|  |
| --- |
| Circle Language Spec: System Objects |

## Assignment

Calls to **Get**, **Set** and **Use** commands might not usually be seen directly. Those may be more likely to be called indirectly. An assignment command might indirectly execute a **Get** on one object and a **Set** on another object, trying to yield over an aspect from one symbol to another.

Different aspects (such as a value, object or class) might correspond with different types of assignment. Below an attempt to list more common types of assignment.

The overview also tries to illustrate, which **Get**, **Set** and **Use** commands might be called while performing the assignment.

Because assignments might occur commonly, an implicit simplified notation is suggested here.

To demonstrate the difference between object-bound aspects and reference-bound aspects, the following notation is employed. When a reference-bound aspect might be **Get** or **Set** then a reference might be displayed with a parent around it:



When an object-bound aspect might be **Get** or **Set** then a targeted object may be displayed without a parent around it:



### Basic Assignment Types

|  |
| --- |
| **Value Assignment** |
| **Value Get 🡪**  **Value Set 🡨**  *Aims to copy a value of one object*  *to another.*  *The result might be:* |
| Object Assignment |
| **Object Get 🡪**  **Object Set 🡨**  *Tries to let one symbol point to*  *the same object as the other.*  *So attempts to yield over the object aspect.*  *The result might be:* |
| Class Assignment |
|  |
| **Use As Class (~= Object Get) 🡪**  **Class Set 🡨** |
| *Tries to use the one object as*  *the class for another.*  *The result might be:* |

In this assignment notation the line type (dashed, wavy) might indicate which aspect would be yielded over. The access mark (the smaller line crossing the longer line) might indicate direction. The diamond shape in the middle would indicate a call to an assignment command.

### Cross-Aspect Assignments

Perhaps a more common way to use the **Class** aspect in an assignment might be to **Get** the **Object** aspect from one reference and assign it to the **Class** aspect of another reference. But it might also be done the other way around: attempting to get the **Class** aspect from one reference and assign it as the **Object** of another reference. It might be called *Class-to-Object* assignment or a *Class-Get* *assignment* perhaps. Less conventional ways of yielding over aspects like that, might be called *cross-aspect* assignment.

Also noteworthy might be, that there could be two ways to get the **Class** aspect: **Get** the **Class** aspect of an *object* or **Get** the **Class** aspect of a *reference*.

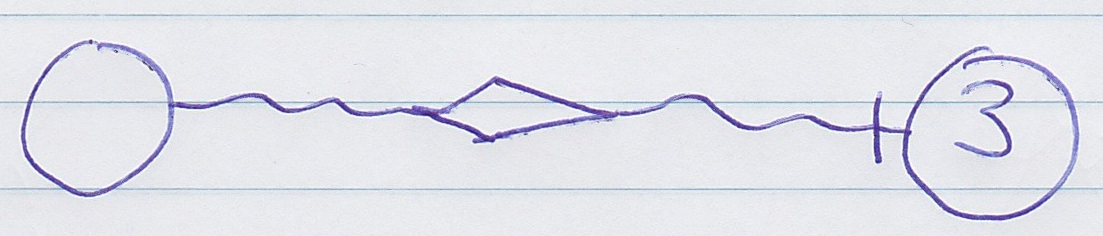
|  |
| --- |
| **Reference-Bound Class to Object Assignment** |
| **Reference-Bound Class Get 🡪**  **Object Set 🡨**  *The result might be:*    *The object reference on the right may now point to the class of the object reference on the left.* |
| **Object-Bound Class to Object Assignment** |
| **Object-Bound Class Get 🡪**  **Object Set 🡨**  *The result might be:*    *The object reference on the right may now point to the class of the object on the left.* |

### Ideas about Assignment

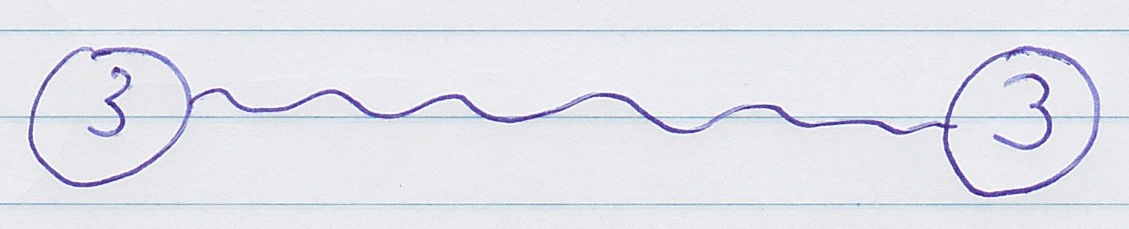
There were some ideas let go of in the main text. They are mentioned here, in case they might be used somewhere else or worth reconsidering at one point.

#### Leaving Out Diamonds for Value Assignment?

Currently the idea might be to express value *assignment* and value *correspondence differently:*

**

*Value assignment*



*Value correspondence*

In the past there was the thought, that value lines might always mean assignment. So that assignment:



*Value assignment*

Might not need a diamond shape:



*Value assignment?*

Since the idea was value lines always meant assignment. But this might conflict with wanting to express value assignment and value correspondence distinctly and consistent with other parts of the language.

Another idea was: still, if a value line has an access mark to indicate direction, what else could it mean but value assignment?



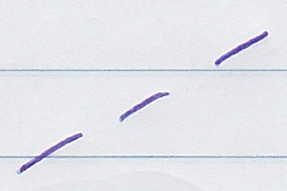
*Value assignment?*

So there still seems to be some wiggle room around the idea. A choice was made to make a notation leading, that has less implicit rules and more equal ways of symbolizing things across the language:

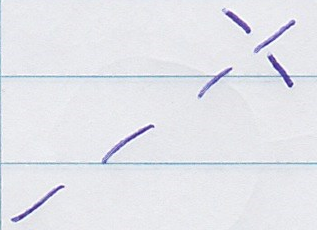
A line might mean correspondence:



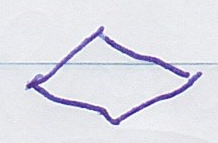
Line type could indicate the aspect:



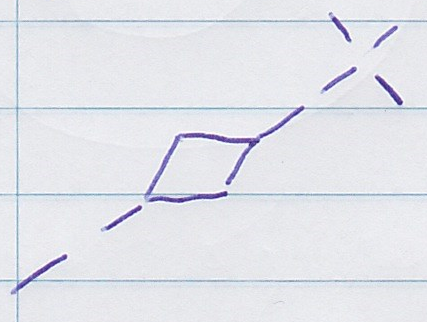
Access mark may mean direction:



Diamond would mean a call:



An assignment call in this case:



#### Opposite Direction Indications?

There used to be an idea to flip the indication of direction in assignment notation.

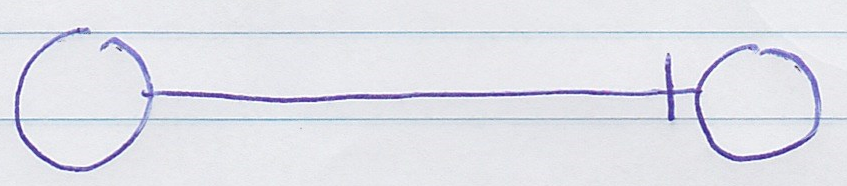
**

**Object Get 🡪**

**Object Set 🡨**

(opposite direction)

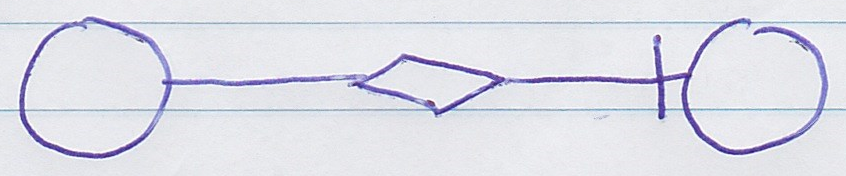
Access marks seem to indicate direction in this language. It may make sense that the access mark might indicate the assignment direction and that the assignment direction means from source to target. But what might make less sense is when the result of the assignment would seem to be in the opposite direction of the assignment direction:



*Result of the assignment*

Perhaps it can be imagined seeing live changes in a diagram, assignment calls be executed live, seeing their effect, it would perhaps not be so intuitive when lines would seem to flip direction all the time.

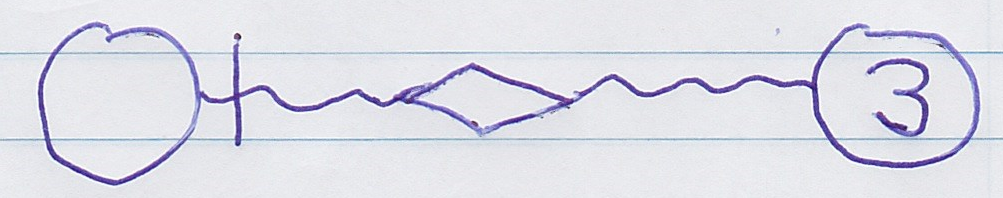
That is why the suggestion was made to make the assignment look more like the direction of the result:



**Object Get 🡪**

**Object Set 🡨**

The idea to put the assignment direction mark at the opposite end, might have some intuitivity to it, when looking at value assignments, since this would sort of indicate the direction of the flow of the data.

**

*(opposite direction)*

**Value Get 🡪**

**Value Set 🡨**

But the price of inconsistency might not be worth this intuitivity, and might actually be less intuitive.



**Value Get 🡪**

**Value Set 🡨**

In the main idea the access mark sort of points out what the value is going to be, rather than where the value will go to.

That said, there is a little bit of doubt. That is why this idea is preserved here.