|  |
| --- |
| Circle Language Spec: Inheritance |

## Enforcing & Preventing Specialization

The world of programming languages offers several ways to prevent *or* *enforce* specialization. This article will give this a place inside the new computer language.

The concepts from other languages can be boiled down to a simpler model that has to provide the following:

- Protected

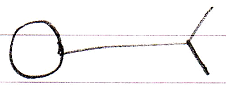
- Overriding

- Required / Optional

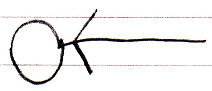
- Requirements for other side of connection

### Protected

The protected access connector makes sure that a member is not publicly available, is privately available, but also available to all references (triangles) that merge with their container.

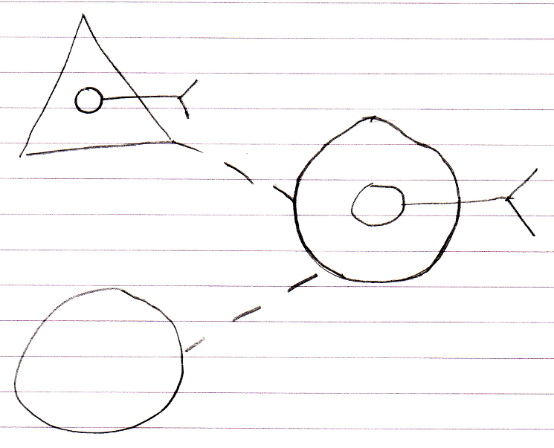


Protected Object Set

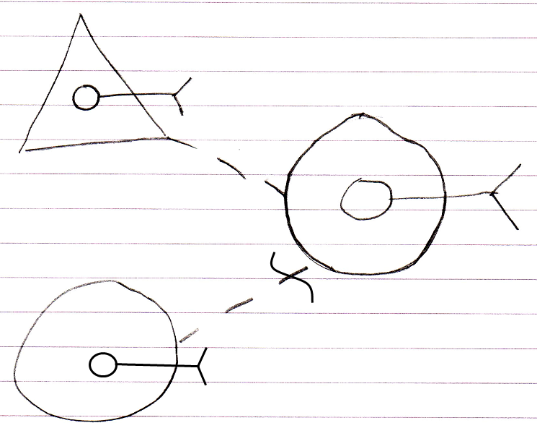


Protected Object Get

In other words: Protected will be accomplished by providing ‘public’ access through a triangle reference, but ‘private’ access through a circle reference. In the picture below the triangle offers access to the member, but the circle does not.



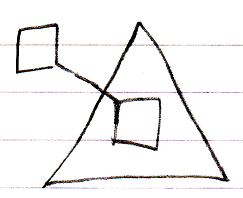
Note that Protected members are still privately accessible, which means that Friend objects can always access Protected members as well.



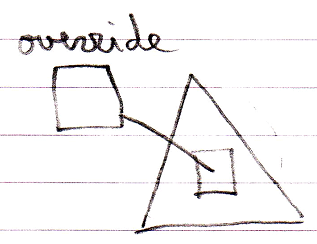
### Overriding

The overriding concept is already explained in the specialization article. There are two possible notations:

*Object reference notation:*

**

*Event notation:*

**

### Optional / Required

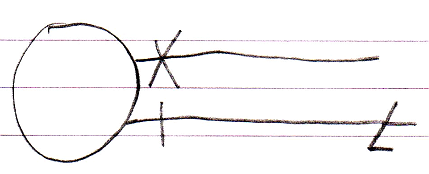
The concepts from other languages for preventing or enforcing specialization require imposing that something *must* be filled in. The concept of *required* is already worked out in the article *Optional & Required*. The notation looks as follows:

|  |
| --- |
|  |
|  |
| Optional |
|  |
|  |
|  |
|  |
| Required |

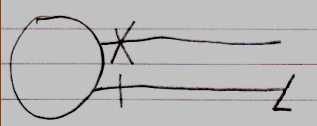
*Required* is expressed by putting half of the expected symbol at the end of the connector. *Optional* does not get such a symbol, because it is the default behavior that things are optional to fill in. Required is an extra rule imposed.

### Requirements for Other Side of Connection

The concepts from other languages for preventing or enforcing specialization require being able to specify that the other end of the connection needs to be a triangle. For now this is expressed as follows:



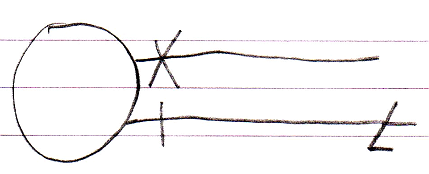
So it is a notation where the requirements for the other end of the connection are expressed at the other end of the connector. The connector line continues after the specified requirement as to not conflict with the notation for Required in general, like this:



### Gut Feeling

A gut feeling of mine is that Protected and requirements for other side of connection could be made more alike, because both are about how the other end must be a triangle, but in one case that the direct reference must be a triangle, and in the other case the parent must be a triangle. So these requirements for it being a triangle might have to be merged into one concept one day.

The notation come up with for requirements for the other side of the connection may be inspiration for a notation for a more elaborate security model one day.



### Applied to Concepts from Other Languages

The following keywords from other languages that prevent or enforce specialization are covered along with how they translate to the new computer language:

Protected

Virtual

Abstract member

Abstract class

Sealed / Final

Interface

Interface member

Non-Overridable (the default in other languages)

### Protected

For now Protected is an intrinsic part of the new computer language. It has its own type of access mark:

|  |
| --- |
|  |
|  |
| Protected |

### Virtual

The keyword virtual from other languages means that you *can override* the member. It is also called Overridable. In the new computer language this means that overriding should be Public. In the two notations for overriding this would look as follows:

|  |
| --- |
| *Object reference notation:* |
|  |
|  |
|  |
| Public Object Set |
|  |
|  |
| *Event notation:* |
|  |
|  |
|  |
| Public Object Override |

In some languages overriding is private by default, protecting a base class from specialization.

### Abstract Member

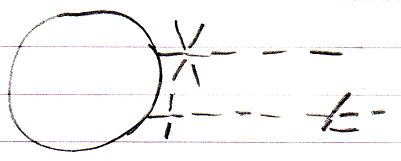
When the keyword abstract from other languages is applied to a member, it means that you *must* override the member. It is also called MustOverride. In the new computer language is a combination of making overriding Public *and* Required. In the two notations for overriding this would look as follows:

|  |
| --- |
| *Object reference notation:* |
|  |
|  |
|  |
| Public Required Object Set |
|  |
| *Event notation:* |
|  |
|  |
|  |
| Public Required Object Override |

### Abstract Class

When the keyword abstract from other languages is applied to a class, it means that you *must* inherit from the class, or better said: you cannot just reference the class, the reference must merge with its container. It is also called MustInherit.

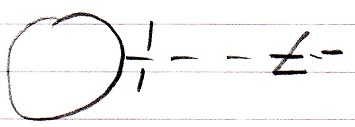
In the new computer language this gets the following preliminary notation:



Private Use As Class

Public Triangle Use As Class

Or perhaps simply:



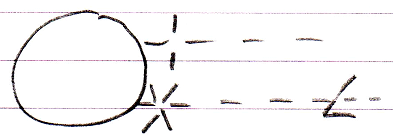
Public Triangle Use As Class

Do not get confused: abstract applied to a class does not mean that its commands are abstract, so must be overridden, it just means that the class must be inherited from. An abstract class *can* have method *implementations* in it.

### Sealed / Final Class

The keywords sealed and final mean the same thing. A sealed class, or final class, is the opposite of an abstract class: instead of only being able to inherit from the class, you are explicitly not able to inherit from the class. It is also called NotInheritable.

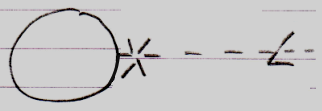
In the new computer language this gets the following preliminary notation:



Public Use As Class

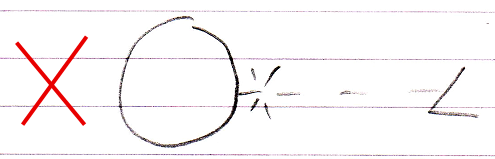
Private Triangle Use As Class

Or simply just:



Private Triangle Use As Class

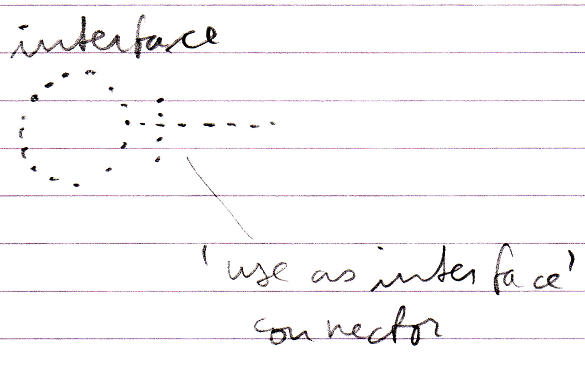
Here it becomes apparent that the notation should not be just an open half triangle, because this would imply that the connector is Required, which it explicitly is not.



The use of this enforcement is questionable to me, because with a little effort you could wrap a non-inheritable class into an inheritable class again. I can imagine that a final class could not have virtual or abstract members, so such a trick would not make it possible to override members, but it will make it possible to do any other inheritance trick in the book.

### Interface / Interface Member

Interfaces are an intrinsic concept inside the new computer language. In the new computer language any object can serve as another object’s interface by using the Use As Interface connector.



This is a bit different from other languages.

The differences are pointed out here, but it is questionable whether it is important to do anything about those differences. In other programming languages the interface construct imposes several extra restrictions compared to the class construct. An interface is:

- abstract / must be inherited from

- only has abstract members / members must be overridden

- cannot have an implementation

Normally you do not see these restrictions. They only come to light as soon as you break a rule. That is why the interface constructs seems like a simpler construct: you can do less with it, but you could also see it as a more complex one, because compared to the class construct, *more restrictions* are imposed and these restrictions are also define somewhere. You just do not see this in other languages.

#### Rule 1: Interfaces are Abstract

The first rule for interfaces in other languages is: interfaces are abstract and must be inherited from. This rule is questionable though, since some languages allow you to declare an *explicit* interface, which basically replaces the inheritance characteristic of the interface with aggregation characteristics. In the new computer language you can also choose whether to implement an interface in an implicit or explicit mannar / choose between inheritance and aggregation.

However if you want to enforce that you can only use the interface in an inheritance way, you could express that by imposing a rule for the other side of the connection:

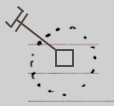


This expresses that the other end of the connection should be a triangle.

#### Rule 2: Interface Members are Abstract

The second rule of interfaces is: members must be overridden / members are abstract. In the new computer language all interface members are always present inside an implementation of an interface. This is not considered overriding: you have no say in whether to override something or not, the members are just automatically there as soon as you implement an interface.

However unnecessary, you can implicitly define that a member is abstract, which usually looks as follows:

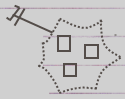


Meaning the *interface’s* *square* member has a Public Object Set access connector that is Required.

But if you want to do that, you would have to do it for all members of an interface.



If you want to set this rule for all members at the same time, there is no other notation for that, than to turn the interface into a list object, in which you can control the aspects of all list members at the same time:



But all in all, you should not worry about this, because if you use an interface, automatically all members of the interface are implemented.

#### Rule 3: interfaces Cannot have an Implementation

The new computer language offers no way to enforce that the interface object has no implementations. Period. You can put implementations in interfaces, only they will never be used.

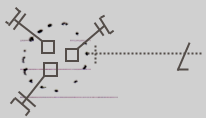
Something must be considered here compared to other programming languages. In other programming languages the implementation of an interface is composed of method implementations and private members. In the new computer language, method implementations are considered *private content*. So in the new computer language the interface and implementation are distinguished by separating the public content from the private content.

So how this translated to implementations inside interface objects: if you make something inside an interface object private, it will never be used.

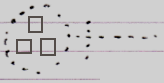
You can make the private content usable again by either making it public again or by making the object usable as a class or plainly as a object.

#### Gruesome

In the picture below both interface members being abstract and the interface being abstract are expressed:



This looks pretty gruesome for trying to define the interface construct, which should be simpler than the class construct, so please just use:



The access connector is not even required, so you can also notate:



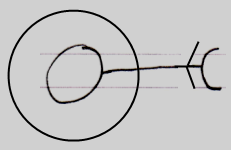
### Non-Overridable

Not being able to override is the default in other languages, but in the new computer language it is an imposed restriction, making overriding Private, which looks as follows in the two notations for overriding:

|  |
| --- |
| *Object reference notation:* |
|  |
|  |
|  |
| Private Object Set |
|  |
| *Event notation:* |
|  |
|  |
|  |
| Private Object Override |

### More Combinations

With the concepts explained so far you can form other combinations of protection against specialization. For instance the following:



Protected Required Object Set

But this is also possible in other languages by making a member protected abstract.

### Courtesy

All of this is just courtesy to support all these concepts in the new computer language. My gut feeling is that something simpler will be come up with in the future.

For instance the Protected connector notation was come up with at a very early stage, long before the other inheritance-related modifiers were considered. My gut feeling is that I have to let go of the notation for Protected and go for a notation more concise and integrated with all of the other things that can be expressed. Protected really does not deserve such an intrinsic notation compared to the rest of the inheritance-related modifiers.

This article is just put here to make sure that at least all of this stuff is possible in the new computer language, but my gut feeling is that a replacement for all of this must be found.

### Enforcing & Preventing Other Specialization Methods

The ideas above will provide other language’s capabilities to enforce or prevent specialization. But what about the other specialization methods? How are they enforced or prevented? The following specialization methods were not covered:

- Altering the Member Set:

- Member Addition

- Member Exclusion

- Member Inclusion

- Detouring Members:

- Shadowing

- Altering Command Implementations:

- Command Pre- and Post-Extension

- System Command Pre- and Post-Extension

- Overriding System Commands

- Shadowing System Commands

To get it in line with the rest of the system, the following methods for protection of enforcement of specialization should become applicable to each of the specialization methods:

- Protected

- Required / Optional

- Requirements for other side of connection (for it to be a triangle)

However, these other specialization methods do not really disturb the class’s original implementation. Anything that alters the base’s behavior should be preventable. The rest is not that important to prevent. The only specialization methods proposed now that change the actual base object is overriding and cancelling on pre-extension. The rest just augments an object. The work required to work out how all specialization methods are preventable is not done right now, because it is not considered very important or worth the hassle.

### Preventing Pre-Extension with Cancellation

Pre-extension with cancellation is an alternative to overriding. It should also be preventable, because it alters the base’s behavior. Clearly when something is not overridable, you should also not be able to cancel it. The exact implementation of this is not important. The only thing that is important is that it prevention of pre-extension with cancellation *will* be implemented.

### Preventing & Enforcing Data Replacement

Data replacement is not regarded a specialization technique. However, anything inside an object is data, even the code, command names and command parameters. By default any of this data can be replaced. The way to prevent data replacement is by either making something Private, so that only Friend objects can access it, or by making the data Static inside the class, so that the data cannot be changed through instances of the class.