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

## Objects

### Objects

The\* most basic\* element of a computer program is an object. An object represents a thing, an idea or a place, a number or a collection of other things or anything else. All those things are called objects.

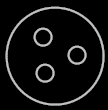
An object in a diagram is represented by a circle, drawn with a solid line:



### Sub-Objects

One\* thing can be composed\* of other things. In that case\* it can\* be said that one\* object contains a number of sub-objects.

When one\* object contains sub-object, the\* sub-objects are placed right inside another object.



### Object Reference

An object can\* contain sub-objects, as explained in the\* article *Sub-Objects*. A sub-object can\* also\* point to another object, that actually resides elsewhere in the\* system, so that\* an existing object can function as another object’s sub-object. In that case\* the\* sub-object is an object reference, that redirects to another object.

An object reference is also called a *pointer*.

When\* a sub-object is actually an object reference, pointing to an object elsewhere in the\* system, then\* the\* symbol representing the\* sub-object will get a line, that is connected to the\* symbol representing the\* actual\* object, which usually resides outside the\* parent object.



The\* sub-object **A** is an object reference to object **B**. Object references usually point outwards, out of an object.

### Related Objects

Because\* sub-objects can\* also be references to object that reside totally elsewhere in the\* system, they are also commonly called an object’s *related objects*.

Related objects are sub-objects or object references inside another object. Related objects are another name for sub-objects and sub-object-references.



All the\* circles inside the\* bigger object are the\* bigger object’s related objects.

### Nothing

When\* object **A** has a related object **B**, sometimes\* nothing is filled in yet\* as the\* related object. In that case\* it is said that **B** is **Nothing** or **null**.

When\* a related object is nothing, a cross is filled in inside the\* shape:



Other shapes introduced later can also\* be marked out\* with a cross, to indicate\* nothing\* is filled in.

### Values

Most objects are just composed\* of sub-objects. But\* some objects represent a simple\* value, like a number. An object can\* store a piece\* of binary data on top of\* storing references to other objects. It is mostly simple\* types of objects, that need to store a small piece of binary data. But any object is free to reserve a piece\* of binary storage to use at its own discretion\*.

The\* value of an object can not be directly\* displayed in a diagram. A binary value first\* needs to be converted to text, that can be displayed on screen. The\* conversion of a binary value into text is implemented as a concept. This concept is called *literals*. An object can support the\* concept, to be able to\* display the\* binary value as text. A literal is a textual representation of the\* binary value of an object. A literal can be displayed in the\* diagram. The\* binary value itself can not.

### Multiplicity

Multiplicity as a coding principle is the\* distinction\* between single and multiple.

A related object can\* be a single object, but related objects can\* also\* be part of a list. In that case\* the\* list of objects gets a name, but the\* individual objects do not.

A single\* related object is also\* called a *related item*, while\* a list of related objects is called a *related list*.

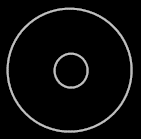
A single\* related object can\* hold a reference to one\* object, but\* the\* reference does not have to be filled in, in case of which\* the\* reference points to no\* object at all. So\* a related item has a multiplicity of between 0 and 1. In some\* other programming languages this is expressed as: 0 .. 1 . A related *list* can\* be filled with zero\* or more related objects. In some other programming languages this is expressed as: 0 .. \* .

The\* two types of multiplicity are also\* called 1 and *n*. A related item is often called a 🡪 1 (say: ‘to one’) related item. A related list is often called a 🡪 n (say: ‘to en’) related list.

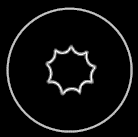
Other types of multiplicity (such as 1 .. \*) are not part of the\* coding principle of multiplicity. The\* fundamentally necessary types of multiplicity are 1 and n . Other types of multiplicity can be implemented as a *concept* programmable in the\* new computer language itself, rathe\*r than a *principle* that the\* system cannot do without.

In Computer Language 0.9 there was also a multiplicity called x. X was a list with a fixed number of items in its collection: for instance a list of exactly three items. Each of the\* items had a name. The\* idea was abandoned, because the\* effects of synchronizing the\* two sides of the\* relation were unpredictable (see the\* article *Relation Synchronization*), and 🡪 x could could easily be replaced by for instance three separate related items.

If\* an object contains a related item, it is displayed in the\* diagram as just a circle contained by another circle.

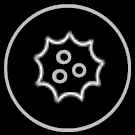


If\* an object contains a related list, the\* related list is displayed inside an object as a *nonagon*.



The\* reason this shape is used, to express a related list, is because it has many corners, symbolizing a multitude. It is also like circles would fit in between the\* pointy corners. This way it seems to be the\* inversion of a circle: the\* opposite of a singular object.

The\* items the\* related list are placed inside the\* nonagon again.



### Attributes

An attribute is a value stored inside an object. An attribute is actually a sub-object with special properties.

An attribute is often an object of a simple\* nature, for instance a number, a boolean, a date or a simple\* piece of text. But an attribute can really be any type of object.

The\* main aspect that makes a sub-object an attribute, is the\* fact that it has a fixed logical residence\* inside its parent. The\* parent is the\* sole container of it. It is not an object that can be shifted around like other objects. It is fixed inside the\* parent. An object usually does not get a fixed logical residence\*, so this is what is special about an attribute.

Another aspect that makes a sub-object an attribute, is that it is always created, never destroyed, never recreated again, and a different\* object is never assigned to it. Another object’s *state* can be assigned to it, but that’s all that can be changed about it.

A third aspect of an attribute, is that it usually has a small piece of\* binary data stored directly inside it. For instance a number is stored as a piece of\* binary. Direct storage of a piece of\* binary data is not required for an object to be considered an attribute, though\*.

A logical aspect of an attribute is that it is more part of what the\* object *is*. The\* other sub-objects are often just considered references to other objects, not as much part of the\* object itself.

You\* can point to an attribute, but an attribute can *not* be a pointer to something else.

If\* an attribute either becomes a pointer, gets destroyed or recreated it would loose its status as attribute, and will just be a related object again.

Any object reference could be made an attribute. The\* attributes are scattered around an object’s list of related items. The\* attributes could be returned by the\* object as a separate list, instead of looking them up in an object’s collection of related objects.

Here follows a summary of the\* aspects of an attribute:

- Requirements:

- Fixed logical residence\* inside an object

- Always created, never destroyed, never recreated, never a pointer

- Logically:

- More part of what the\* object *is*

- Usually:

- Has a binary value stored directly inside it

- Is an object of a simple\* nature

- And an extra:

- An object can return a list of just its attributes, instead of just all related objects.

#### Attributes in a Diagram

An attribute is a value stored inside an object. An attribute is actually a sub-object with special properties, as explained earlier. Because an attribute is an object, it is represented by a circle drawn with a solid line.



An attribute is often an object of a simple nature, for instance a number, a date or a simple piece of text. The\* value is textually expressed. An object’s literal value is shown inside the\* object, usually in the\* center of the\* symbol:



Or closer to the\* top of the\* symbol if any other symbols are shown inside the\* attribute.



### 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.

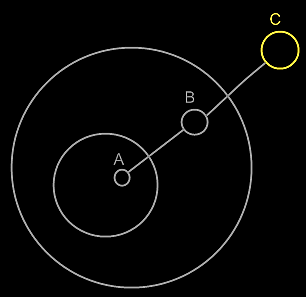
The\* term target object is also used to denote the\* direct reference target, not necessarily the\* final target. What kind of target is denoted, will be clear from the\* context.

#### 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

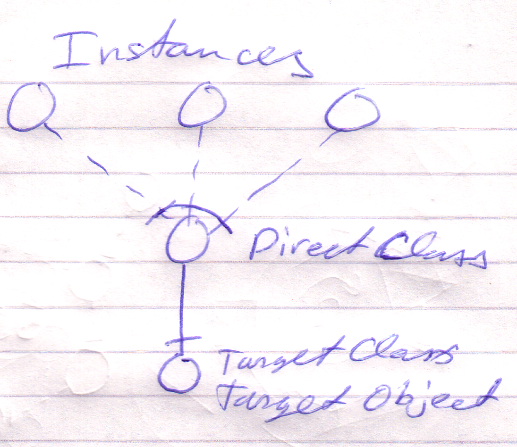
< 2008-10-06 Probably not right anymore. >

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.

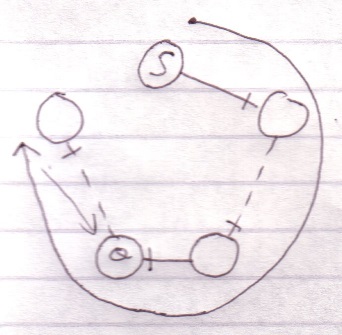


Each instance of the\* type is actually the\* same object.



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:

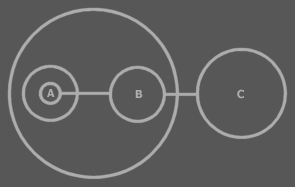
Int \*\*\*TripleRedirected

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.

### Loose Ideas about Multiplicity

Multiplicity,

A collection symbol only makes

sense when the\* class of an object references is fixed,

because when classes are not fixed, you\* can already

put any amount of objects inside another object,

but with a fixed class this is not the\* case.

JJ

Multiplicity,

2010-05-15

Actually, one of the\* points of putting objects inside a separate list is that the\* list can centrally control certain aspects that each item of the\* list must conform to.

JJ