|  |
| --- |
| Circle Language Spec: Black Boxes |

## Black Boxes

### Introduction

The concept of *Black Box* is also known as the concept of *Public and Private*. *Private* makes something only accessible inside the object. *Public* makes something accessible outside the object. That way, the inner workings of an object are hidden away and you only work with the input and output.

The code generator version of the the computer language (version 0.9) simply borrowed the public and private concept of the target programming language for which code was generated. The *generic module* version of the new computer language (version 2.0) did not have any black boxing yet at all. The black box implementation of the version to come is worked out in this documentation folder.

The main articles about black boxing are finished, but you will also find a lot of unfinished material at the end of the documentation, which might be worked out later. Nevertheless, the main idea is worked out completely, and it is clearly denoted which material is not finished.

The idea for the future is that *user* access control and *black boxing* access control will become a single concept of ‘what has access to what’, but that will not immediately be done at first. A simpler black box method is worked out first: one much like what you see in contemporary programming languages. However, this method is adapted to the new computer language, because the new computer language has a different view on objects and commands. Which commands get access to the private contents of which objects is less obvious, because a command is not part of a specific object or class. In fact, no command gets access to the private contents of an object, unless that command is a *friend* of the object. This creates a tighter bond between the command and the object. Another thing, that is different in the new computer language, is that commands are no more than objects. They are the same thing. This really impacts the way you think about black boxing.

### Main Concepts

#### The Black Box Principle

Most computer programming languages make it possible to apply the *black box* principle. Using the black box principle allows you to hide and protect the inner workings of an object. You only get to see some properties, parameters and a selection of commands through which you can control the object while the inner workings are hidden away and protected.

There are two key purposes of the black box principle: data protection and complexity hiding.

Complexity hiding can also be called abstraction.

#### Private & Public

The inner workings of an object or command are its *private* members. You only get to work with an object or command’s *public* members.

So the privates of an object are protected and hidden away, while the public contents can freely be used.

#### Friends

There is no point to private content if it is not used by anything. Even though private contents are protected, *something* needs to access and protect that data.

An object’s public contents can be accessed by anything that has a reference to the object. Private contents can only be used by *friends*. Friends are objects that have access to the private contents of another object.

What other programming languages call class methods, the new computer language calls friend commands of a class object.

The object with private contents declares which other objects are its friends.

The friendship does not automatically go both ways: the *befriended* has access to the privates of the friend, but the friend does not have access to the privates of the befriended. For that you would have to establish a mutual friendship.

#### Friend Relations

A befriended object declares which object is its friend. The friend object needs to be able to access the befriended’s privates. Therefore the friend needs a reference back. So a friend connection requires a mutual relationship.

It is not like one class gets access to the privates of another class. It is more specific than that. A friendship is a relation between specific objects. A friendship between objects applies to only one specific relation. One object gets access to the privates of another object through one specific reference.

Friendship can also be applied to related lists instead of just related items.

An item or list in a specific relation is typed as friend.

#### Accessible & Inaccessible

Black boxing is based on two simple concepts: *accessible* and *inaccessible*. You can express those concepts as follows:

|  |  |
| --- | --- |
| Accessible | Inaccessible |

The concepts are not litterly used inside the language, but all the other black boxing concepts arebased on it.

The notation of Accessible and Inaccessible *can* be used in abstract diagram expression, though, which does not have to be syntactically correct, but is only used to make a point.

#### Access Controlling System Aspects

Each aspect of a symbol can be separately access controlled. Those are aspects such as:

- Object

- Class

- Value

Each aspect’s Get and Set are separately access controlled, for instance:

- Object Get

- Object Set

- Class Get

- Class Set

- Value Get

- Value Set

So when Object Get is Public, you can reference the sub-object.

When Value Set is Public, you can assign a value to the object.

When Value Set is Private, you can not assign a value to the object except through Friend objects.

Only *system procedures* get access controllers. Objects do not get access controllers directly. Access to an *object* is controlled by access controlling the Object Get procedure.

There is not just one kind of Private or Public. There is a separate Private and Public for every Get and Set of every system aspect.

#### Summarized

System aspects of sub-objects are access controlled by their parent object.

You access control system aspects, making their Get or Set, Private or Public.

A normal reference only offers access to the Public content of an object.

You can type an item or list in a specific relation as Friend.

When a Friend has a reference to an object, all its sub-objects are accessible, even when they are Private.

The full access that Friends get, applies to accessing the direct sub-objects, not deeper sub-objects.

### Public & Private

#### Public Set / Access Connectors

The general notation of an access connector is as follows:



The sub-object is access controlled by the parent object. The line with the stick at the end is the access connector.

It is also the general notation of a *Public* access connector.

More specifically: it is a *Public Set* access connector, because it denotes, that you can make it point to something.

#### Public Get

The notation of the Public *Get* access connector is:



Because this denotes, that you can point *to* it.

#### Private Set

The general notation of Private is:



The cross expresses ‘inaccessible’ or better said ‘restricted’. Access is restricted to only a limited group of objects: the *friends*.

More specifically: it is a Private *Set* connector, because it denotes that making it point to something else is restricted.

#### Private Get

The notation of the Private *Get* access connector is:



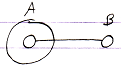
Because this denotes, that pointing *to* it is restricted.

### Friend Declaration

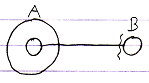
This article covers friend declaration in several situations.

#### Friend Declaration, in a Unary Relation

The following is a unary relation:



In the following picture object A is declared *friend* of object B:



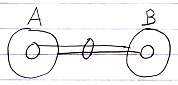
Object A can now access anything private of object B through the reference contained inside object A.

It is object B who declares itself Friend, so object B gives access to its private members.

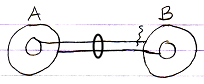
But the friend declaration is part of **A**’s reference to B, because it makes the reference *special*, using the friend access symbol.

#### Friend Declaration, in a Dual Relation

The following picture expresses a dual relation:



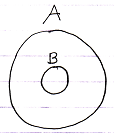
In the following picture B is declared friend of A:



A can access everything of B, even the Private stuff. It does not go both ways: B can only access the Public contents of A.

#### Friend Declaration, Container is Friend

In the following picture, a related object is only displayed as contained inside another object:



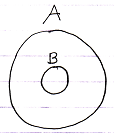
To declare A as friend of B you can use the following notation:



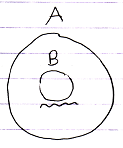
Now B can access everything of A.

#### Friend Declaration, the Contained is Friend

In the following picuture a related object is only displayed as the container of another object:



To declare B as friend of A you can use the following notation:

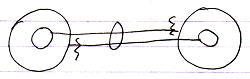


A can access anything of B.

There is no line to place the friendship declaration on. The relation is implied by the fact that something is contained inside something else.

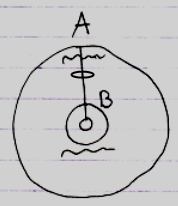
#### Mutual Friendship Referential Notation

Mutual friendship can be expressed as follows in a referential notation:



#### Mutual Friendship Containment Notation

Mutual friendship in a containment notation would look as follows:

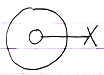


### Notations of Private

Private access connectors get a different notation in different situations.

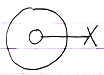
#### Private in a Definition

In a definition the private access connector is a cross:



This is because access is restricted.

Private Set in definition:



Private Get in definition:



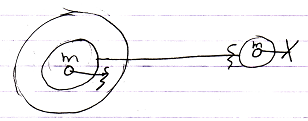
#### Private in a Friend Reference

Inside a friend reference, a Private member becomes accessible, so the access connector it is displayed using a wavy line:



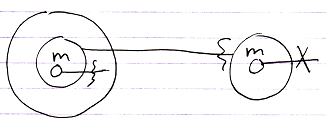
Which expresses *accessible*, but only to a select group of objects.

The following diagram expresses a member with Private Set accessible from within the befriended:



The circle on the right is the definition, so the its member is displayed with a private access connector there (a cross). The access connector within the Friend is different. It is drawn with a wavy line, expressing ‘access conditionally’. In that situation the Private access connector is also called a Friend access connector.

The following diagram expresses a member with Private *Get* accessible from within the befriended:

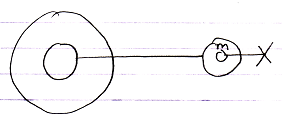


You can differentiate the Friend *access connector* from Friend *declarations* because an access connector has nothing at the end of its line.

#### Private in a Normal Reference

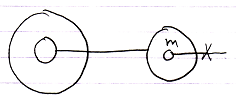
In normal references, private members are *not shown at all*.

The following diagram shows a member with Private Set not accessible or visible from in a normal object:



And that is where the complexity *hiding* kicks in. The circle on the right is still the definition, so it does show the Private member while the normal reference on the left does not show the Private members of its related object.

The following diagram shows a member with Private *Get* not accessible or visible from within a normal reference:



### Friend Declarations in Instances

The points made in the section *Notations of Private* are demonstrated here again in a different situation, where objects have external definitions. To summarize the rules:

- Privates visible in definitions

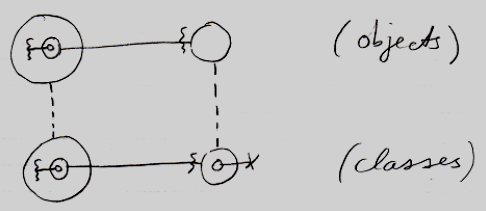
- Privates invisible in normal references

- Privates visible in friend references

In this section of the documentation it is demostrated how this looks if objects have an external class, as opposed to defining their own contents and behavior.

The main thing clarified here is: “when do you show privates and when do you not”. The same rules apply here as in traditional text code, but now they are demonstrated in a diagram.

#### In a Unary Relation

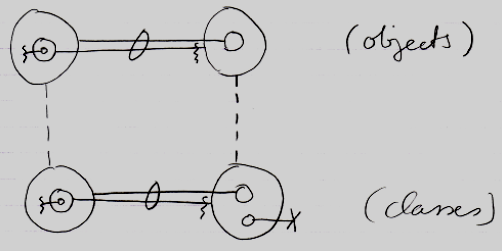


The classes show privates, but the objects do not, unless they are Friend.

The effect that you see, is that wavy access connectors stay there in the objects, while cross connectors disappear in the objects. So the main rule is: crosses disappear, wavy lines stay visible.

#### In a Dual Relation

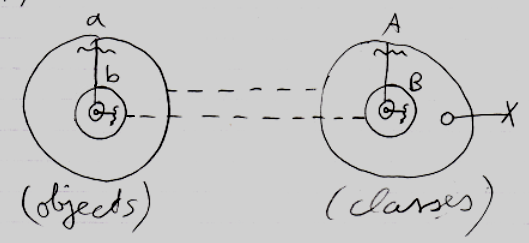
The main point to the diagram below is that the classes show privates, but the objects do not, unless they are Friend. It shows the situation in case of a dual relation.



The effect that you see, is that wavy access connectors stay there in the objects, while cross connectors disappear in the objects.

#### Container is Friend

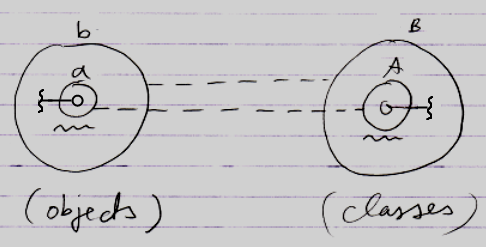
The main point to the diagram below is that the classes show privates, but the objects do not, unless they are Friend. It shows the situation where the *container* is Friend.



The effect that you see, is that wavy connectors stay there in the objects, while cross connectors disappear in the objects.

#### The Contained is Friend

The main point to the diagram below is that the classes show privates, but the objects do not, unless they are Friend. It shows the situation where the *contained* is Friend.



Here something different is going on in the notation. Earlier it was stated that: crosses disappear, wavy lines stay visible. But the picture above contains no crosses at all.

In the classes on the right, something unusual is going on.

The problem is, that *A* is both a definition and a reference at the same time.

A is:

- a definition

and also:

- B's (Friend) reference to A.

In a definition it should be a cross. In a Friend reference it should be a wavy line.

To express the member’s accessibility, the choice was made to display it with a wavy line and not with a cross.

This is simply a side-effect of the notation where *the contained is Friend*.

#### Loose Ideas

##### Objects Take Over Class Access Control

Objects take over the access control properties of their class.

< And references take over object-access control. >

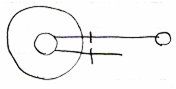
… is objects taking over friend relations from classes as straightforward as you’d think?

When it is classes, the friendship counts for all the objects derived from that class.

Usually, access to procedures is controlled in a *class* and each of its objects adopts those same access settings.

### Public & Friend Connections

An actual connection does not replace the access *connector:* an access connector is always displayed, because others can still connect to the object as well:

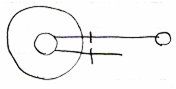


The line at the top is an actual connection.

The line at the bottom is the access connector.

#### Public Get Connection

A connection to an object, that required a Public Get access, looks like this:



The access connector stays visible, so that others can also connect to the object using Public Get.

#### Public Set Connection

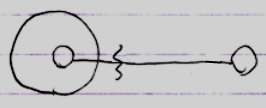
A connection to an object, that required a Public Set access, looks like this:



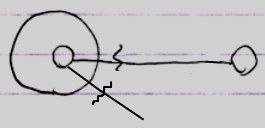
The access connector stays visible, so that others can also call Public Set and change the connection of the object.

#### Friend Connection

The general notation of a Friend connection is as follows:

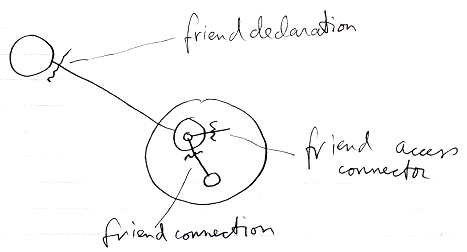


But remember that the *connector* always stays visible:



#### Friend Declaration, Connector and Connection

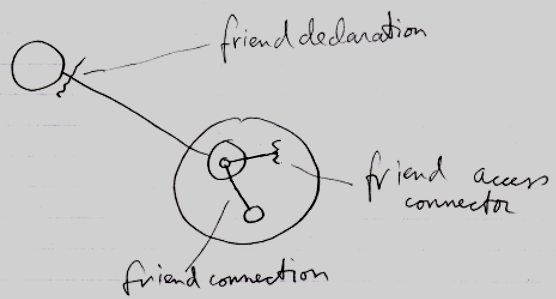
Here the difference is demonstrated between the notations of Friend declarations, Friend connectors and Friend connections. It will also become apparent how their notations disambiguate.



The friend *declaration* is always shown directly *near* the befriended (the situation at the top).

The friend access *connector* is always a loose end (the situation in the middle): one end of it is not connected yet. It is a placeholder for possible connections. A friend *connection* (the situation at the bottom) *is* connected at both ends. But the access symbol is placed in front the parent of the target symbol. This is because you always access an object through the parent symbol. The parent symbol can impose access restrictions onto its child objects.

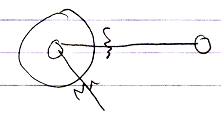
If the friend connection would be an outward connection, no friend access mark would be displayed at all:



The friend connection shown there is not really a *friend* connection either. It is just a *connection*, an outward *connection*, a *passive* connection. What a releaf that a passive connection does not require an access symbol at all, or it would have conflicted with our friend declaration notation, which looks a bit like an outward connection with an access mark displayed in it, but it is not an access mark; it is a friend declaration.

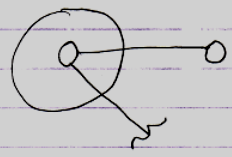
#### Friend Get Connection

The picture below shows a Friend connection to Private Get:



#### Friend Set Connection

The picture below shows a Friend connection, that required to Private Set access:



#### Loose Ideas about Public & Friend Connections

< Remembering, that the access connector stays visible has to be adressed in a single section and these explanations should not be bothered with that. >

### Access Controlling Aspects

The diagrams in the previous articles explained the concept of accessing Public or Private members. Those were all demonstrated using the Object aspect. But there are more aspects you can access, such as the Value aspect and the Class aspect. Those can also be made Public or Private.

The following sections lay out the workings and the notation of access controlling any of the system aspects. Connectors and connections of the following aspects are covered:

- Object

- Class

- Value

- Data

- Clone

- Execute

- Existance

- Add & Remove

Each aspect has its own unique behavior.

The Interface aspect is not covered here, because it is covered by the *Interfaces* article group.

### Object-Bound, Reference-Bound & Access Control

For the basics: see the article *Object-Bound & Reference-Bound* in the *System Objects* article group.

#### Key Example: Value Get

If the Value aspect were reference-bound and made read-only by a reference and you could still Get the Object, you could assign the Object to another reference and write the Value there. Read-only protecting the Value aspect would be non-waterproof and more like a *suggested* access control, rather than *real* access control. By realizing that the Value aspect is object-bound, you can not make a read-only Value aspect all of a sudden writable through another reference.

Reference-related aspects only count for a single reference and not for the whole object, and can be freely overridden by another reference. Object-bound aspects count for all references and can not be overridden by another reference.

By binding some aspects to the object and some aspects to a reference, each aspect can be safely access controlled without any loop-holes through other references.

#### Another Example: Use As Class

Use As Class is basically an Object Get, access-controlled for the specific purpose of using the object as a class. By making Use As Class object-bound it is possible to design the system so that it indeed is not possible to use the object as the source in a class assignment. So things like Use As Class can also be made safe now they are object-bound. When Object Get is Public, Use As Class can still be Private.

#### Object-Bound Access Control

Object-bound access control is visible through each reference and can be controlled through each reference, but object-bound aspects always apply to the whole object.

Object-bound aspects and reference bound-aspects do not require two separate friend declarations. With one friend declaration the Friend can access the object-bound aspects of the object, and access the reference-bound aspects of its sub-objects. It is like a combo-deal.

#### Reference can Access Control Object-Bound

Even though object-bound aspects are primarily access-controlled by the object, a reference can impose additional access control over them. So object-bound aspects can be access controlled by a parent to further protect what the object can already protect. But that is part of the *Deeper Exclusion* concept, explained elsewhere.

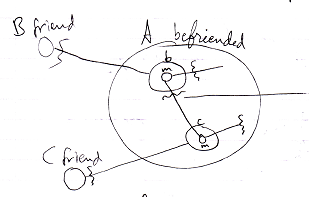
#### Object has Access to its Privates

The parent object automatically has friend access to its own object-bound aspects and to the reference-bound aspects of its sub-objects. That is logical: the object automatically has access to its own privates. The access connectors express potential connections that can be made by other objects and are displayed on the outside of the object.

(But compared to the traditional approach: when an object has a command (a method), the command must be made friend in order to access the privates of the object.)

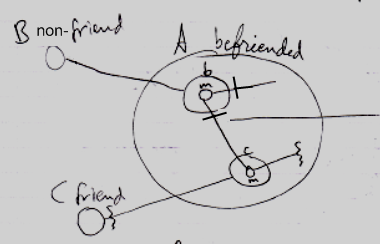
### Assignment between Friend Members

A connection or assignment always involves two objects. In a connection or assignment that involves a friend object, the access mark is a wavy line. It is only clear which of the objects are friend-accessed from the friend access *connectors*.



In the example above both members involved in the connection are friend. It is only clear which objects in the access are friend from the presence or absence of a friend access *connector*.

In the following example, one of the members is public while the other one is private.



Member m of reference **b** is Get-public. The access mark of the connection is changed to a Public access mark, because the Get-access is public, and an access mark expresses the Get-access. To see which member involved in a connection or assignment is friend, you should look at the *access connectors* and not necessarily at the access mark on the connection or assignment call.

### Friend = Wavy Line

But why the wavy line? The wavy line is already used within the language to express ‘value’, ‘data’, or ‘content’. Now it is also used to express *friendship*.

Compared to the other line types:

The wavy line expresses something special. Wavy is also like the symbol ~, which means approximate or ‘under certain conditions’. And under certain conditions you can access Private members, which are then denoted with a wavy access connector. So it looks *like* a straight access connector (meaning Accessible), but then under certain conditions, so the line is wavy. A wavy line expresses *special access*, *conditional access*.

Wavy lines already expressed *value*, *data* and *content*. Using wavy lines in the friend notation lets you access the Private *value, data and content*. So there is a vague parallel there.