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

## Objects

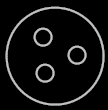
In object oriented programming, an *object* might be considered the most basic element of a computer program. An object might represent a thing, an idea or a place, a number or a collection of other things or possibly anything else. All those things might be called objects.

An object in a diagram might be represented by a circle:



### Sub-Objects

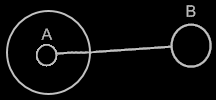
One thing might be composed of other things. For that, an object might contain a number of *sub-objects*. This might be drawn out in a diagram as circles inside another circle.



### Object Reference

Another possibility is that a sub-object would point to another object, that might resides elsewhere in the system. That way an existing object might serve as another object’s sub-object. In that case that sub-object might be considered a link or reference, that could redirect to another object. Such an object reference might also be called a *pointer*.

The following picture may show a sub-object that would depict an object reference. It may be pointing to an object elsewhere.



That sub-object's symbol would be given a line here, connected to the symbol that may represent the actual object, which may reside outside the parent object. Sub-object **A** would be a reference to object **B**. It would be a notational choice, that the direction would point outwards.

### Nothing

Sometimes nothing would yet be filled in for a related object. To display that in the diagram a cross might be placed inside the shape:



An object reference would then be *Nothing* or *null*. Shapes other than circles may also be given a cross drawn inside it, to indicate it is empty.

### Multiplicity

A simplified description of multiplicity might be that it is the distinction between single and multiple.

A related object might be a single object, but objects mught also be part of a list. It could be that this list of objects is given a name, while its individual items might remain nameless.

A single related object might be called a *related item*, while a list of related objects might also be called a *related list*.

A single related object might hold a reference to one object. But the reference might not be filled in. Then it would point to no object at all. Therefore, it might be said that a related item has a multiplicity of between 0 and 1. This might be expressed as 0 .. 1 in some notations. A related *list* might contain zero or more related objects. In that case the multiplicity would be between 0 and n. This might be denoted as 0 .. \*

<< comparisons >>

These two types of multiplicity might also be called 1 and *n*. A related item could be called a 🡪1 (pronounced ‘to one’) related item. A related list could be called a 🡪n (pronouced: ‘to en’) related list.

<< exceptional cases >>

There might be other types of multiplicity not part of the notation just yet: Multiplicity not between 0 and 1, but precisely 1 and the multiplicity between 1 and n. They may boil down to a special case of single and multiple where it is required to always have at least something filled in.

<< comparisons >>

Other types of multiplicity (such as 1 .. \*) might not be part of the multiplicity notation just yet. Just what might be the main two types of multiplicity might be given a place here: 1 and n.

In case of a single item this might be displayed in a diagram as a circle:



In case of a list this might be displayed in a diagram as a nonagon:



Items in the list might be placed inside the nonagon again:



### Values

Some objects might be composed of just sub-objects. On the other hand, there can be objects that represent a simple value, like a number. Simple types might commonly store a piece of binary data. But one idea is that any object might be free to reserve some binary storage to use at its own discretion. An object might store some binary data, next to those references to other objects. It might be able to store both.

A value of an object might not be directly displayed in a diagram. A binary value might first be converted to text, that could be displayed on screen. This text might be called a *literal*. A literal would be a textual representation of the binary value of an object. A literal might be able to be displayed in a diagram. The binary value itself might not be.

### Attributes

#### Attributes might be Objects

Attributes might not be something intrinsic to Circle language, but more of a concept that some may be familiar with.

In one interpretation an attribute might be thought of as an object with specific characteristics. If an attribute would be an object, it might be represented by a circle drawn with a solid line.



What might an attribute be compared to other objects?

#### Values

An attribute might be an object of a simple nature, for instance a number, a Boolean, a date or a piece of text. An attribute might be a value that could be stored inside an object.

The value might be textually expressed. An object’s literal value might be shown inside the object, possibly in the center of the symbol:



Or perhaps closer to the top of the symbol if other symbols would be shown inside the attribute.



An attribute might be an object that has a piece of binary data stored inside it. For instance a number might be stored as a piece of binary. Storage of binary data alone might not turn an object into an attribute.

#### Fixed Logical Residence

Another aspect that may make an object an attribute, could be that it might have a fixed logical residence inside its parent. The parent would be the sole container of it. It might not be an object that could be passed around like other objects might. It would be fixed inside its parent. Usually an object might not get a fixed logical residence, so this would be something special about an attribute.

#### Life Time

Another aspect that may turn an object into an attribute, could be it might be always created, never destroyed, never recreated again, and never assigned a different object to it. Another object’s *state* might be assigned to it, but that might be all that could be changed about it.

#### Part of Parent Object

It might be said that an attribute would be more part of what the object *is*. The other sub-objects might be considered references to other objects, not as much part of the object itself.

#### Not a Pointer

Something might point to an attribute, but an attribute might *not* be a pointer to something else.

When an attribute would become a pointer or would be destroyed or recreated it might give up its status as attribute, and might just be considered a related object again.

#### Summary

Here follows an attempt to summarize what might be the aspects of an attribute:

* Might be an object of a simple nature.
* May have a binary value stored inside it.
* Sort of more part of what the object *is.*
* May have a fixed logical residence inside an object.
* Might be always created, never destroyed, never recreated, never a pointer.