|  |
| --- |
| Circle Language Spec |

## Pointers

### Brainstorm

The 'pointer-to-pointer' issues seem to be a bit spread over the chapters. When a chapter is explained, afterwards it seems to evaluate how things would look in pointer-to-pointer situations. The idea is that all of those pointer-to-pointer situations might be put here in this chapter instead. Topics like objects, classes, interfaces, assignment, seem to able to live without thinking about pointer situations, and pointers just seems a single problem area that might be desirable to cover separately.

It might be worth highlighting there may be different interpretations of pointers, lines and their direction. They seem to be non-competing. Here is an attempt to summarize some of them:

* Interpretation 1:
  + Line direction would not matter, only aspect correspondence would matter.
* Interpretation 2:
  + Direction tends to point outwards, if inward, this would be denoted with an access symbol.
  + The notational choice would be arbitrary and carry no special meaning.
* Interpretation 3:
  + Directions tend to point outwards, if inward, this would be denoted with an access symbol, like previously.
  + Inward directions would actually be more 'active' redirections/accesses: Pointer-to-pointer redirections, getter accesses, calls to procedures returning an object, etc.
  + Outward directions, would be more passive. They might represent 'simple' pointers, not represent getter calls or anything, more like indications of aspect correspondence.
* Interpretation 4:
  + All symbols would be pointers, kind of like in some languages objects might be accessed through singly-redirected object references (C# assumably).
* Interpretation 5:
  + There would be one symbol in the diagram, that represents the actual object, not a pointer to it.
  + It might be found by first following all outward redirections, then all the inward ones.
  + Where it ends, might be the 'target' symbol: The actual target of the redirections that might be said to be represent the actual object, rather than just a reference to it.

### Brainstorm Ref-Ness

Another topic that might be covered, is a comparison with other languages (even though one of the strategic items is to not try and compare so much in this text, with the idea that 'where would it end?') An exception to the rule could be made here to add a comparison to other language's ref-ness, because Circle seems to be 'make a mockery' of the concept ref-ness in a way. C# or C++ seem to be specific about ref-ness. (C++ might make you specify asterisks \*\* to indicate how many redirections a pointer variable makes; C# and .NET seem to assign intrisic importance to defining parameters as ref or out and what other 'ref-nesses' have you? Anyway, they seem quite specific.) Circle however, seems to make a 'mockery' out of this, because all you need to do is add a line and the ref-ness changes. And the ref-ness does not seem to be specified near the start of the pointer redirection, but you might arbitrarily let redirections be added by the thing you are pointing to. 'mockery' is a meant a bit humoristically here, of course. It is just a notation. If the diagrams might represent something from C#, rules are probably just bound by what you can do in C#. You simply might not be able to add more redirections, or might not validly specify something with not enough redirection. Getter accesses in C# might actually be C#'s own embodiment of indeterminate ref-ness. Or depending how lightly you might want to apply the diagram language, it might not really matter that much, this ref-ness issue and these diagrams. But what might become a splinter in your brain, is that Circle does not seem to have a notation (yet) to specify fixed ref-ness. And what might rub some against the fur, is that Circle seems to like indeterminate ref-ness while some might hold determinate ref-ness in great value perhaps. The notion that there are these ideas about that, might justify thinking about it and perhaps describing a way to elegantly solve it or perhaps find a way to live with things the way they are.

### Target Objects

An object reference can\* point to another object reference, which\* points to another object reference and so on. The\* first\* object found in this redirection, that does not refer to another object again\*, is called the\* *target object*. Even\* though\* any of the\* object *references* can be used like it is the\* object itself, the\* *target object* is considered the\* real object and not just\* a reference to it.

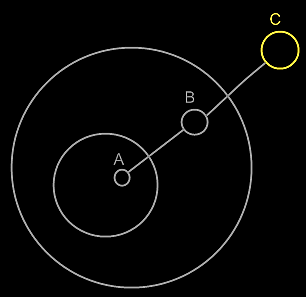
The\* term target object is also used to denote the\* direct\* reference target, not necessarily the\* final target. What kind of target is denoted, will be clear from the\* context.

#### Compared to C++

In C++ you\* had to specify\* in advance\* the\* number of pointer redirections of a variable. In the\* new computer language a symbol can follow any\* amount of indirections, from zero\* to infinity. You\* do not specify\* the\* amount of redirections in advance\*. You\* can just add a redirection by turning\* the\* target object into a pointer.

#### In a Diagram

The\* target object is the\* last\* point in a string of object reference redirections.



Symbol **A** is an object reference to symbol **B**. Symbol **B** is an object reference to symbol **C**. Symbol **C** is the\* target object of both\* symbols **A** and **B**.

The\* idea of target objects is also\* a way to make a single\* symbol in the\* diagram represent the\* actual object, whereas the\* others are just\* seen as references to the\* object: to have the\* actual\* object only\* represented by a single\* symbol in the\* diagram.

## Loose Ideas

### Loose Ideas about Target Objects

Objects,

Target,

2008-07-26

I need to rename the\* term Target Object, Target Class and Target Interface to Final Object Target, Final Class Target and Final Interface Target, because I’m not targeting an object, class or interface, but I’m targeting an object reference representing an object, class or interface.

Also the\* term object target is the\* same as direct object target. That also counts for classes and interfaces.

The\* term Target Object, Target Class and Target Interface have less of a use now. But the\* way they are used now is misleading.

JJ

#### Out of the\* original Symbol documentation

##### Object Trace

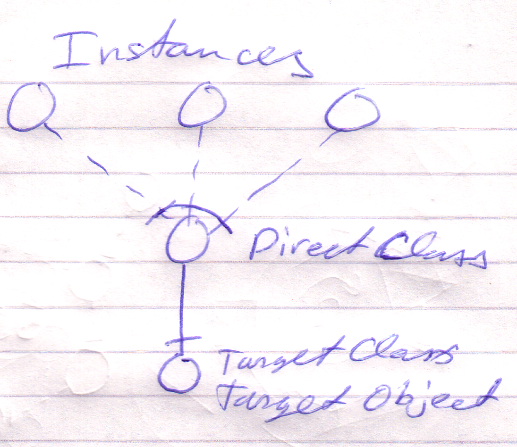
< 2008-10-06 Probably not right anymore. >

To find the\* target object, you\*’d expect to only follow object lines. However, there’s a pitfall: a situation that does not occur a lot, though.

If a type line points to a symbol with an object line, the\* type is a single object.

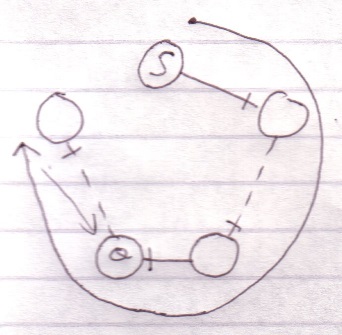


Each instance of the\* type is actually the\* same object.



Therefore, a type line can redirect the\* object of the\* symbol. Therefore, type lines need to be followed to find the\* object.

The\* last symbol pointed to by an object line is the\* object.



This kind of redirectioning is called an *object trace*.

Delegating the\* object aspect is the\* main type of object redirection.

##### Idea

In C++ bepaal je de redirection diepte vooraf:

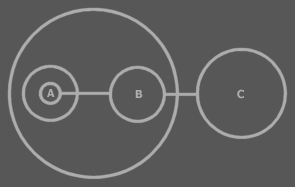
Int \*\*\*TripleRedirected

In Symbol kan je de redirection diepte achteraf bepalen

Als je in C++ een object referenties toewijst aan een object referentie, dan wijs je niet naar de object referentie, maar naar het target object. Symbol heeft meer structurering hier.

##### Multiple Redirection and Final Targets

If an object symbol has an object line to a symbol that again has an object line, there is redirected until a symbol without an object line is encountered: the\* *target object*.



C is the\* target object of A and B.

The\* target object symbol is regarded to represent the\* object for real. The\* other symbols are references to the\* object.

The\* same way there are symbols serving as a *target type* or a *target interface*. Also a procedure has an interface target. A procedure also has a call target and reference target. In both those cases reference lines are followed.