|  |
| --- |
| Circle Language Spec: Object Resolution |

## Object Resolution

Object Resolution,

2020-05-24

Note that the Inheritance chapter might have content about Object Resolution topics too.

JJ

### Ideas

#### Inheritance en Overloading (overloaded default properties)

2008-06-11

< 2008-09-02 Inheritance and overloading is very tightly related if you look at it from the following perspective. Perhaps other object resolution principles could also go under inheritance this way. >

Een attribute met sub objecten, die overloaded default properties zijn:

If FutureCancellationsWipedOut = True And FutureCancellationsWipedOut > 10 Then

End If

FutureCancellationsWipedOut heeft dan twee default properties:

Truth As Boolean

Count As Integer

Als je FutureCancellationsWipedOut vergelijkt met een boolean, dan resolvet dat in het vergelijken met FutureCancellationsWipedOut's Boolean property Truth. Als je FutureCancellationsWipedOut vergelijkt met een integer, dan resolvet dat in het vergelijken van FutureCancellationsWipedOut's Integer property Count.

Default Property = Inheritance from a sub object

Iets mag wel inheriten van twee dingen met dezelfde interface, maar als je een van de twee interfaces raadpleegt, moet je wel disambiguaten, dus direct naar een bepaalde interface wijzen. Je kunt niet een referentie met interface A laten verwijzen naar object, dat van twee objecten inherit, die beide alleen maar interface A hebben. Je zult dan moeten kiezen en verwijzen naar een specifieke interface.

Als je refereert naar een default property van iets, dan heb je in je diagram en je text code TOCH een referentie naar de parent van de default property.

Disambiguaten moet dus toch in zowel het diagram als in text code. Als je verwijst naar een object via een referentie met een bepaalde interface, die ook wordt ondersteund door dat object, dan wijs je niet direct naar de interface van dat object, maar naar het object en implicit verwijst het door naar een specifieke

interface, maar dat zie je niet. Dat zie je hooguit wel imaginarily aangegeven in het diagram, doordat de ononderbroken lijn verbonden aan het object, met een doorzichtige stippellijn doorverbonden wordt met een interface, maar dat laatste lijnstuk is imaginary.

JJ

#### From the original Symbol documentation

##### Procedure Resolution

The following methods let a parent mess with the procedures of a child:

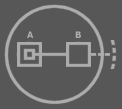
* Overriding
* Exclusion
* Overloading
* Shadowing
* Merging

Overloading, shadowing and merging are about which procedure to call if several procedures have the same name. In text code you couldn’t easily see which procedure of the bunch is called. In diagram code it is always clearly visible which procedure you’re calling, but if they have the same name, then in text code it is less easy to see. Overloading, shadowing and merging work with this phenomenon.

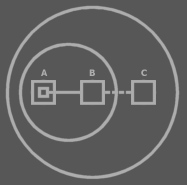
Overriding and exclusion are techniques equally useful in both diagram code and text code. The others: overloading, merging and shadowing, are mostly useful in text code.

###### Overriding

Overriding means that a parent can replace a procedure of its child by another. A child decides whether a procedure of his is overridable or not.

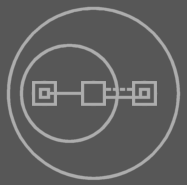


This is a type with an overridable procedure B, called by the child. The access symbol on the right means that the Procedure Reference Set is public. Therefore you can make B redirect to a procedure that the parent defines.



The parent overrode the overridable procedure B. A call to B will resolve to a call to C instead. You can see that overriding means that the parent gives a child’s procedure a reference line to one of its own procedures.

In the replacement procedure, you can still call the base procedure. That way you can both override procedures of the child or extend them.



Overridden procedures are called by the child and that way they’re a means for a child to call a procedure of the parent.

An overridable procedure that is only for communication to the parent is left empty by the child and is only filled in by the parent. This kind of overridable procedure is therefore also called a virtual procedure of the child.

###### Exclusion

Exclusion is already covered extensively, but it can also be regarded a procedure resolution technique. A parent can exclude members from a published object or base object.

###### Overloading

<Overloaded objects>

<A procedure can also have multiple interfaces by using triangle symbols. This enables the possibility to have multiple parameter configurations for the same procedure.

>

<This is an example of how text code can affect the behavior of the diagram. Call line automatically changes when parameters of different types are assigned. Oh, shit, after compilation the text identifiers might not exist anymore. Then it’s gotta become a deep system aspect the overloading, independent of text code. Text code and diagram code would simply be affected by the overloading concept. It should be like this, because of reasons discussed in the language integration chapter.

>

Overloading means that you give several procedures the same name, while they have a different interface. This is handy if you wish to make different variations of similar functionality.

The operator procedure + for instance is richly overloaded. Operator + always has two parameters: the two operands. The two operands can be of many different types and each type configuration requires a separate procedure. Which procedure is called is determined by the types of objects assigned to the parameters.

In diagram code you may see the difference between a call to one procedure or the other. In text code, however, as well as in certain abstract views on the diagram the effect is more subtle and you won’t see the difference between a call to one procedure or the other. This is actually a nice, and wanted effect.

Given is the text code below:

Object Type 1

Integer X

Double Y

Float Z

Procedure Procedure 1

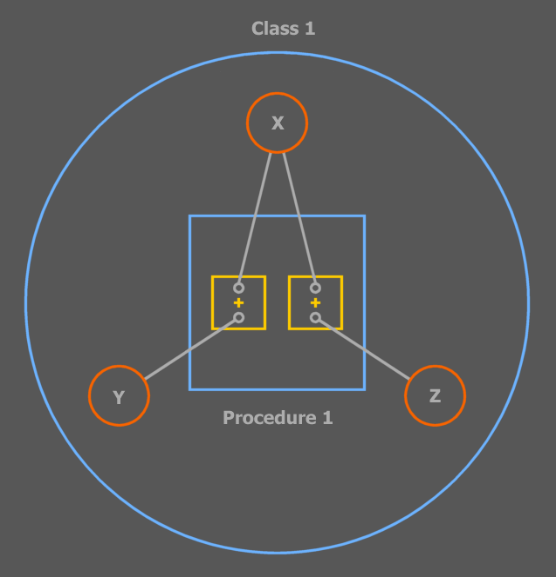
X + Y

X + Z

End Procedure

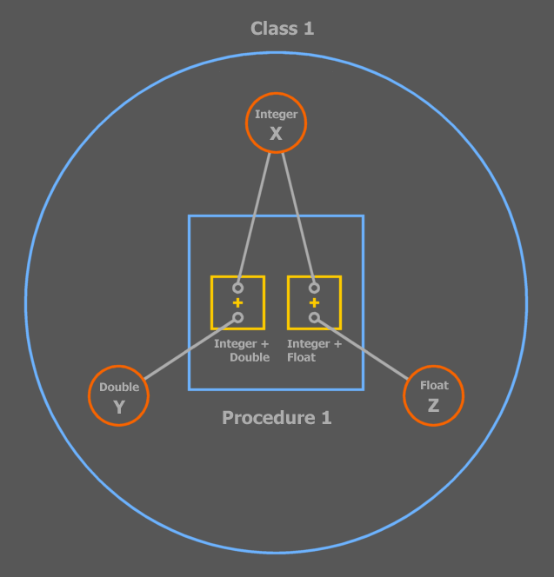
End Object

The following would be the diagram code without mentioning the types of objects nor procedure calls:

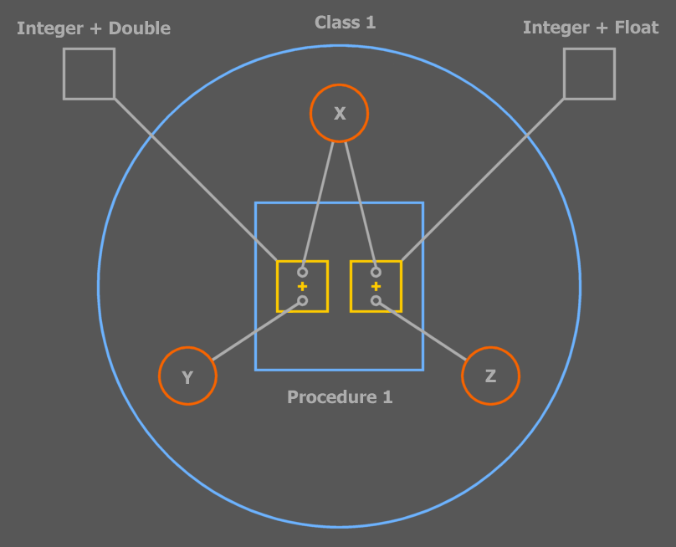


The outer circle is Type 1. It declares the three basic object X, Y and Z. It also contains the procedure Procedure 1, in which are performed two plus operations to which references to the basic objects are passed.

However, when you do mention the types of objects and procedure calls, you see that two different procedures are called.



When you display the called procedures the subtlety is gone, but in this abstraction of the diagram you’ve simply *chosen* to see the difference:



All three diagrams above describe the same code, only in one diagram some parts are left out and some are visible while in another diagram other things are left out or visible. Abstract Symbol is covered in detail in another chapter < >.

The + procedure has a special notation: an operator notation, which is a customized representation of a procedure call and its parameters. This method is discussed in the ‘Text Code’ and ‘Language Integration’ chapters.

The names of procedures that overload eachother are always equal. The configuration of a procedure’s public objects determines which procedure is called. This configuration of public object includes: the object count, object types and object names.

Usually the count and types of parameters are varied, but if two overloaded procedures only differ in parameter *names*, in text code it is required to *name* the object when you call the procedure.

< Picture: text code and diagram code with overloaded procedures that only differ in parameter names. >

When you change a parameter to an object of a different type, then the call line of the call will automatically point to another procedure.

< I need pictures with that. >

###### Shadowing

<Can shadowing change so that the target of a line is changed at run time automatically because of shadowing? If that’s so then shadowing is a way in which text code affects the behavior of the diagram. Same shit problem after compilation no text identifiers anymore>

Shadowing is handy when you use a triangle: a base type or an extra interface implementation.

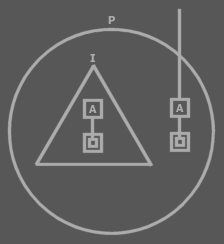


The two lines are calls to the triangle’s members.

When you want to use a name that’s already used by the base type, you can shadow it and calls will be made to the shadowing member instead:



The difference with overriding is that the base type still calls its own member, not the shadowing member.



*I* still calls its own A, while P and anyone else calls P’s A.

A parent can still call the shadowed member, but this would require qualification with the interface name, or the shadowed member will be called:

A



I.A



I.A is inaccessible outside the parent, for the parent has chosen to shadow it.

A member is automatically shadowed if the parent of a triangle defines a procedure with exactly the same characteristics as the triangle’s procedure. The parent can freely shadow any member of the child.

###### Merging

When a procedure has the same name and configuration of public members then in text code there is no way to distinguish calls to either procedure. What happens automatically is that the ‘*nearest’* member *shadows* the member *further* away.



A second option when you can’t distinct a call to one procedure from a call to the other is simply for *both* procedures to be called. This is called merging the two procedures and in that case the call will resolve in calling *both* procedures. If you want to address a specific procedure, you could qualify the call with the name of the container of the procedure, if that would make a difference. In diagram code you can see the two procedures as separate squares and you *can* see the difference between a call to procedure A and a call to procedure B.

< Picture 27: of text code call to merged procedure with next to it the diagram code >

< Picture 28: of text code with qualifications that call a specific procedure >

Merging is handy for instance when a symbol contains two interface implementations and each interface implementation contains a member with the same name and public members.

< Picture of that >

When the merging procedures reside in *exactly the same container*, they cannot be separately called. They are always simultaneously called. In diagram code you might be able to express a call to a specific procedure, but it is forbidden to make such a call, because this would be impossible to express in text code.

< Picture of that (29): the legal situation and a picture with a red cross through it showing a forbidden way of calling and showing that the call would look the same in text code >

That ensures that diagram code and text code are exchangeable and can be mutually used.

< Merging should be a technique, not embedded in the language. When it’s about calling both procedures of the triangles, you have to shadow those procedures with a procedure that calls both. >

### Other ideas

Conversion / Object Resolution,

2009-05-12

What if you have multiple possible conversion commands? One has to be selected for implicit conversion. This has to be disambiguated in that case, through object resolution or explicit conversion. You do not have to do with all the conversion commands in the world, just the ones out of modules you selected.

Implicit conversion is object resolution.

JJ

-----

Object Resolution,

2010-05-18

Object resolution only happens when things are ambiguous.

Things are only ambiguous if you use triangles to merge member sets (except for overloading (disputable)). Then things could start getting in eachother’s way.

JJ