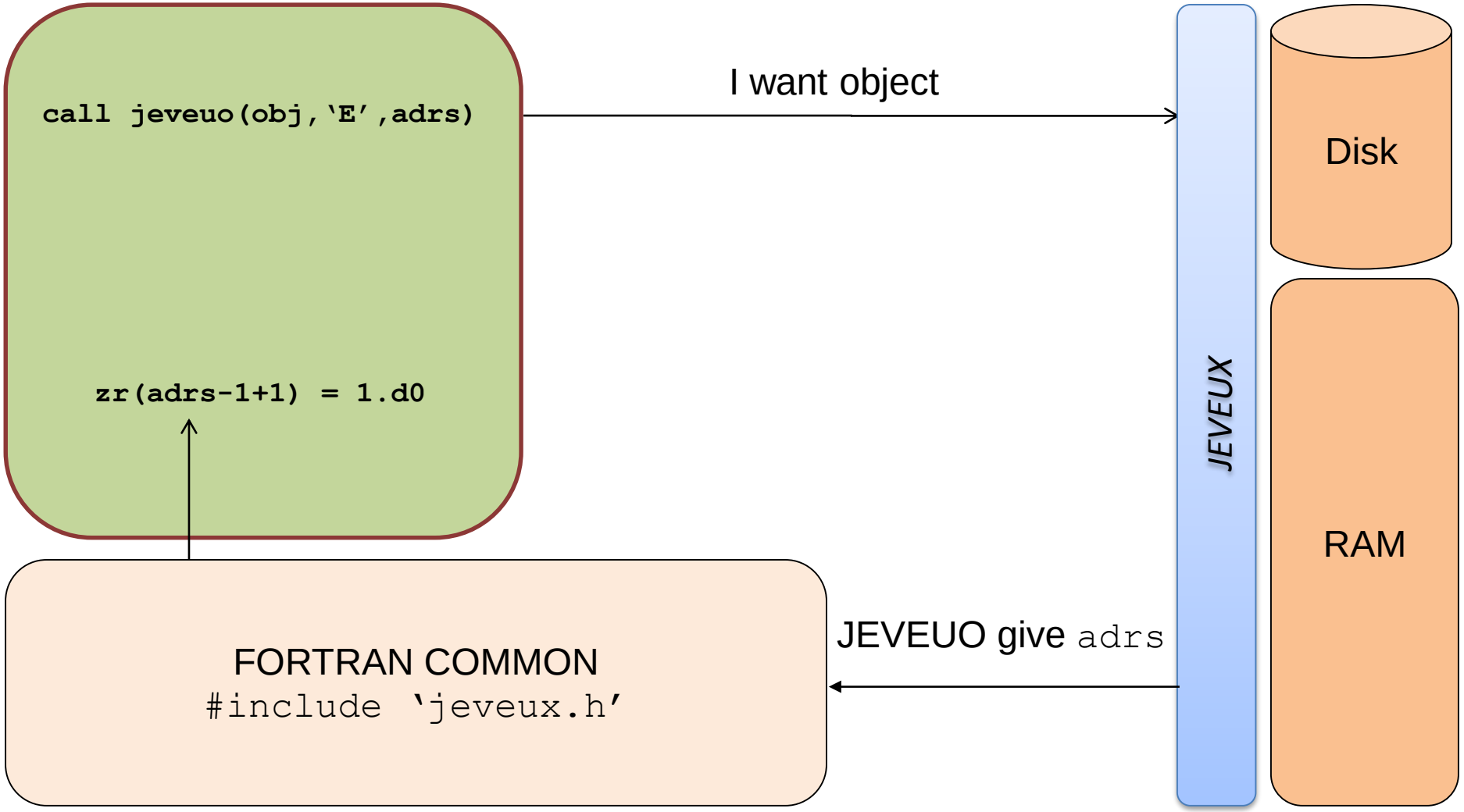
- If explicitly requested (rare, for a very large object) `call jjldyn(obj_name)`
- When RAM is needed

Access to an object (6)



When `jedema`: JEVEUX is allowed to unload object on disk ***if necessary ->***
Address is now invalid !

Access to an object (7)



Other operations (1)

Destruction of an object

```
call jedetr(obj_name)
```

- Don’t forget: all VOLATILE objects are destroyed at the end of current command

Read an attribute

```
call jelira(obj_name,attr,ival,kval)
```

- `ival` for integer type attribute
- `kval` for character type attribute

Increase size of an object

```
call juveca(obj_name,new_length)
```

- Careful: this operation COPY old data in new object: the address of object before juveca is not the same after `juveca`. You must call again `jeveuo`.

Other operations (2)

Duplication of an object

```
call jedupo(obj_name,new_obj,' ',flag_coll)
```

- The `new_obj` is destroyed if exists before
- `flag_coll` is for collection (see documentation)
- Could be very expensive if big object !

Existence of an object

```
call jeexin(obj_name,iexist)
```

- `iexist=0` object doesn't exist
- `iexist=1` object exists

Commands for debug & print: see documentation

THE JEVEUX UTILITY COLLECTIONS

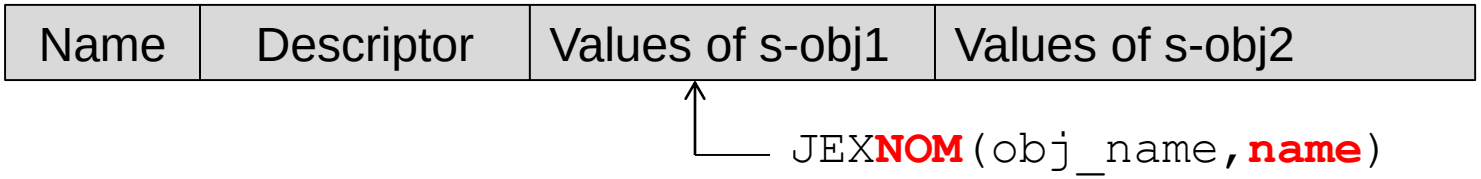
Collections – General definition

A **collection** is a list of several JEVEUX objects (sub-object):

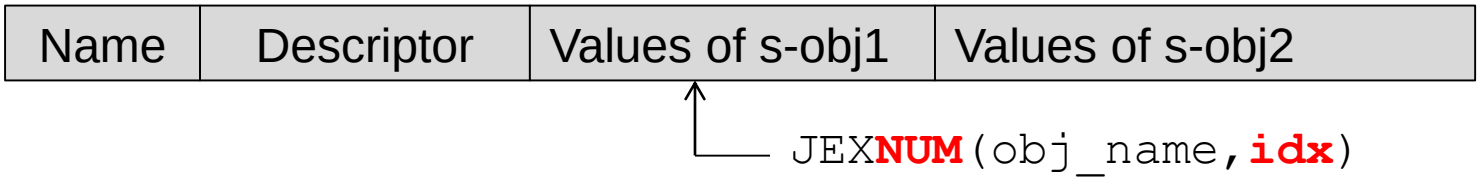
- All sub-objects are the same type
- Each sub-object in collection could have different lengths
- Access to an sub-object by name or by number

Collections – How to access to sub-object (1)

Access to sub-object with a string:



Access to sub-object with a number:



Collections – How to access to sub-object (2)

Access to sub-objects by a string need names' repository JEVEUX object

```
call jecreo(rep_name, type)
```

Type of object: `type = 'V N K8'`

- First: JEVEUX base (`V` or `G`)
- Second: kind `N` for names' repository
- Third: type of values `itype='K*'`

```
call jeecra(rep_name, 'NOMMAX', nbn_maxi)
```

Maximum number of names in repository: `nbn_maxi`

Collections – How to access to sub-object (3)

- Create names in names' repository JEVEUX object

```
call jecroc(jexnom(rep_name, name))
```

Name in names' repository: name

- Get number of names `nbn_used` already defined in names' repository JEVEUX object

```
call jelira(rep_name,'NUTIOC',ival=nbn_used)
```

Collections – How to access to sub-object (4)

What is the index of specified name in names' repository ?

```
call jenonu(jexnom(rep_name,name),idx)
```

Index of name in names' repository: `idx`

What is the name of specified index in names' repository ?

```
call jenuno(jexnum(rep_name,idx),name)
```

Name at index in names' repository: `name`

Collections – Create (1)

- Collection: define name of collection

```
call jecrec(obj_name,type,acs,store,tlen,size)
```

  - Name of JEVEUX object (collection): **obj_name** (same rules from JEVEUX object's naming)

Collections – Create (2)

- Collection: define type of sub-objects

```
call jecrec(obj_name,type,acs,store,tlen,size)
```

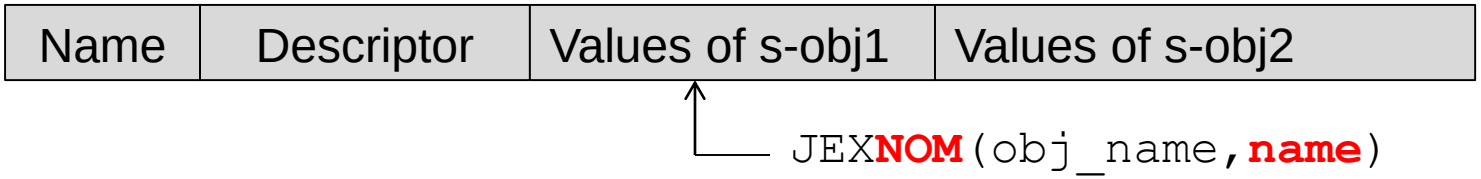
- Type of sub-objects: **type** = 'V V itype'
  - First: JEVEUX base (`V` or `G`)
  - Second: kind `V` for vector for each sub-object
  - Third: type of sub-object values `itype`

Collections – Create (3)

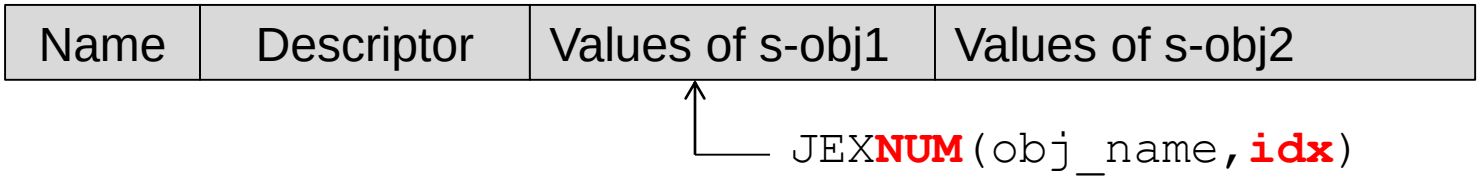
- Collection: define access (by string or number)

```
call jecrec(obj_name,type,acs,store,tlen,size)
```

  - Type of access of sub-objects: **acs='NO'** for string access



  - Type of access of sub-objects : **acs='NU'** for number access

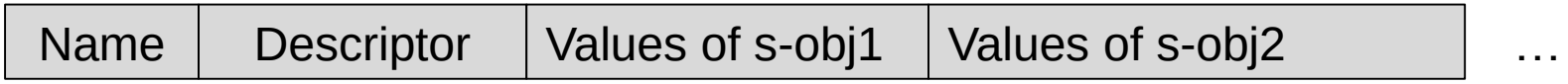


Collections – Create (4)

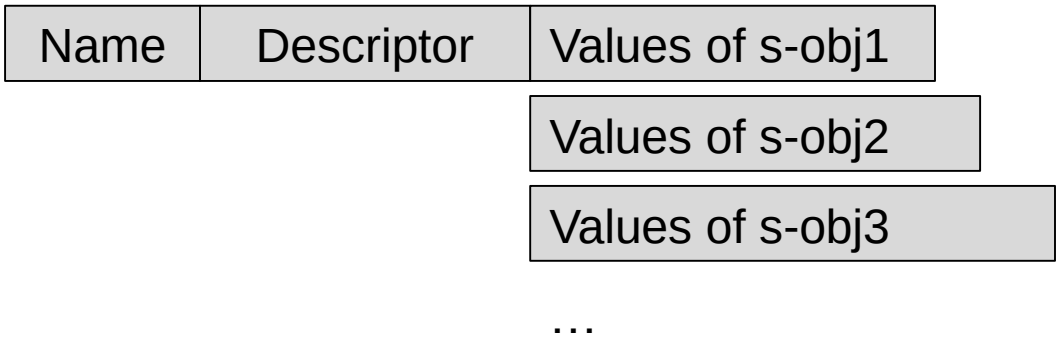
- Collection: define storing

```
call jecrec(obj_name,type,acs,store,tlen,size)
```

- Type of storing of sub-objects: **store='CONTIGU'**



- Type of storing of sub-objects : **store='DISPERSE'**



Choice ?

CONTIGU is more CPU efficient in access but with more consumption of memory

Collections – Create (5)

- Collection: define length of sub-objects

```
call jecrec(obj_name,type,acs,store,tlen,size)
```

- Each sub-object **is not** the same length: **tlen='VARIABLE'**

Name	Descriptor	s-obj1	s-obj2	s-obj3

- Each sub-object **is** the same length : **tlen='CONSTANT'**

Name	Descriptor	s-obj1	s-obj2	s-obj3	s-obj4

Collections – Create (6)

- Collection: define number of sub-objects

```
call jecrec(obj_name,type,acs,store,tlen,size)
```

  - Number of sub-objects in collection

Create a collection (1): access by string

Automatic generation of names' repository:

```
implicit none
#include 'asterfort/jecrec.h'
#include 'asterfort/jexnom.h'
#include 'asterfort/jecroc.h'

integer :: size
character(len=24) :: obj_name

obj_name = '&&TOTO.EXAMPLE'
size = 3
call jecrec(obj_name, 'V V I', 'NO', 'DISPERSE', 'VARIABLE', size)
call jecroc(jexnom(obj_name,'NAME_1'))
call jecroc(jexnom(obj_name,'NAME_2'))
call jecroc(jexnom(obj_name,'NAME_3'))
```

Create a collection (2): access by string

Using previous names' repository:

```
implicit none
#include …

integer :: size
character(len=24) :: obj_name, rep_name

obj_name = '&&TOTO.EXAMPLE'
rep_name = '&&TOTO.NAMES'
size = 3
call jecreo(rep_name, 'V N K8')
call jeecra(rep_name, 'NOMMAX', ival=size)
call jecroc(jexnom(rep_name,'NAME_1'))
call jecroc(jexnom(rep_name,'NAME_2'))
call jecroc(jexnom(rep_name,'NAME_3'))
call jecrec(obj_name, 'V V I', 'NO '//rep_name, 'DISPERSE','VARIABLE',size)
```

Create a collection (3): total size

For CONTIGU storage:

```
implicit none
#include …

integer :: size, size_s(3), size_t
character(len=24) :: obj_name, rep_name

obj_name = '&&TOTO.EXAMPLE'
rep_name = '&&TOTO.NAMES'
size = 3
size_s(1) = 10
size_s(2) = 15
size_s(3) = 7
size_t = size_s(1) + size_s(2) + size_s(3)
call jecrec(obj_name, 'V V I', 'NO', 'CONTIGU', 'VARIABLE', size)
call jeecra(obj_noma, 'LONT', size_t)
```

Create a collection (4): total size

For DISPERSE storage:

```
implicit none
#include …

integer :: size, size_s(3)
character(len=24) :: obj_name, rep_name

obj_name = '&&TOTO.EXAMPLE'
rep_name = '&&TOTO.NAMES'
size = 3
size_s(1) = 10
size_s(2) = 15
size_s(3) = 7
call jecrec(obj_name, 'V V I', 'NO', 'DISPERSE', 'VARIABLE', size)
call jeecra(jexnom(obj_name,'NAME_1'), 'LONMAX', size_s(1))
call jeecra(jexnom(obj_name,'NAME_2'), 'LONMAX', size_s(2))
call jeecra(jexnom(obj_name,'NAME_3'), 'LONMAX', size_s(3))
```

Structuration with JEVEUX (1)

JEVEUX allows to mimic derived type (sd aster – doc D4):

Example:

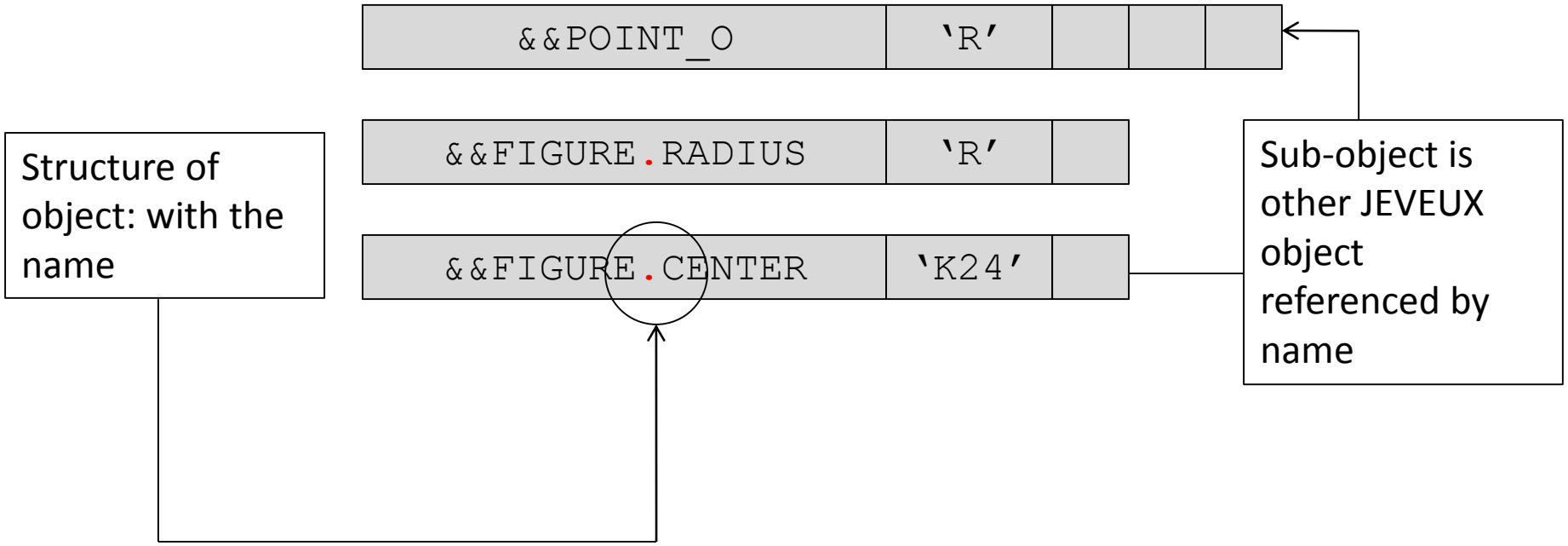
- `point` defines the coordinates of a point
- A circle is defined by a `point` (sub-object) and a radius

```
type point
   real(kind=8), dimension(3) :: coord
end type


type circle
   point         :: center
   real(kind=8)  :: radius
end type
```

Structuration with JEVEUX (2)

JEVEUX allows structuration of datas:

Each object is identified by its name: we can use different types



SUMMARY

About…

- Documentation for JEVEUX: D6.02.01 and D2.06.01
- Don’t create JEVEUX objects in a low-level subroutine (under `CALCUL` or in behaviour laws)
- Prefer `AS_ALLOCATE` for temporary objects
- JEVEUX objects on `GLOBAL` base must be documented in D4 documents and in datastructure catalogs (./bibpyt/SD)

End of presentation

Is something missing or unclear in this document?

Or feeling happy to have read such a clear tutorial?

Please, we welcome any feedbacks about Code_Aster training materials.

Do not hesitate to share with us your comments on the Code_Aster forum dedicated thread.