|  |
| --- |
| Circle Language Spec: Commands |

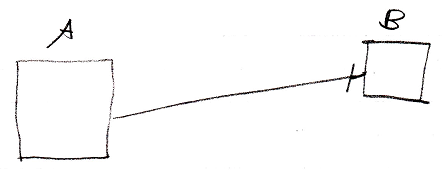
## Command Redirection

### Inactive Command Object Redirection

An inactive command object redirection makes an inactive command symbol a command reference.

An inactive command reference can point to another inactive command as well as to an active command object.

The picture below displays an inactive command, that redirects its definition to another inactive command.



Inactive command symbol A is a square because it is an *inactive* command. Inactive command symbol A redirects to the inactive command symbol B. Symbol B is also a square because it is also an *inactive* command. The redirection from command A to command B is displayed as a solid line, which is an object line. The direction of the line is indicated by a line dissector, close to symbol B.

Symbol B could also be replaced by a diamond, because inactive pointers to active commands are allowed.



### Inactive Command Class Redirection

Inactive command class redirection is less common than inactive command object redirection. Inactive command *object* redirection is a reference to a command definition. Inactive command *class* redirection creates a new command definition for which another command is the prototype.

Inactive command *class* redirection makes a command definition take over all aspects of another command definition, but creates a *new* command definition object, for which the other command definition is the prototype. The new command definition can only redefine the other definition’s *default values*. That’s all that is additionally possible, compared to inactive command *object* redirection.

Inactive command class redirection is analogus to having a class redirect its class, making one object the prototype for another prototype.

An inactive command class redirection can also point to an active command, making an *executable* command object the prototype for the new command definition. That is even less common, but possible all the same.

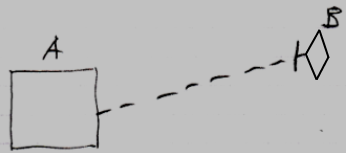
Do remember, that the new prototype can *not* be executed by pointing to it with an active command reference, because the new prototype is a new *inactive* command object, and an inactive command object can not be executed.

#### In a Diagram



Inactive command symbol A is a square because it is an *inactive* command. Inactive command symbol A redirects its class to the inactive command symbol B. Symbol B is also a square because it is also an *inactive* command. The redirection from command A to command B is displayed as a dashed line, which is a class line. The direction of the line is indicated by a line dissector, close to symbol B.

Symbol B could also be replaced by a diamond, making an *executable* command the prototype for a new command object. That is less common, but possible all the same.



### Executable Command Class Redirection in a Diagram

#### Concept

If an executable command redirects its class, then it is a command call. The class redirection points to the command definition of the command call.

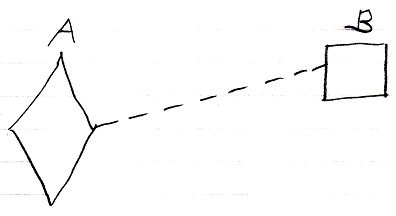
If an executable command *does not have* a class redirection, then it defines its own contents. An object redirection automatically also redirects the class aspect of a symbol. So then it also does not define its own contents. If an executable command, that does not redirect its class or object resides inside another command, then it is called an *active clause*. If the executable command without redirections resides inside an object, then it is simply an executable command with no definition, that can be manually run by a user.

You could let an executable command redirect its definition to another executable command, but this is far less common. But an active command *can* function as the prototype for another executable command.

#### In a Diagram

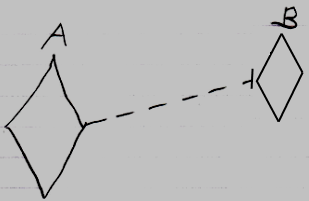
Executable command class redirection is explained in the article *Executable Command Class Redirection*. The current article demonstrates the concept in a diagram.

The picture below displays a command call, redirecting its definition to an inactive command.



Active command symbol A is a diamond shape, because it is an *active* command. Active command symbol A redirects its definition to the inactive command symbol B. Symbol B is a square because it is an *inactive* command. The redirection from command A to command B is displayed as a dashed line, which is a class line. This makes symbol A a command call, for which command B functions as a definition.

You could let an executable command redirect its definition to another executable command, but this is far less common:



### Executable Command Object Redirection

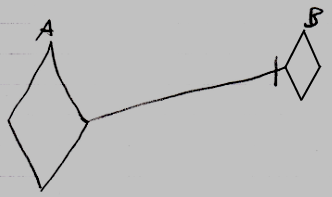
An executable command, that redirects to another executable command, is a reference to a command that could run, or could be running. Say you run the command on a site and you want to navigate away from the site, but keep a reference to it in your user object, that points to a process running on a site somewhere, and take along the reference anywhere you go on the internet.

Executable command object redirection is less common than inactive command object redirection.

You can also object redirect an executable command to an inactive command. If the final target of the object redirections is an inactive command, then you can not run that unexecutable command object through an active reference to it. It is kind of pointless to point to an inactive command object through an executable command reference, so it is very uncommon.

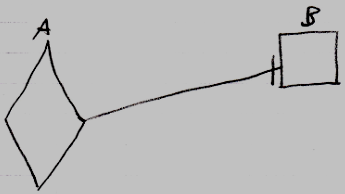
#### In a Diagram

The concept of execution object redirection is explained in the article *Execution Object Redirection*. The current article explains its expression in a diagram.



Active command symbol A is a diamond shape, because it is an *active* command. Active command symbol A redirects its object to the active command symbol B. The redirection from command A to command B is displayed as a solid line, which is an object line. This makes symbol A a reference to an execution running elsewhere.

You can also have an active reference to an inactive command:



But this situation is very uncommon. You can not run command B through symbol A, because an inactive command object can never be run.

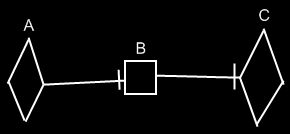
### Target Command Object

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

The term target command 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.

#### In a Diagram

The target command object is the last point in a string of object redirections of command symbols.



Symbol A is an active command reference to symbol B. Symbol B is an inactive command reference to symbol C. Symbol C is the target command object of both symbols A and B. Symbol C represents the command object itself.

### Target Command Definition

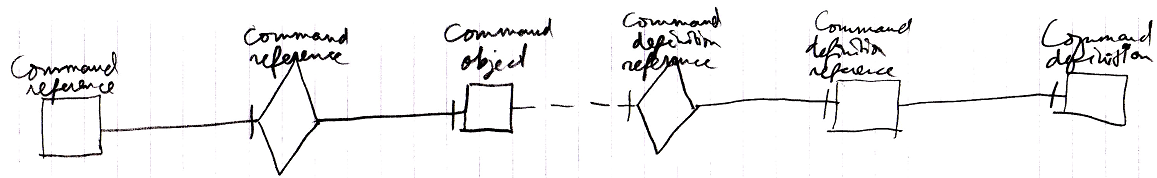
A target command definition is completely analogus to a *target class*.

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

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

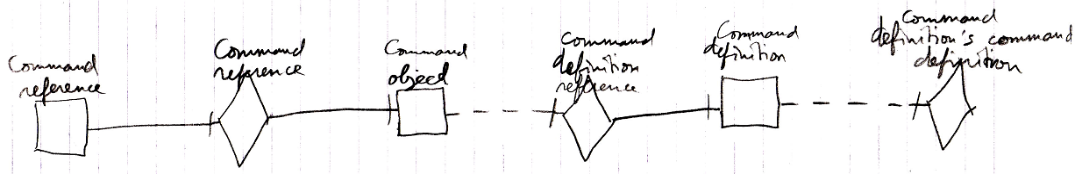
#### In a Diagram

To find the target definition, you first follow *all* the object redirections, then *one* class redirection, then *all* the object redirections and there it ends.

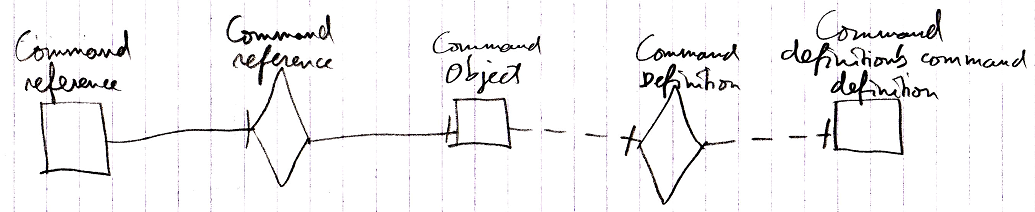


It does not matter if whether it is a diamond, that can be executed, or a square, that can not be executed.

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



The target definition of the first object reference is the symbol Command Definition, not the symbol Command definition’s command definition. The same counts for the diagram below.



### Recursion

#### Concept

Recursion in commands happens when a command contains a call to the same command definition again.

It can also occur when one command calls another command and the other command potentially calls the first command again.

In fact, in any string of commands, that somewhere along the line calls the first command again, all the commands in the string are called recursive.

The *point* of recursive execution is that it is not endlessly recursive after all. At some point, a potential execution should not be executed, in order to exit the seemingly endless recursion.

The problem with endless recursion is solved, by the fact, that private contents of an executable object are only created just before it is actually going to execute.

When a command definition contains a call to the same command definition, then the call does not contain any private content yet, because it is not executing yet.

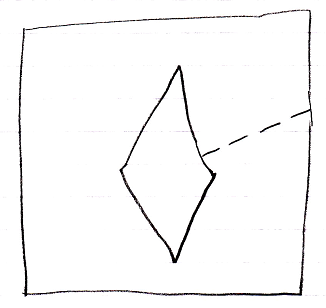
When a command call contains a call to the same command definition, then the call also does not contain any private content yet, before it executes.

Only when the call actually executes, you can see private contents inside the recursive call. So only when recursive calls are actually busy executing, this recursive repetition is present in the system.

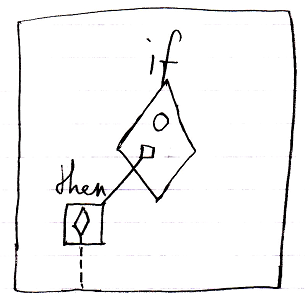
So only creating private content just before a command actually executes solves the problem of endless recursion.

#### In a Diagram

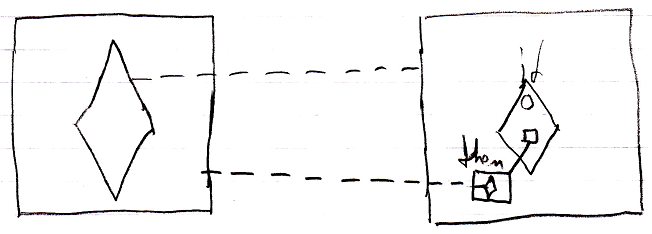
Recursion in commands happens when a command contains a call to the same command definition again.



The recursion is usually conditional, for instance by putting a break on it using an If statement:

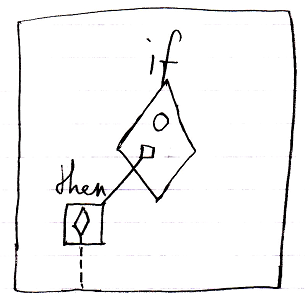


Recursion can also occur when one command calls another command and the other command potentially calls the first command again.

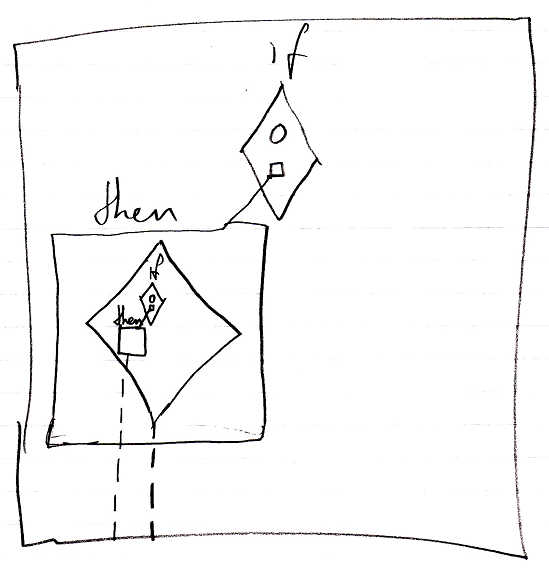


The problem with endless recursion is solved, by the fact, that private contents of an executable object are only created just before it is actually going to execute.

When a command definition contains a call to the same command definition, then the call does not contain any private content yet, because it is not executing.

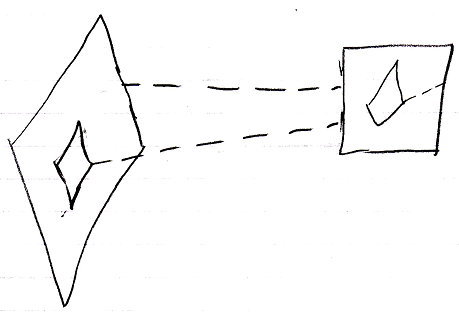


The call in the then clause does not execute, so shows no private contents. Would the then clause be executing, then the diagram would look like this:



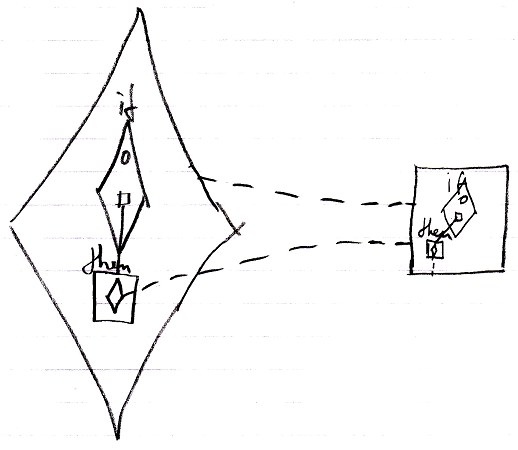
But that never happens, because an inactive command object never executes.

When a command call contains a call to the same command definition, then the call also does not contain any private content yet, before it executes.

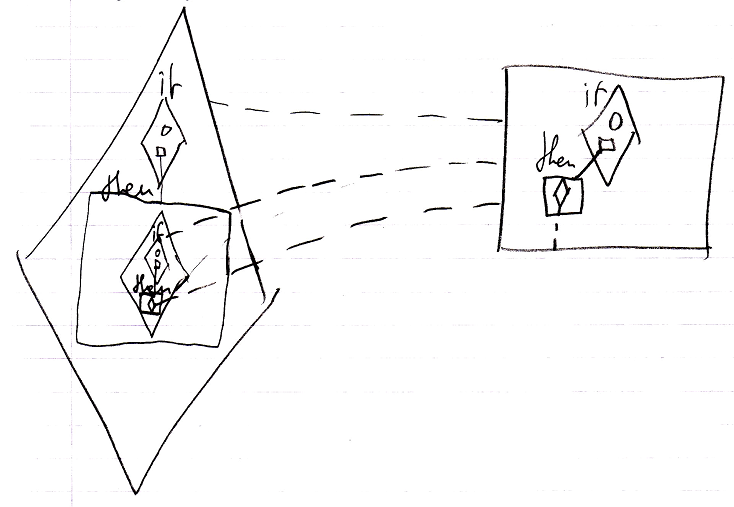


The call on the left contains a call to the same definition, but it shows no private content, because it is not running yet.

The picture below displays a call to a *conditionally* recursive command:

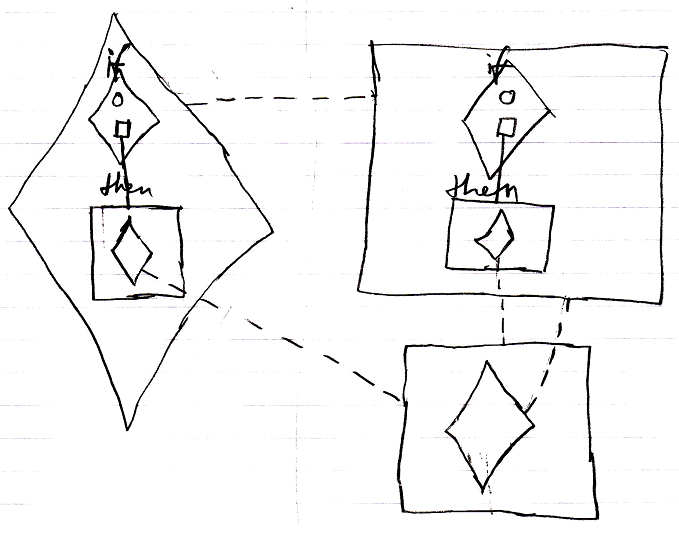


Only when the then clause in the call on the left is actually executed, the call’s private contents are created *and* visible in the diagram.

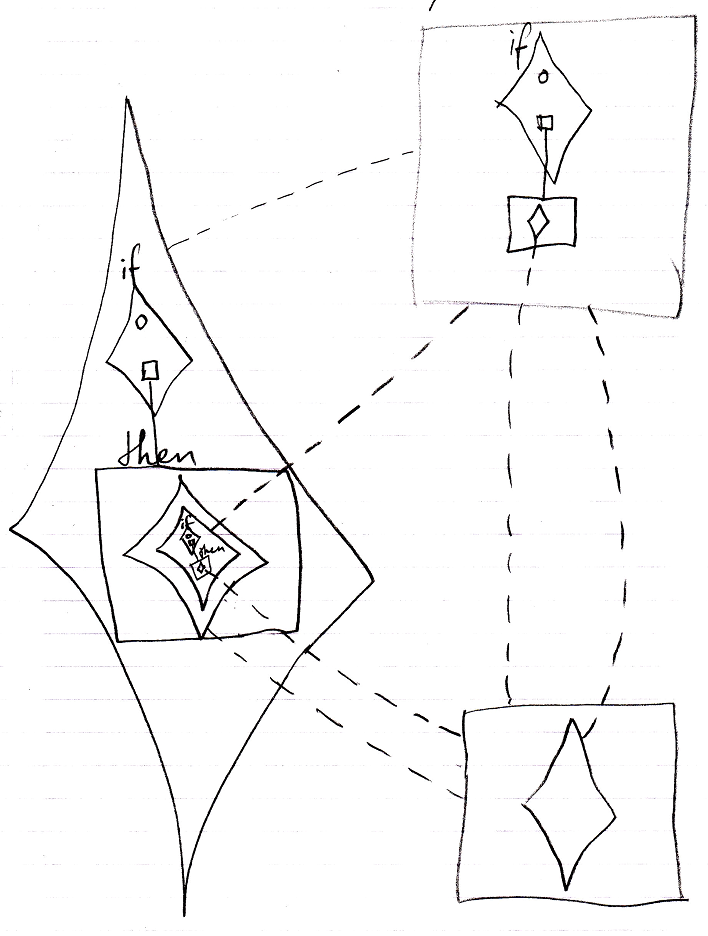


So only when recursive calls are actually busy executing, you this recursive repetition is visible in the system.

Here is a diagram of a call to a command that delegates to another command, that delegates to the first command again. The two command definitions are on the right, the calls are on the left:



The call on the right might be executing, but its call inside the then clause is not executing yet. Only when the then clause is executed, the private contents of the diamond inside the then clause are created *and* visible in the diagram.



Yes, there are a lot of class lines, but each separate call has a separate class line to the right definition. If there is a recursive command running, then each execution of the same command has a separate execution symbol, because there will be an execution contained in an execution, contained in an execution, etcetera.

## Loose Ideas about Command Redirection

### Ideas (about Target Command Object)

(Out of the original Symbol documentation)

#### Execution Trace

To find the execution you do the following.



What you do is that you follow reference lines until you encounter a symbol without a reference line. The last symbol in the trace to be a *diamond* is the target execution.

Tracing the execution target only seems to be relevant when the source symbol is a diamond. It usually is. Usually you’ll only do this trace for a diamond. However, the target execution is also relevant for a square, because if a square’s redirection has a diamond in it, any call to the square is a call to the same execution.

#### Tracing Procedure Aspects

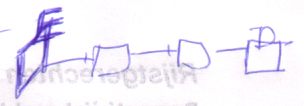
Even though there are only *two* line types for procedure symbols, there are *three* aspects of a procedure symbol to trace:

* Execution
* Definition
* Interface

A definition is a lot like the type of the procedure, while an execution is an instance of the procedure.

##### Redirecting to an Execution

A lot of times you will redirect an execution, but you won’t redirect *to* an execution.



By redirecting to an execution, it is possible for multiple symbols to represent the same single execution.



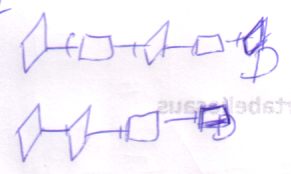
Also note that when redirecting to an execution, the definition is always the same execution.

### Ideas (about Target Command Definition)

(Out of the original Symbol documentation)

#### Definition Trace

The definition trace is quite easy: follow reference lines until you bump into a symbol without a reference line. That symbol’s the definition



#### Execution-Definition Trace

Just as with the object-type trace, the execution trace requires you to find the definition anyway, so when you need both, the execution trace will suffice. It’s called a *execution-definition trace* when you use an execution trace to find both execution and definition.

## Loose Ideas about Recursion

### Out of the original Symbol documentation

* Recursive calls... hmmm... advanced issue. Hoef je je klomp niet over te breken.
  + Recursive calls... hmmm... advanced issue. Hoef je je klomp niet over te breken.

< Maar lijkt een Execution basics. Hmmm… de term Execution Basics is niet goed genoeg dan. De term basics is niet goed. Execution Facts… nah.