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

## Classes

### Main Concept

#### Concept

The contents of an object can be totally arbitrary\*. You\* can put anything inside an object. This is handy for users, who just want to group objects together into a parent object, much like they group together files in a folder.

However, an object can also select another object to function as its *prototype*. A prototype is also called a *class*. Classes describe the rules by which objects behave. Objects of the same class contain the same kinds of related items and related lists and also support the same commands.

An object will have the same *structure* as its class, but not the same data. The values of the attributes can freely change for each object. *Which* objects are referenced is also different for each object. But initially the object will be an exact replica of the class. The class’s attribute values and object references only function as a default.

Any object can be used as a class. At first there was the idea, that an object could be fixed in its role as a prototype, but in that case\* you\* could not establish a reference to a class anymore.

There used to be a misunderstanding about something. When\* an object does not have a class, it actually *does* *not* have a class. The object can however be *used* as a class. Formerly this was mistaken for an object’s *defining its own* class. But this is not true. An object without a class *does not* define its own class, *just* because it can be *used* as one. An object, that does not have a class, also does not define its own class; it simply has no class assigned to it. Its contents are totally arbitrary\*. *No class* stands for *arbitrariness*.

You have to keep that in mind. To understand why you have to keep that in mind, you need a prime example of a case in which it becomes a problem.

For instance: the article *Class Commands* introduces the concept of *commands and classes loosely coupled*. It says, that when a parameter gets a class, the command will be available in every object of that class. If a parameter has no class, it becomes a problem when you think of that, as the parameter’s defining its own class. Because in that case\*, the command will only be available from objects, that point out *that parameter* as their class. A strange situation. But the real situation is, that a parameter without a class, actually *has no* class. That adds the command to *any* object, because *no class* stands for *arbitrariness*\*.

An object can be assigned a class. An object *reference* can also be assigned a class. If an object reference does not have a class, then the object reference can point to *any* object. That is another example of how *no class* stands for *arbitrariness\**. When an object or an object reference is assigned a class, you can not easily change that class. If an object has a class and you assign another class to it, it would erase the object’s original contents. If an object reference has a class, and it points to an object of that class, then when you change the class of the object reference, what happens to the target object, that still has the original class? These are exceptional situations, for which the most practical behavior needs to be determined in the future.

#### Diagram Notation

Any object can serve as another object’s class. So any object can be the prototype for another object.



When you actually use an object as another object’s class, then its symbol is drawn with a dashed line. A dashed line stands for classes.



In a diagram a class will usually look like that.

If a symbol, that functions as another object’s class, is also referenced as an object, the symbol gets a double border, indicating its dual function as both an object and a class.



In fact it probably does not have a dual function, it is a class, but there are also *references* to the class (established with an object redirection to the class).

### Class Reference

An object’s specification of which class it has, is also called a class reference.

It is expressed in a diagram by connecting an object symbol to its class with a dashed line:



The object on the left has the class on the right.

### Class Commands

#### Concept

Objects of the same class have the same set of commands.

However, commands are separate entities, not defined by a class.

When a parameter of a command is not given a specific class, then the command is available from any object.

This gives an object a lot of commands. But this will not result in a mess, because commands are nicely grouped together inside the object. Each module creates its own group of commands inside an object. If you do not trust a module: do not use the command.

So when the class of a command parameter is not fixed, the command is available from any object. When the class of a command parameter *is* fixed, this makes the command only available from objects of that class. This considerably limits the amount of objects that get the command.

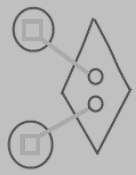
In that sense a *command* defines behavior of a class. Which makes sense from a real-world point of view, because you can always invent new ways to use an object.

A command is available from any object that has anything to do with the command.

#### Diagram Notation

The article *Command Arguments in a Diagram* demonstrates, how a command argument also makes the command part of an object.

A command is executed on an object.



If a command definition does not fix the class of one of its parameters, then the command will be immediately available from *any* object. The command will also be visible in *any* class.

Here is an example of a command definition, two objects and a class. One of the objects has that class.



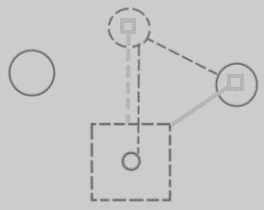
When a parameter is added to the command, and the parameter does not have a class assigned to it yet, the following happens:



Because the command definition got a parameter with no class assigned to it, the command immediately becomes available from any object or class in the system. The command symbols inside the objects and the class are tied to the definition, to indicate mutuality of definition. Immediately the command is very present in the system, because it can be executed on any object.

In theory, all the lines between the squares could have been drawn with a solid line. But the *class* and the *command definition* are tied together with a dashed line, because it is a relation between structure elements. Now all structure elements and their relations are drawn out with dashed lines. It is more intuitive that way.

If you assign a class to the parameter, then the command will only be available from objects of that class. The command will also only be visible inside just that class, not just any class.



The connection between the class and the command definition is now crowded with two lines. The lines are merged together, to express the tight bond between the command parameter and the class command.



This also better expresses, that the two directions of the bidirectional relation between the command and the class are linked.

So in short, this:



When adding a parameter with a class, turns into this:



What is visualized is, that the system got expanded with a connection between the class and the command, which also added the command to the only object of that class.

### Target Classes

#### Concept

A target class is found by following the redirections, that lead to a symbol’s class.

Do not follow more than one class redirection, because if a class points out a class again, then the second class is *another* class object, that the first class is just *based* on. If the class is an object reference itself, you have to follow all object redirections to find the target class object. Then you have found the target class. That’s where redirection following ends. If the class object has a class itself, you might be tempted to follow the class object’s class redirections as well, to find the final target class, but you should not do that. The first class redirection indicates the class. If that class object has a class itself, then the class object is only based on another class, but it *is* a class on its own. An object redirection is just a much tighter bond like that, than a class redirection.

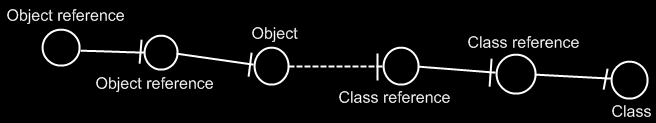
#### Diagram Notation

The concept of target classes is explained in the article *Target Classes*. This article only explains their expression in a diagram.

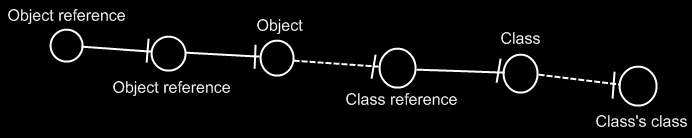
The target class is found by following the redirections, that lead to a symbol’s class.

When you want to find the class of an object, and the object is actually an object reference, you first need to follow all object reference redirections, to find the target object. When you found the target object, you can find the target class, by following one class redirection.

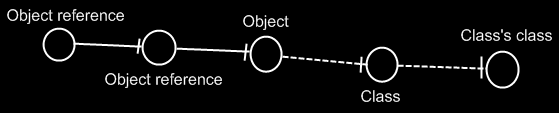
So to find the target class, you first follow *all* the object redirections, then *one* class redirections, then *all* the object redirections and there it ends.



If the class has a class as well, this does not redirect the original object’s class, because the second class is *another* class object, that the first class is just *based* on. An object redirection is just a much tighter bond, than a class redirection.



The target class of the first object reference is the symbol Class, not the symbol Class’s class. The same counts for the diagram below.



If you wonder what could be that different between Class and Class’s class: they could differ in default values. The main point is: finding the target class is about finding the class object.

## Ideas about Classes Main Concept

Pointer to class of,

2008-08-17

Consider the notation of pointing to the class of an object reference, used in the article Class Referrers in a Diagram.

I need a notation for explicitly referring to a pointer or to the class of an object or to the class of an object reference.

Do consider that the target object in a diagram really needs to represents the object. You should not think of it as an object reference, because that will make it harder to see through the system.

JJ

2004,

Every of those objects has a type. The type determines the contents of the symbol. Every object of the same type has the same contents and the contents of these objects changes simultaniously as you edit it.

< 2008-10-12 That is no longer true. Objects of the same class can have different contents. But what does change simultaneously, when you edit the class? >

JJ

Classes,

2009-05-12

Another synonym for class is *type*.

You have to mention this somewhere.

JJ

Classes,

2008-11-13

If you can see object usage, you can not see class-sub-object usage.

You'd have to look at the usage of the sub-objects of the objects of that class,

to see the class's sub-object usage. Indirectly you will be able to see the dependency on a class's sub-object.

Doesn't a sub-object have a reference to the class's sub-object or does the parent

object only have a reference to the class?

JJ

Classes,

2008-11-26

The remark below might give you a clue about an exact sum-up of the uses of classes. One of the uses is having more than one of something. Another use is to selectively have none at all of something, so only a selection of things. Another use is being able to more easily reorganize separate units, if they are separate objects. Another use is being able to reference the same thing from multiple places. That's not a use of classes, but a use of objects. Perhaps all of this is the use of objects, not necessarily the use of classes.

Om van projectfases losse units van te maken, in plaats van één document, kun je makkelijker de units schuiven en rangschikken en slechts een gedeelte van de fases gebruiken, en een fase meerdere malen hergebruiken, eigenlijk precies zoals je dat met classes doet.

JJ

Classes

2009-05-12

I do not know yet how to ventilate changes to classes to their derived objects.

JJ

## Ideas about Class Reference

Class reference,

2008-07-30

Right now I define class reference as being an object’s specification of what is its class.

But accidently I used class reference as a pointer to a class, as being analogus to a command reference.

JJ

### Out of the original Symbol documentation

An object symbol can also serve as its own type:



Then it is an object that defines its own type. But if it has a type line, it redirects its type to another symbol. Then it is no longer its own type, but a mere object from an existing type, also called an instance of a type. The target of the type line is regarded the type itself.



A is an instance, B is the type.

## Ideas about Target Classes

### Out of the original Symbol documentation

#### Tracing Object Aspects

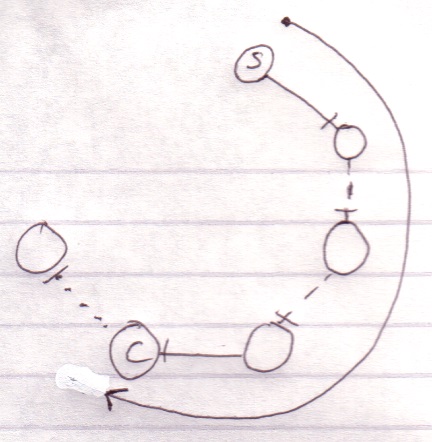
Formerly I’ve said that when you encountered a symbol that doesn’t have a type line, then it is the target type. But in *Object Basics* I said that when a symbol doesn’t have a type line, the object line functions as the type line. Therefore, if a symbol has no type line, the type can still be redirected by an object line.

Finding the aspects of a symbol, such as target object or target type, is called a *trace*.

##### Type Trace

You’ll use type and object lines to trace the type. Follow the type line if it exists, else follow the object line. When you run into a symbol with no type or object line, then that’s the type.

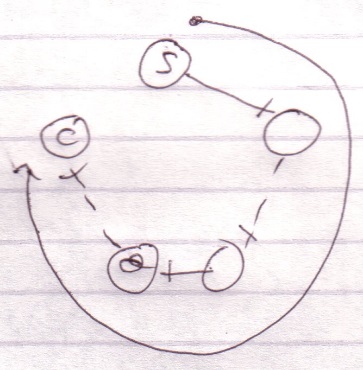
When there is no type line, the object determines the type.



Interface lines are not followed. Note that the target type doesn’t have to be pointed to by a type line.

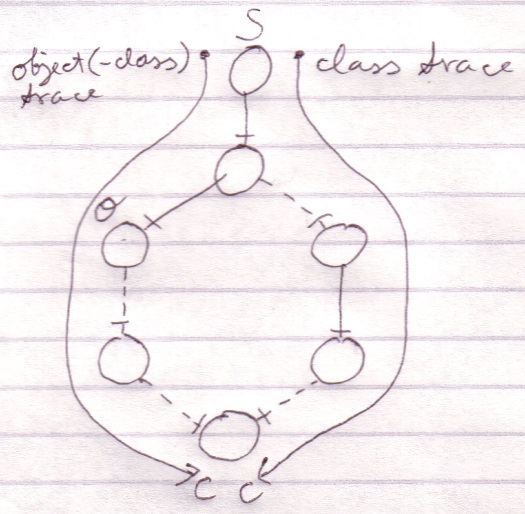
##### Object-Type Trace

The last symbol in the object trace altogether:

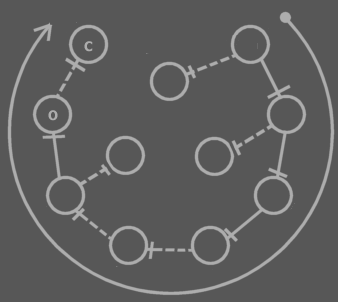


is the target type.

Therefore, *object* trace can also point out to the target *type*. The difference with a *type trace* is that a type trace prefers to follow type lines over object lines and an object trace prefers to follow object lines over type lines. However, both redirections lead to the exact same symbol.



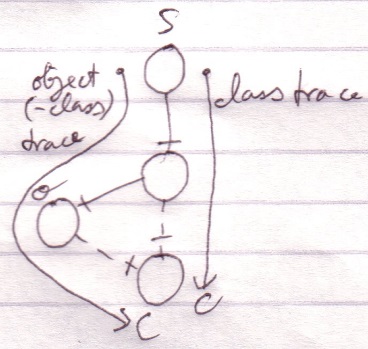
It happens a lot that you want to find out the object and the type in one blow. So you may as well use the redirection of the object trace for the benefit of finding the object and type in one blow. The trace is then called a *object-type trace*.



The last symbol in the redirection altogether is the target type (**C**). The last symbol pointed to by an object line is the object (**O**). Note that the target type may be pointed out by an object line.



When you only want to find out the type, it is better to use a type trace than it is to use an object-type trace. Type trace prefers type lines over object lines. Type lines generally follow less redirections before reaching the target type than object lines do.



##### Tracing is Not Always Hard

If an object symbol has no object line or type line, then finding the target object and type is much simpler, because no redirectioning at all takes place. The symbol is its own object and type.



Traces usually don’t require as many steps as in the examples above.

Targets,

2010-05

> I do not know how it works yet. Now my mind says: follow all redirections, including multiple interface redirections… but in the Target Class story I stopped doing that. Maybe it is just what you want the term Target Interface to define. Maybe it is not even important. I don’t know.

> Perhaps there should be a distinction between interface definition and target interface. I do feel that both the ‘follow only one class or interface step’ version is a concept to be aware of, but the target interface concept would actually be following all redirections to find the object that actually determines the publics.

> Yes. What is now called Target Interface should probably be called the *Interface Object* and the *Target Interface* is the object after following all types of redirections in any order.

JJ