|  |
| --- |
| Circle Language Spec: Interfaces |

## Interfaces Main Concepts

### Introduction

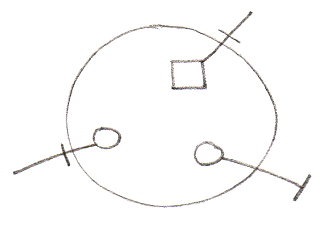
Up till now the idea of interfaces was fully left out of the stories. This chapter makes the concept of interfaces catch up with the rest of the documentation. The basic concept of interfaces is covered along with its diagram notation. Also covered are: interface assignment, interface reference and relations that form through the use of interfaces. Another concept, yet to be worked out, is how commands of an object are grouped by site: an object can have commands, defined on different sites, and the commands should be grouped by site in order to judge how reliable the commands are *(grouped by source)*. As interfaces can be applied to objects, but interfaces can also be applied to *commands*.

At some point the idea was, to also cover all the different *uses* of interfaces. But the different uses of interfaces are adequately described by other literature. The focus will lie on explaining the raw concept of interfaces and their presence in the new computer language.

The main concepts of interfaces are explained here along with their diagram notations.

### Interface = Publics

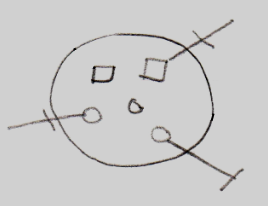
An object’s interface consists of all of its public members.



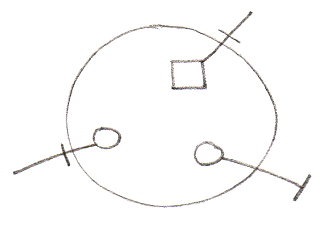
### Interface & Implementation

An interface defines an object’s public members. The *privates* of an object are called the object’s *implementation*. So the concept of interfaces offers the ability to separate publics from privates, in other words separate interface from implementation.

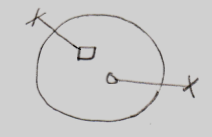
*Object:*



*Interface:*



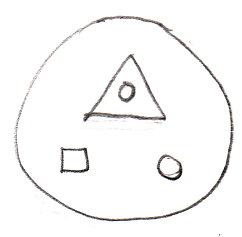
*Implementation:*



### Interface = Objects Melting Together

Another concept which employs the word *interfaces*, is an object (an interface) melting together with its container.

This is displayed in the diagram by a triangle that represents an interface object.

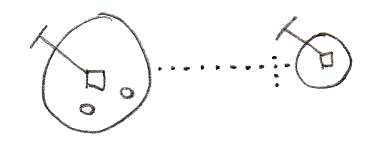


The members of the triangle are accessible as if they are members of the large circle.

### Interface = Contract

Interfaces also offer the functionality of being a contract.

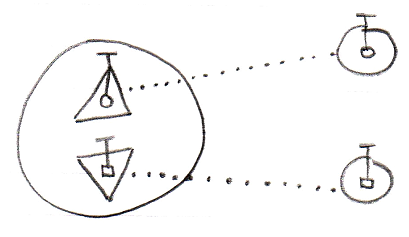
You can establish this by defining the interface externally.



If an object has its interface defined externally, the public contents of the object can not be freely changed. If the interface of the object needs to change, the externally defined interface needs to be changed.

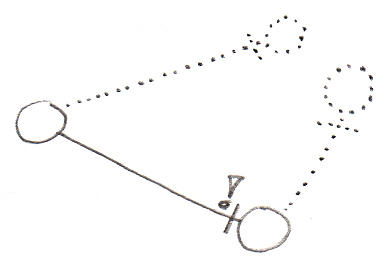
### Multiple Interfaces

If you want an object to support multiple contracts you can use several triangles inside the object and define their interfaces externally.



### Interface Type Control

Interface type control should for instance occur when you reference an object and the reference has the wrong interface. Restrictions must then be enforced.



The situation above is not valid, because the reference does not have the same interface as the object it points to.

### Explicit Interfaces

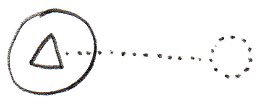
You can speak of an explicit interface when you externally define a sub-object’s interface, you but do not let the sub-object melt together with the container.



A reference to the parent can not automatically resolve to a reference to the interface. It requires you to refer to the interface directly in order to access it.

### Implicit Interfaces

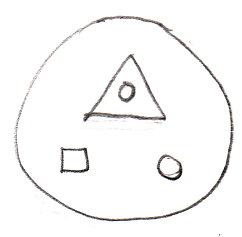
You can speak of an implicit interface when you externally define a sub-object’s interface, but you do let the sub-object melt together with the container.



You can reference members of the interface directly through the circle, without explicitly mentioning which interface you are accessing (see *Interface Object Resolution*).

### Polymorphism

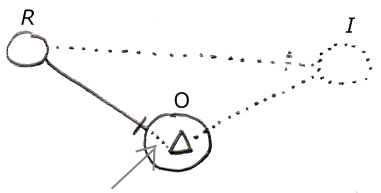
A triangle’s difference with a circle is that triangles basically melt together with their container.



Next to all members of the triangle being accessible as if they were members of the parent circle, the container can also be referred to as if it has the type of the triangle. That is another effect of melting the objects together.

### Interface Object Resolution

Interface object resolution happens when you reference an object and the reference has an interface defined, that the object supports. It automatically redirects the reference to the specific interface of the object.



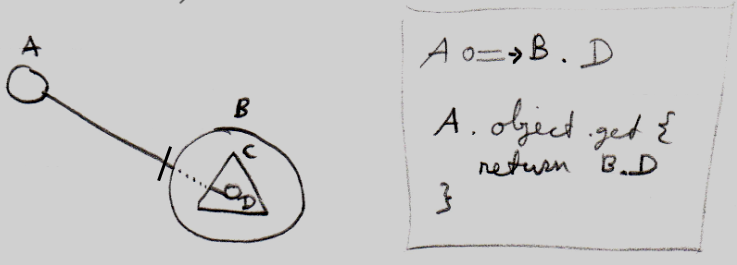
It is about the bit of dotted line denoted by the arrow. This is an automatic extension of the solid line on the left. Because the the reference (R) on the left has an externally defined interface (**I**) on the right, and the object (O) supports this interface, the reference’s line is automatically redirected to this particular interface of the object. Automatic redirections like that are called *object resolution*.

There are a lot of other object resolution concepts related to interfaces and inheritance, such as overriding and shadowing, but they will be covered separately in the *Object Resolution* chapter.

The situation above only works in case of implicit interfaces.

It also does not work when the same interface is defined twice. If it is, you have to refer to the interface explicitly.

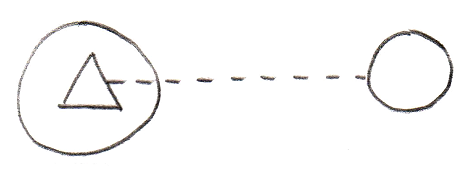
Below is another example of interface object resolution, but now in case of referencing a member of the interface, while not explicitly accessing the interface:



Again it is about the dotted piece of line in between, which expresses the automatic object resolution. Next to the picture is a bit of pseudo-code that expresses how reference A points to D without qualifying it with interface C.

### Inheritance

Traditional inheritance is related to the concept of interfaces. What happens is that an object melts together with its container and next to its interface being defined externally, the implementation is also defined externally. The result is classic inheritance: one object takes over all characteristics of another object.



So because the triangle has an externally defined class (dashed line) instead of an externally defined interface (dotted line) the situation turns from implicit interface into classic inheritance. In the situation above one object inherits from another object. Expressing it more like one class inheriting from another class it would look as follows:



Classic inheritance is not the only form of inheritance in the new computer language. Different forms of inheritance are covered separately in the chapter *Inheritance*, but it is important to point out here, that classic inheritance and interfaces are closely related.

### Main Usages of Contracts

When you externally define an interface, it becomes a contract. You can creatively use this concept for any purposes, but some of those purposes deserve to be pointed out explicitly.

It can be a contract of a *guarantee* to support a sure set of members. So this offers a guarantee to ones that will be using the object. So you know an object has *at least* those members with that exact configuration.

An interface can also serve as a contract of *usage*. This means, that an object has *exactly* the members defined by the interface. So now someone using the object will only be able to use it in an exactly defined way.

A powerful application of such a contract, is the interface as a contract of *participation*. You can give a type an already defined interface. That allows objects that can work with a certain interface, to link to objects of the new type, even when the system was not originally aware of that type. This concept makes it possible to introduce new types of objects into a system, that was not originally aware of them.

### Loose Ideas

*The texts below are loose ideas yet to be turned into good documentation.*

The interfaces an object supports are interesting. It’s also interesting to know if an interface is supported by an object. Of course you could derive that from the first data, but that’s not the fastest.

You should be able to group symbols inside a special symbol that controls the access for all the directly contained symbols. <2010-05-07 Just use a triangle and access control the triangle.>