|  |
| --- |
| Circle Language Spec: Parameters |

## Imported Parameter Concepts

The following articles will explain some concepts from other programming languages and the how they are implemented in the new computer language.

- *Required & Optional*

- *Variable Amount of Arguments*

- *Return Values*

- *Parameter Order*

### Required & Optional

#### Concept

Some programming languages have a thing called required and optional parameters. In other programming languages basically all parameters are required, unless they are typed optional. In the new computer language it is the other way around: any parameter is optional unless it is typed required. This is because in the new computer language commands are the same as objects, that just happen to be executable, and the parameters inside a command are analogus to sub-objects inside another object, whose filling-in is also totally arbitrary.

Some parameters can be made required, so you *have to* fill them in as input. In the new computer language, this means that reading and writing any accessible aspect of a parameter can be separately made required or optional. When output is required, it means you have to use the output. You will be unable to execute the command without picking up the return value or returned object.

Basically each system command can be made required or optional. Usually you make access from the *outside* required. When you make access from the *inside* required, this can give the user of the command guarantees about the parameter’s usage.

#### Required & Optional in a Diagram

*Optional* is the default behavior. *Required* is an extra rule imposed. Required is expressed by drawing half a shape at the end of a connector. It should be half of what has to be connected to it. Most of the times it will be half a circle:



For command objects it is half a square:



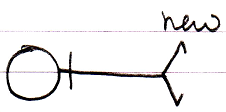
And when a command needs to be executed it is about 5/8th of a diamond:



It is not half a diamond because that already symbolizes the Protected access modifier, which is half a cross:



For system commands that require the command call connector notation, also the 5/8th of a diamond is shown when a call is required:



The ‘half a shape’ will also get a line type adapted to the role the required object will get. If the connector specifies that it gets a class role, the half-shape will be drawn with a dashed line.



Below you will find examples of *required* connectors. Not every possible connector is shown, but enough to cover the notation.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Public Required Object Set | Protected Required Object Set | Public Required Object Get |
|  |  |  |
|  |  |  |
| Private Required Object Set | Friend Required Object Set | Private Required Object Get |
|  |  |  |
|  |  |  |
| Protected Required  Object Get | Friend Required Object Get | Public Required Class Set |
|  |  |  |
|  |  |  |
| Public Required Use As Class | Private Required Class Set | Friend Required Value Set |
|  |  |  |
|  |  |  |
| Public Required Clone (2) Get | Public Required  New | *(as opposed to)*  Public New |
|  |  |  |
|  |  |  |
| Friend Required Execute | Public Required Object Get | Public Required Use As Class |
|  |  |  |
|  |  |  |
| Public Required Use As Class  (might be a better notation, since it is more likely to be used for plain command calls) |  |  |

#### Ideas about Required and Optional

(Out of the original Symbol documentation)

* Just as in procedures, you could make certain members of a type required, while others are optional. You have to fill in the required members on creation of the object. This should be there for the same reason as required parameters are there in a procedure: the function of the object just doesn’t make sense unless you write the required members. The programmer is made extra aware of that by making the members required.

### Variable Amount of Arguments

#### Concept

Other programming languages have a concept called variable amount of arguments. This means, that a command’s parameter list does not have a fixed amount of parameters. The last parameters can be any amount of parameters. This is specified as the last parameter’s being an *array* of parameters.

In the new computer language, any parameter can be an array. In other programming languages, parameters could also be arrays, but for the array, that represented the variable amount of arguments, each item of the array was separately listed out at the end of the command call, as opposed to other parameters, that were arrays, which were just references to arrays, that were initialized elsewhere, not inside the command call.

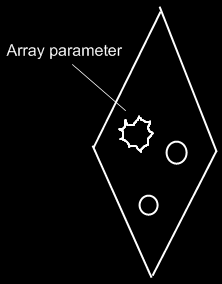
In the new computer language, for backwards compatibility purposes, and textual representation purposes, a parameter may be typed as the array that will represent the variable amount of arguments. In other programming languages, variable amount of arguments was also just for textual representation.

Perhaps in the future, it can be applied, that multiple array parameters are considered the variable amount of arguments. But how the notation will be kept unambiguous then is not clear yet.

Important to remember, is that parameters are not always single items, but a parameter can also be an array.

#### Variable amount of Parameters in a Diagram

The array parameter, that represents the variable amount of arguments, does not have a special display in the diagram (yet). What is important to remember about the concept of variable amount of arguments is that: parameters can be arrays.



#### Ideas about Variable Amount of Parameters

(Out of the original Symbol documentation)

In text code, parameter assignments before a call are notated something like this:

Procedure A ( 0 , 4 )

Procedure A is the procedure name, the two numbers between brackets are the two arguments. Text code is covered in detail lateron.

The parameters of a procedure are listed in a fixed order. Even the optional parameters have a place in this fixed list and when you don’t want to fill it in you must leave the position open:

Procedure A ( 0 , 4 , , 1 )

After the fixed parameter list, a procedure can allow a variable amount of parameters to follow. This variable amount of parameters is stored in a single special parameter. This parameter is of type Array, a type defined in the Sigma Data module. To support variable amount of arguments, the procedure must have a public Array object that is assigned to be this special parameter.

Procedure A ( 0 , 4 , , 1 , 3 , 7 , **…** )

Variable amount of arguments is a purely textual notation. The diagram notation shows working with the Array object. However, an Array parameter needs to be tagged and in the diagram code this is simply shown by displaying the words ‘Arguments’ with the Array object.

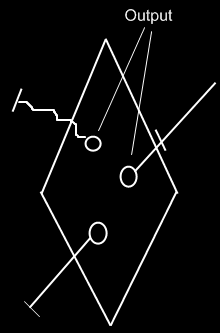
### Return Values

#### Concept

Other programming language used to have only *one* return value. The new computer language can have *any* amount of output values or output objects, and no single parameter is selected to be *the* return value. I guess the concept of one return value dates from when a command was meant to be like a mathematical formula. I do not think, that limiting the amount of return values of a command to one, has a place in a more modern programming language anymore, because we have to realize, that programming is not mathematics.

#### In a Diagram

As discussed in the article *Return Values*, the new computer language does not limit the amount of output objects. Any object readable from the outside, may be an output object.



The correct usage of the term output will be discussed only later, in the *Advanced Command Topics*.

### Parameter Order

#### Concept

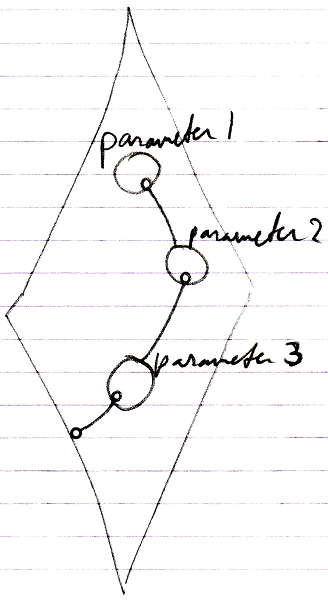
Other programming languages were textual. The parameters were put in a certain order. In the new computer language parameters are more free in order, just like sub-objects inside another object usually have no particular order. The parameters are not really a list of objects anymore. A command call is an object, that has a set of object references in it, that are individually set, and do not have a particular order.

Even though all parameters can be individually accessed by pointing to its symbol or by using its name, you *can* still give the parameters an order of display. The articles, that follow, will elaborate on the diagrammatic and textual expression of parameter order.

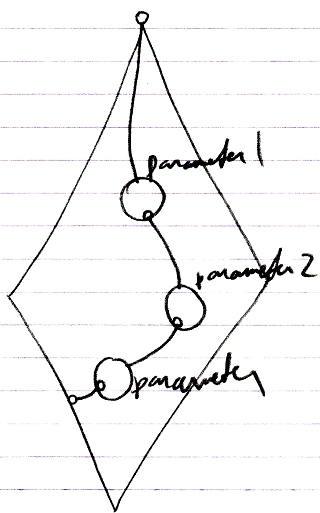
#### In a Diagram

The article *Parameter Order* introduced the concept of an ordered list of parameters. The diagramatic display of parameter order is still not final.

In a diagram the order of the parameters could be expressed just like the order of consecutive commands are ordered by next-command references. See *Execution Control, Normal Execution Order*. This display is yielded over to ordering objects, and ordering parameters as well.

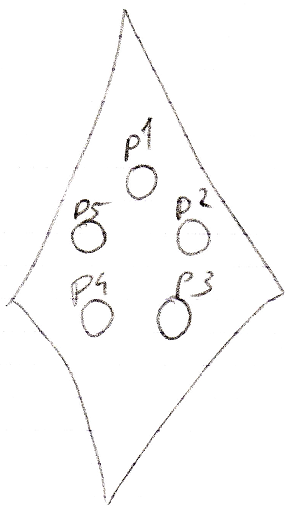


Perhaps the first item will also be denoted like this:



The eventual display is all up to the implementation of *Object Order*. That will determine the ways sorted objects are displayed. The ideas introduced here are preliminary. Eventually parameter order will comply to object order.

The objects may also be displayed clockwise:



And possibly, when there are a lot of items in the collection, you may need scroll through them, because they can’t all be individually displayed:

