#0
Element Syntax
by Stephan Kammel
date and time:
+ 2023|06|19 - 2023|06|20 #0~(0)~(1)~(1.1)~(1.2)~0#
location: @home

ES concept(0):
- most needed syntax is to be replaced with symbols and signs
- oop is achieved via standardized element objects
- instances can be added to or removed from an object instance to free up ram
- ram usage calculation should be based on element instance count within ES
- rank and dynasty based approach for multiple inheritance of code sources
- target platforms: existing os infrastructure

Element definition(1):
- an element is constructed from es keys, keywords and inbuilt functions
- each element has the exact same design
- each element is part of a source code dynasty and has a rank
- rank and dynasty interactions of elements can be modified via modifiers
- each element has a name, default "element"
- each element has a unique id, incrementally descending per program rank to lesser ranks within the dynasty, not setable, but accessible by the programmer, dependend on time of creation of the element instance, an incremented integer value will be returned, the dynasty root instance must be informed on dynastic activities and must assign the id to an requesting element, as well as store or update stored information on dynasty exchange, elements need to be stored in values list of root element, as well as the value list reference and the id where the objects at in the index value dictionary of the requesting element
- each element carries 3 internal integer values, smallest datatype possible as long as unset, later it must be resized and inform ES about the size change, in case f.e. an integer expands beyound 32k, the next biggest data type must be used, and ES must be informed or already knows it, that more space is needed in storage, it should request the needed space from the os accordingly or it needs to manage it for the element, f.e. assigning a second basetype and fill it until it is full and so on, default for the three values is 0, they are for potential future stuff, where spatial storage or 3 dimensional orientation or text might be a thing or the element would represent f.e. an 3d object like a cube, representing pixels or color values or whatever triple stuff
- each element carries an empty string, default ""
- each element carries an empty list, default empty, for values like names, data, etc.
- each element carries an index value dictionary, default empty, whenever an element is added to a list, a reference to the list and the indexvalue are stored in the dictionary, as well on every change, so that each element knows everytime where it is stored at,

neglect storage if assigned object and place are not transmittable by the caller
- each element with the appropriate modifiers can traverse ranks and dynasties and be attributed
to other elements as an instance attribute, f.e. a class with the modifier ?!last & ?! fame
could start in program level and be passed as a parameter to end up in another rank and dynasty,
but other elements could never inherit from it, with the modifier ?!theone the code could not be
executed within the namespace realm of ES if ?!theone has one instance already active in the ES
internal namespace instance element tree
- ES &| es allow for configuration of maximum element storage size, above an element buffer error
must be raised and the running application terminated or at least the executed code must be halted
to contain stack overflows within a calculatable amount
- there are no base types, base types are !?silent modified elementa accessible only from es rank (-1)

- each element is built as follows:
-<
0. rank (es)(element){ < ES.RANKNAME }
1. identifier (ES)(element){ < ES.IdRequest} <- IDF, once received, storage creation is initialized
2. element storage(<- not part of the declaration, created upon element definition
2.0 name(string){"element"} <- string
2.1 x (smallint){0} <- char/smallint
2.2 y (smallint){0} <- char/smallint
2.3 z (smallint){0} <- char/smallint
2.4 text (string){""} <- string
2.5 values (list){[]} <- empty list element
2.6 indices (list){[=]} <- empty dictionary element
2.7 optional additions <- one instance must already exist, via ES or special builtin function
3. dynastic elements (element){n args} <- DYE
4. traded elements (element){n args} <- TRE
5. constructor elements (element){n args} <- COE
6. lesser rank elements of element <- LREOE
7. modifiers (element) <- MOD
8. adress (ES)(bytearray){ < ES.AdressRequest} <- only visible to negative ranks
9. rank
10. optional identifier again
>-

Element Ranks(1.1):

-<

-2. ES virtual machine, dont know yet, the program of ES which monitors compiler activity

-1. es the compiler, can only be executed by ES

0. member .. lowest rank

1. element .

2. method ;

3. class :

4. program ! if highest rank it is the dynastic root of the element instance

(5. server ??)

(6. system ??)

(7. infrastructure ??)

>-

- those are the ranks, in the future they could be expanded upwards to represent greater abstraction levels and higher combined complexity of all subsequent elements not yet known as a concept by name

- each element with a positive rank is written as follows in a source code file:

-<

0. member
..IDF(DYE){TRE} ..>COE<.. LREOE ?!MOD..IDF

1. element
.IDF(DYE){TRE} .>COE<. LREOE ?!MOD.IDF

2. method
;IDF(DYE){TRE} ;>COE<; LREOE ?!MOD;IDF

3. class
:IDF(DYE){TRE} :>COE<: LREOE ?!MOD:IDF

4. program
!IDF(DYE){TRE} !>COE<! LREOE ?!MOD!IDF

(5. server
µIDF(DYE){TRE} µ>COE<µ LREOE ?!MODµIDF)

(6. system
%IDF(DYE){TRE} %>COE<% LREOE ?!MOD%IDF)

(7. infrastructure
&IDF(DYE){TRE} &>COE<& LREOE ?!MOD&IDF)

(8. realm of technology
°IDF(DYE){TRE} °>COE<° LREOE ?!MOD°IDF)

(8. knowledge nexus / totality / ...
=IDF(DYE){TRE} =>COE<= LREOE ?!MOD=IDF)

>-

- some examples:

```
!helloWorld
    .(Console){"Hello World"}. print{.text}.
?!executable
!


!Calculator
?BasicCalculations
```

```
??Console

!> operand_left(int), operand_right(int) <!

    :Square(<*)

        ;get_area
            > operand_left * operand_right

            .get_diagonal
                > :;calculate_diagonal
            ?!
            .
        ?!
        ;

        ;calculate_diagonal(<*)
            > ^{2~operand_left} + ^{2,expoperand_right}
        ?!loyal
        ;


    ?!loyal
    :

    ;run(+Console){ a(int) b(int) }
        # print{@"d = {!:Square.get_diagonal}\n"+
            @"a={!:Square;get_area}"
        }
    ;


?!callable
!
```

- at first ES will only support negative ranks and ranks up to rank 4

Element modifiers(1.2):

<- hierarchy visibility

<- ?!unknown <- invisible to higher ranks within ES, access managed by ES or es <- lower ranks must be of the same rank descendant within ES to see

<- ?!fame <- visible to all ranks within ES regardless of family

<- ?!known <- visible to higher ranks of the same family within ES

<- dynastic behaviour

<- ?!only <- only one active element within the entirety of ES

<- ?!theone <- only one active element within an ES namespace

<- ?!neglect <- won't inherit dynastic elements unless told so

<- ?!last <- other elements can not inherit from

<- basic statements
<- ?! <- higher rank reigns

<- ?!delegate <- lower rank reigns

<- ?!loyal <- won't answer calls unless from higher ranked dynastic element

<- ?!silent <- won't answer calls unless from es

<- ?!executable <- can be run by ES

<- ?!callable <- other elements can call from outside familiy
<- and from all ranks, depending on higher ranks modifiers

<- ?!ES <- can only access rank -1, only element priviliged to see executable elements
0#