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

## Inheritance

### Introduction

This documentation folder covers the concept of inheritance. This concept has been present inside object oriented programming languages for decades. But the new computer language takes it a little bit further. In other languages one class can inherit from another class, taking over all its characteristics before the derived class gets extended or altered. Inheritance goes further than that in the new computer language. Next to class inheritance (the traditional form of inheritance), there is also object inheritance. This means, that one object takes over the characteristics of a specific other object. In fact, the other object melts together with the new object. When you change base members of the new object, the base object’s data also changes. This is callled object inheritance. There are even more forms of inheritance. A collection can be inherited from, so that a whole existing table of items can be extended with new columns. This is called list inheritance. A whole system can also inherit from another system, taking over all of its characteristics, and the derived system can be altered, augmented, specialized and extended. This is called system inheritance. If the original system gets changed, this may also change the derived systems. So basically: everything can be inherited from and this can render specialized versions of any class, object, collection or module.

Such specialization can be also be applied to commands. The inheritance documentation also covers other concepts in the area of specialization and extension. Inheritance is very closely related to the concept of interfaces.

### Main Concepts

The new computer language supports several forms of inheritance. All these basic forms of inheritance will be laid out in the following articles:

*Class Inheritance*

*Object Inheritance*

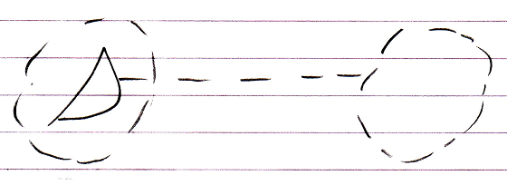
*List Inheritance*

*System Inheritance*

*Interface Inheritance*

### Class Inheritance

Class inheritance is the traditional form of inheritance: one class takes over all characteristics of another class. This is displayed as follows:



The second class merges with the first class. The first class has an interface that delegates its implementation to the second class.

### Object Inheritance

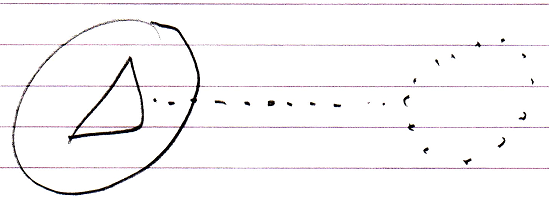
An object can merge with another object, delegating its interface, implementation *and* its values to another object:



So this time the interface has an object line. This means that as soon as the other object’s values change, the derived object is automatically also changed. The other way around is also true: when you change the derived object’s inherited members, the base object also changes. Changes are not *applied* from one to the other: both are just references to exactly the same object.

### Interface Inheritance

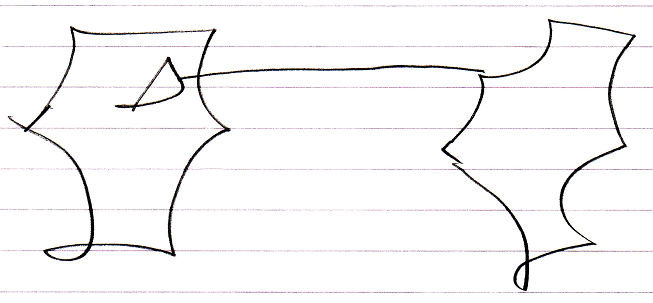
Since any redirection of a triangle is considered inheritance, interface redirection of a triangle may also be called inheritance: *interface inheritance*.



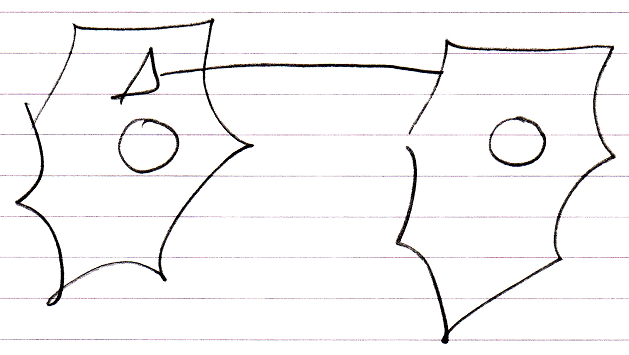
So the main concept of inheritance is basically the same as that of interfaces.

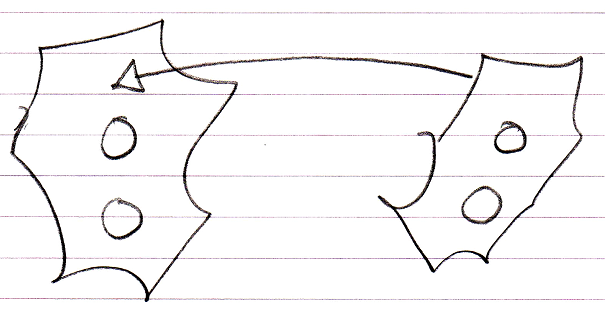
### List Inheritance

List inheritance is a special form of inheritance where one list inherits from another.

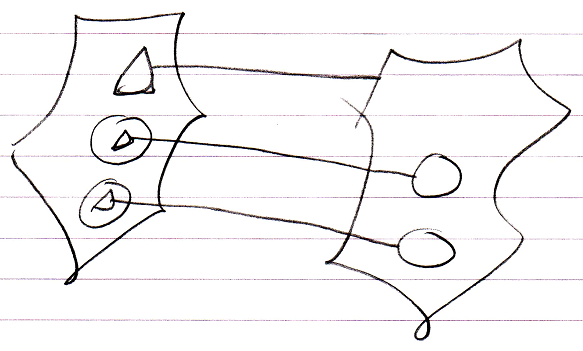


What is special about it, is that when an object is added to the base list, an object is also added to the derived list.





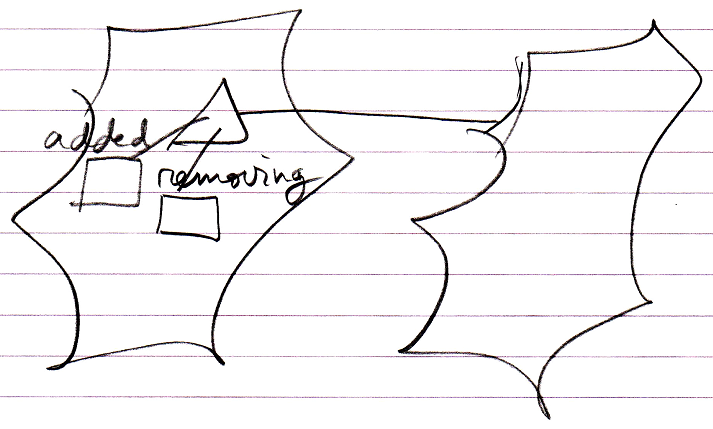
And the list item added to the derived list will inherit directly from the list item in the base list.



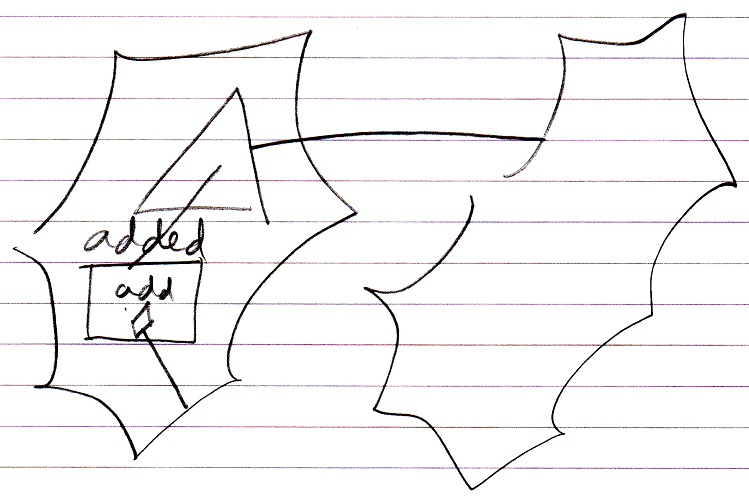
List inheritance is analogus to defining a *parallel list*, adding extra columns to an existing list.

This behavior is specific to list inheritance and does not apply to any of the other forms of inheritance.

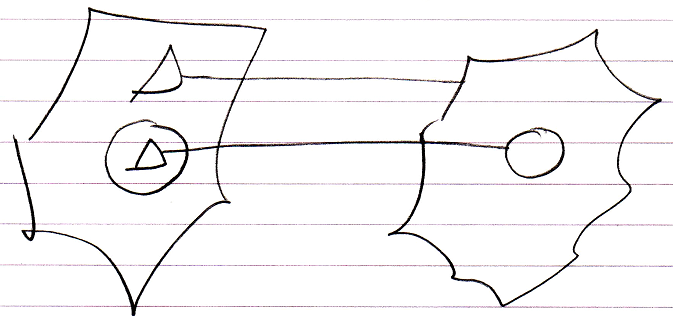
The list inheritance construct realizes such behavior by responding to the List events of the base list: the Added and Removing events. These events are *system events*, that were introduced by the *Events* chapter.



These events make sure that what happened in the base list, will also happen in the derived list.

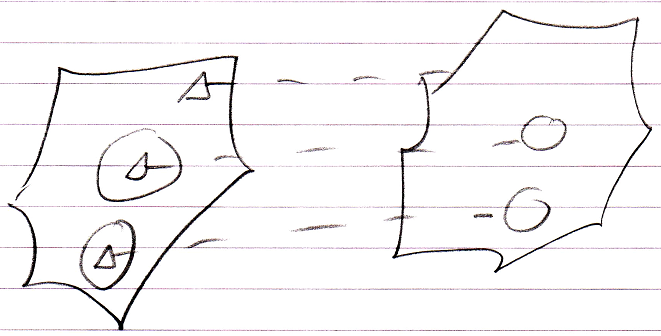


And on top of that, the derived list item will be linked to the base list item through inheritance.



List inheritance also makes a distinction between object inheritance and class inheritance. The default is object inheritance, which causes the behavior you are most likely to expect.

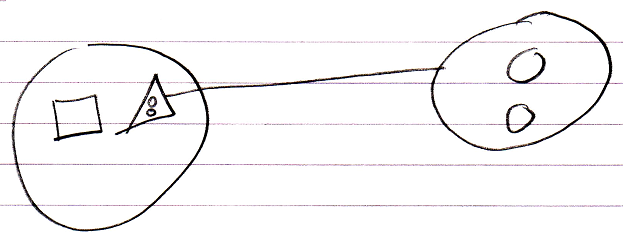
But *class* inheritance is also possible for lists, in case of which the lists items will be linked with class inheritance:



This behavior is not as intuitive as object list inheritance.

For an explanation about how different list objects themselves, such as linked list and stack, can inherit from eachother, see the article *List Concept*. The main idea is that you would have to open up the system interface for that.

The following might be something you could get confused about. A normal object is also a *list* of members. However one object’s inheriting from another object does not cause the same behavior as list inheritance: extra members are not added to the derived object when members are added to the base object. So do not get confused over this.



### System Inheritance

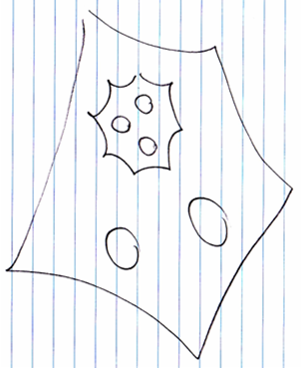
System inheritance introduces the concept of having one module with objects defined inside of it, and inheriting from that module, allowing the derived module to redefine any object defined in the base module.

You can be selective about which objects inside the derived module get specialized. This allows you to both specialize objects with a fixed logical residence inside the base module, as well as specialize objects that are mere references to objects defined totally elsewhere outside the module.

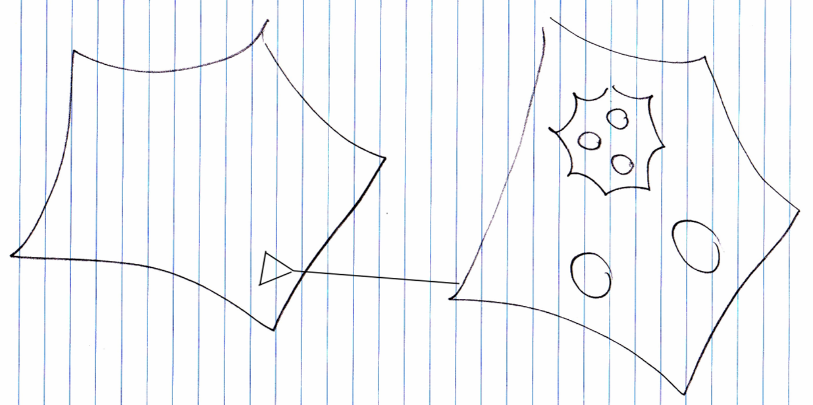
Objects in the derived module will inherit from their counterpart object in the base module. This will all go automatically.

Synonyms for system inheritance are module inheritance or a parallel structure.

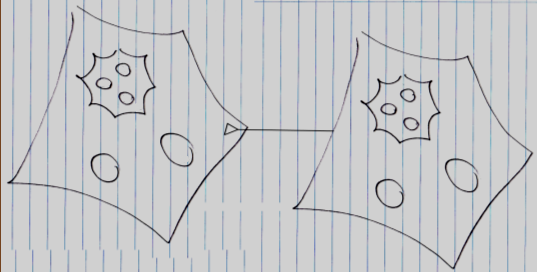
Here is an example system:



Here is another system inheriting from that system:



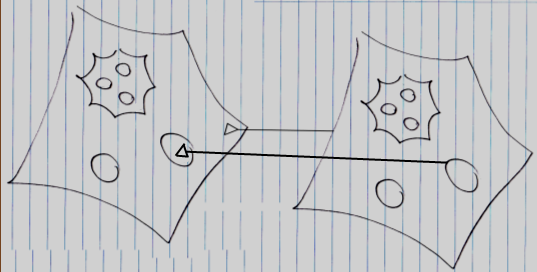
At first the derived system would just contain all objects from the base system



Already you can see a significant difference with normal inheritance: the derived members are not defined inside the triangle, but are defined outside the triangle.

At this point the exact same objects are present in both base system and derived system. This may have been pointed out in the diagram by lines connecting the references to the same objects together, but in the picture above this is left out of the picture.

In system inheritance you can simply pick out the sub-object you are going to specialize and create a derived object right then and there:

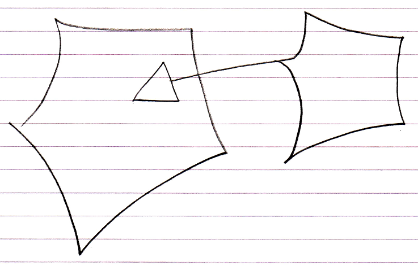


Now the derived object is not the same object as the object in the base system anymore, but the derived object is a new object, which has a triangular reference to the base object.

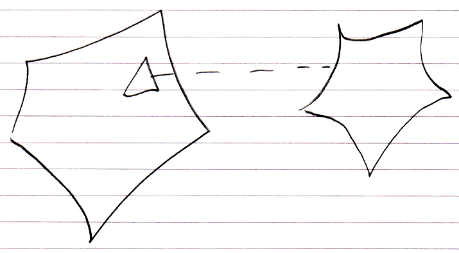
Something to consider though, is that when a much deeper object is specialized, all the levels that lead to the deeper object will also get a specialized version (for more information see: *Even Deeper Member Addition*).

Systems can inherit with either object inheritance or class inheritance.

*System object inheritance:*

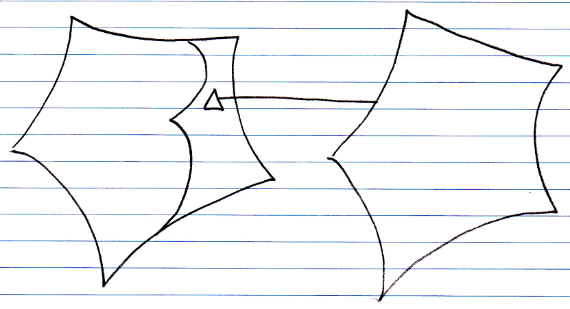


*System class inheritance:*



The difference between the two is completely analogus to normal object inheritance and class inheritance. In system *object* inheritance changing the values of base members will directly change the base system. You are working directly on the base system. In system *class* inheritance you can freely change the values of the base members without affecting the base system.

System inheritance is added behavior compared to normal inheritance, because any of the sub-objects can be redefined. Modules can also inherit from eachother like *normal* objects, but you have to do this using the system interface notation as follows:



You can then add or specialize members from the base system in a normal inheritance situation.

### Alternate Version Through Inheritance

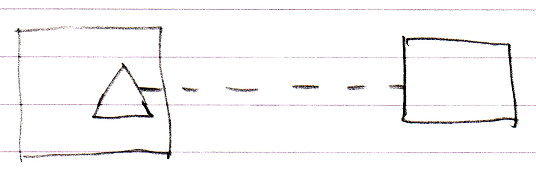
Once there was the idea, that through inheritance and specialization, you can create a *totally* altered version of a system, while the base system stays in tact. But this idea is let go of, because if something is private, then you can not change it, and when a specialization method is made private, then you can not specialize that. So you will not be able to create completely alternate versions of something through mere inheritance.

### Miscellaneous Inheritance Situations

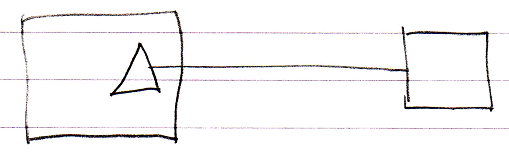
Now follow a couple of miscellaneous situations that can occur when you work with inheritance.

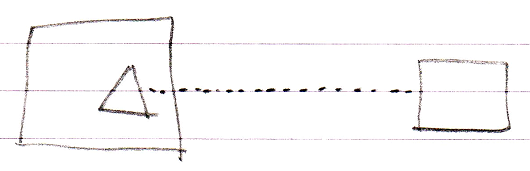
#### Command Inheritance

Because commands are just objects, inheritance can also be used between commands. One command can class-inherit from another command:

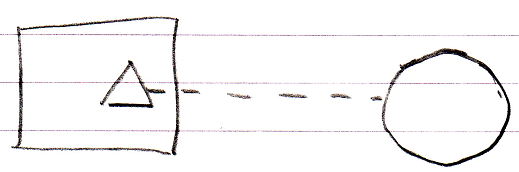


Object and interface inheritance are also possible for commands:



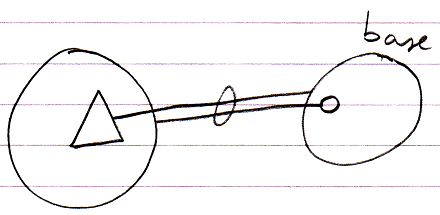


Since commands are just objects that can be executed, a command could in theory also inherit from an object:

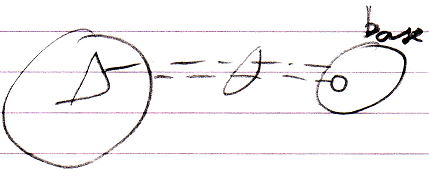


#### Backward Relation to Derived Classes

Due to the duality of relationships, when a derived class refers to its base class, the base class could also point back at the derived class again.

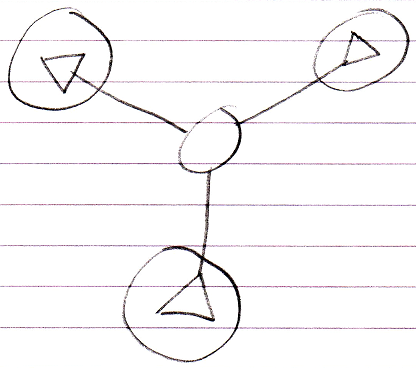


Or for class inheritance:



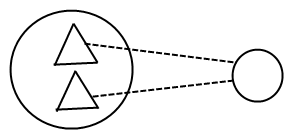
#### Multiple Objects Sharing One Base Object

Because multiple objects can inherit from the *same instance* of an object, you can have a set of objects use another object as their base:



#### Inheriting Multiply from the Same Class

Because you can give an object two sub-objects of the same class, you can also let a single object inherit from the class class twice:



But why would you do that? It does not matter, that’s up to you. You’re the programmer. You can reference the double members individually in the diagram, but when you want to reference a double member textually you have to disambiguate with a qualifier, which does require you to give each base its own name.