|  |
| --- |
| Circle Language Spec: Classes |

## Static

**[Preliminary documentation]**

This is unfinished documentation of lower quality. It only has an unfinished description of the general idea.

### Preliminary Design

*This brainstorm was written in the context of efforts to program a prototype app 'Circle 3'.*

The new version of the new computer language will work with the basics of objects, classes and interfaces. The distinction between objects, classes and interfaces is so loose in the new language and the distinction really leans on the concepts of public / private and the concept of static. These determine the difference between objects, classes and interfaces.

The concept of static is not completely worked out yet in the functional design.

There is a general brainstorm < read the brainstorm > but it has loose ends.

The first thing you have to know about classes and static structure in the new computer language is that what is traditionally part of the static structure, is not by default part of the static structure anymore. By default everything, including method definitions, are just data inside an object. In the new computer language you have to choose whether method definitions are part of the static structure or part of the object data. When it is part of the object data, this basically means that you can change this method definition for each object. If it part of the static structure, objects do not have their individual copy of the data and usually the objects can not change the data either.

So static parts of a class are the parts that can not vary between objects, it is the data that all objects share.

Static data inside the class will be like the class definition, while non-static data inside the class constitute mere default values for objects, but the values can be individually changed among objects.

Just like the public / private concept, you have to set the static flag for each individual access connector, in other words, each system command can be separately made static or dynamic.

Usually you will make static data read-only to the objects, so that the objects can not change it. But objects can also get write access to static data. This is static data that is shared among objects, and can be changed by all objects.

There is another type of static in other languages, which is data that can be accessed only through the class object, and not through instances. I do not like this type of static, because I would create two objects to solve this: one with instance methods and one with utility methods. If we must have a way to specify this in the new language, it would mean that each access connector can be made either accessible or inaccessible from instances.

The idea for now is that each system command will be separately made static or dynamic.

Class objects have full control over any piece of data. But when you use an object as a class, the object will only be able to control what the class object defines an object can control.

< Static read-write to the objects. How do you type-control this? I do not think this can be solved with just friend-public-private modifiers. >

< And what will be the notation for static? >

<

I wonder about making data static or not as opposed to making system coommands static or not. System commands give access to data, so depending on which system commands you make static or not, the underlying data is static or not. Actually, you have to control the staticness of the pieces of underlying data. For public / private it worked to control each individual system command, but for static you need to be aware of what underlying data you make static or not. Usually each aspect has its own underlying data. You can make each piece of underlying data static or dynamic. Furthermore you can make the getter or setter of that data accessible or inaccessible from the objects. So instance-accessible or not.  
> I have to to check what underlying data the aspects have.

So you can make an aspect *static* or *dynamic*.

And you can make a system command accessible or inaccessible for instances (but neat keywords must be come up with for that.)

> I also want to know how (in)accessible from instances relates to public/private.

Public/Private means (in)accessible to non-friends. It defines accessibility among objects.

>

Usable phrase:

Formerly it was fixed which types of construct were static and which were not, but now you can choose.

### Interface-Like Static

Is anything defined inside an interface static? I don’t know. Interfaces will be excluded from the first version.

### Important Brainstorm

2010-05-07

> The Relation concept now is worked out under the assumption that a class determines which other classes it is related to. But it is possible for a class to define which class it relates to and that an object of that class to change the related class.  
It is about Static. I am talking about my kind of static. Not the .NET or VB6 kind of static.  
If nothing inside a class is declared static you may as well not even have a class definition at all, because then everything defined in the class merely constitutes default values, because everything is still changeable inside an object. That is probably the prototype-based thing people talk about: nothing is actually fixed by the class. You just copy the whole class and then you have an object, in which you can still change everything, including the methods and the set of other members.  
To make a class definition worth something, you have to make some things static, which means, that it will be the same for every object of the class, it can not be changed inside any object of the class. Usually you make the command definitions static. But there are other types of static too. There is also a type of static where an object can change something of the class, but all objects share that data. There is also a type of static where something is only accessible through the class definition and not through an object (the kind of static that I do not like), and lastly there is the situation where something is not static, and each object just gets its own copy of the data.  
It is possible for a variable to be declared inside a class, the class aspect of which is static, but the value property of which is not static. So then the class of the variable is fixed, but objects can change the data of the variable and actually have their own copy of the variable. < Static ~= Private for Instances, except that there are also no separate copies of it for instances. > < This is not true. It changed. >  
I guess static where a class defines a variable, the value of which is changeable by the objects, but the object does not have its own copy of the variable, they share the same variable and value, is something defined static but Public to the objects.  
I am getting into the static stuff too much now.  
The thing is: in most object oriented languages it is fixed and unchangeable inside the language which constructs are which type of static, while in the new computer language everything is variable unless you fix it.  
I am mixing up the concepts of Static and Private here: both have to do with fixing something, limiting access to something, but I fail to understand which is which. That is because I have not worked out the concept of Static yet.

> One assumption specifically is wrong: that the class always defines certain constructs, that an object can not change. This also counts for interfaces, which are basically the same as classes but only for the public part of an object.

### Ideas

Fixed logical residence (in automatic containemtn) gives you a clue about static members of classes. That is also a special form of access, like global access.

In the For loop, the Loop procedure reference parameter has a Variable filled in, that is like static to the procedure definition. How does that work?

- Objects set for a command definition or class, that are taken over by all calls or objects. The reference target can not be overwritten by the call or object, but the value can be written or read. It is all about which system members of the object of a definition or class are declared static. The static system members are changeable by the class or definition, but cannot be changed by objects or calls.

> Once again another concept, that required better understanding of the system interface.

Static command data,

The data inside a command definition object serves as default values for the command parameters. The data inside a command definition also might be statically created, which means that calls, instead of creating a copy of the data of the command, refer directly to the data inside the command definition object. A command definition object can be assigned another command definition object, so command object assignment only works for assigning a command object to a definition.

JJ

Static,

2008-06

A local variable may make the method a static method of a class, instead of a method of an object?

JJ

Classes,

Static

Parameters, class redirections & calls,

2008-09-16

Als je een class relatie hebt tussen een command definition en een class,

dan krijgt de class een inactive command erin met een class line naar de command definition.

Als je een object creeert van de class, dan heeft die ook een inactive command erin,

met een class line naar de command definition. Dus dan heb je al twee kopieen van

de oorspronkelijke command definition.

De command in het object blijft inactive. Als je een command runt, dan ...

JJ

Static,

2008-10-01

You HAVE TO give class relations different behavior compared to object relations.

You HAVE TO give class relations specialized behavior.

Class relations are covered by the Class concept.

The Class concept will establish this specialized behavior.

As such, you can let Class relations behave different from

just an under the surface relation to an object, that serves as a class.

As such, you can let object instantiation from a class refrain from

duplicating shared code of a class.

This should be sort of like static stuff. You should have a separation

between static code and static data.

Only for caching purposes static code and data can be duplicated.

Caching is unaviodable in a network and persistance situation.

It looks like Static is a topic, that belongs to the documentation about Classses.

JJ

Static,

2009-06-17

There are two forms of static: static parts of a class, that will become part of all objects of that class and static parts of a class, that can not be called through the objects, but only directly by referencing the class.

JJ

### Out of the original Symbol documenation

< I need a story about where the code resides and that the code isn’t really part of module instances, types, objects or procedures. Those entities only complete the code with a copy of data. >

**Static and Creation**

<An object doesn’t define code. You could say that the type defines code and that a object only defines data. Each module instance, though, has its own instance of the types of the modules, even though the code of the types is still only in memory once. So you can’t really say the code resides in a type either.>

* Type static
* Procedure static
* Module static
* Type static objects
* Procedure static objects
* Module static objects
* Procedure static procedures
* Type static procedures
* Module static procedures
* Type Create
* Procedure Create
* Module Type Create

***Static***

Members usually belong to an object: every object has its own sub objects and procedures. Members of a type that are Static, however, belong to the type rather to individual objects. These members have the same configuration for every instance of the type. A change made to one object of the type changes all objects of the type. The members are considered *static* to the type.

Procedure instances are made on each call to the procedure. A call is a procedure instance. In order to make objects part of the whole procedure, rather than just the call, procedure members can be made Static to the procedure. On each call to the procedure, the static members retain their configuration.

Static members of a procedure are only the same for every call to the procedure. Static members of a procedure are different for every object that holds the procedure. In order to make procedure members static for the whole type of objects you have to make a procedure member Type Static. When I talk about static procedure members I’m talking about Procedure Static members.

Static type members are the same for every object instance. However, each module instance has its own copy of static type members. Static type members are not the same for every module instance. In order to make a type member, or a procedure member static to the any module instance, you make the member Module Static.

An effect of making type members static is that you can call the members even when the symbol isn’t created. Non-static members, or *instance members*, can only be called on a created object. Module static members can be called even when the module instance isn’t created, but the module type is loaded into memory. Static members of a procedure can be accessed without running the procedure.

Procedure static members are like part of their object. They persist for as long as the object is created. Type static members persist for as long as its module instance is created. Note here that each module instance gets its own set of types with static members. Module static members persist for as long as the module type exist. A module type exists when the Sigma module that contains it is in memory.

***Type Creation***

The Object Create procedure initializes the instance. In it you can create sub objects, set up lines and call members. Type members are initialized in another procedure: Type Create. The Type Create procedure of a symbol is called when the type is created. Types are created when a module instance is created so the Type Create procedure of all types in a module instances are called right after the module’s Object Create procedure.

You can only call static type members in the Type Create procedure, because non-static, *object* members can not be accessed yet, because they can only be accessed through a created object.

A procedure has a Type Create system procedure as well, to set up the procedure’s static members.

Members of a procedure that are Type Static are initialized in the type’s Type Create procedure. Any Type Static member, be it of a procedure lives as long as its type. Type Static procedure members are like type static type members, except that they are accessed through the procedure. If the procedure declares them private, they are accessible only to the procedure itself.

A module also have a Type Create procedure. This is called when the module type comes to light. Module types come to light when a Sigma module is loaded. All the Type Creates of module types in a Sigma module are called right after the Sigma module is loaded. The Type Create of the main module of a Sigma module serves as the module initializer. The Object Create of the main module serves as the module instance initializer.

***Old***

**Static Objects**

Type Static Objects

Sub objects belong to their container object. Static objects, however, are part of the type rather than part of individual objects. Each object then contains a sub object that is the same for each object of the type. The static object symbols are shown to all point to the object in the target type. The static object in the target type can be created while the type symbol isn’t shown as created. The target type holds the line targets and data of the static object. When an object of the type tries to change the object line of the static object, it doesn’t change the line of its own object symbol, because that line will always point to the static object of the target type. Instead, the line of the static object of the type changes.

This makes it so that lines can be tied together quite complexly, because static object symbols in object of the type always point to the static object symbol in the target type, which can be lines that travel far. On the other hand: every object of the type with a static object can set the line of the static object, meaning that the static object can point to a wide variety of locations, so it could again be a far away line. The lines of the static object symbol inside objects of the type, that point to the static object symbol of the target type, anyway, these lines don’t really exist. I still must come up with a way to make the notation different so that you can see the static systematics.

Procedure Static Objects

Procedures are much like types. They can hold objects too. If you want the state of an object to persist for multiple procedure calls, you can make the procedure object a *Procedure Static*.

A procedure static object does give every object of its type its own procedure static object.

By making a procedure’s object both Procedure Static *and* Type Static, the procedure object is the same for every call to every object of the type.

Procedure and Type Static objects inside procedures can also have the long lines going from all objects to the type.

<P>

Module Static Objects

A Type Static sub object is the same instance for each object of the type. But each module instance gets its own Type Static object. If you want the object to be the same for every instance of the module, then you must make the object Module Static.

Note that Type Static Objects that are direct ancestors of the module are static to the module type so automatically Module Static.

Note that Module Static objects are automatically also Type Static.

**Static Procedures**

Procedure Static Procedures

If you not just make a single procedure object Procedure Static, but the whole procedure Procedure Static, then all objects in the procedure are Procedure Static.

Type Static Procedures

If you make a procedure, not an object, Type Static, then the procedure static objects inside the procedure are also static for the whole type.

Another effect of making a procedure Type Static is that you can call the procedure even through an object that isn’t created.

Type Static objects and procedures can be accessed even though an uncreated object symbol, but non Type Static procedures and objects can *not* be accessed unless the object symbol is created.

Module Static Procedures

The rules of Type Static procedures also apply to Module Static procedures. Only Type Static procedures’ objects are static to *all instances* of the module.

**Type Create and Destroy**

Type Create and Destroy

In order to initialize and terminate objects and procedures static to the type, object symbols have a Type Create and a Type Destroy system procedure. The Type Create procedure of an object symbol is called long before the object symbol is created and becomes a created object. The Type Create procedure is called when a module object is created. The Type Create procedures of any ancestor object symbol of the module is called, but not of the object symbols in sub modules. Those object symbols’ Type Create procedures are called only when the sub module is created. Many object symbols might not even have a Type Create procedure.

The Type Destroy procedure is called when the module instance is Destroyed. Beware not to uninitialize Module Static members of the type, because those belong to all instances of the module, not just the instance of the module that’s being destroyed.

Note that you can only initialize the members static to the type in the Type Create procedure, not instance members. This is not a restriction that the Type Create procedure imposes. It’s just that the Type Create procedure is called before any object is created and you could only access non static members through a created object, so at the time the Type Create procedure is run, instance objects and procedures are accessible.

Type Create is stuck right at the end of the module’s Object Create.

Type Static members are like part of the module instance, rather than the object instance.

Module Type Static members are like part of the Sigma module, rather than the module instance.

Module Type Create and Destroy

Module symbols also have a Type Create. New module types come to light when a Sigma module is loaded. All the Type Creates of module types in a Sigma module are called right after the Sigma module is loaded. The Type Create of the main module of a Sigma module serves as the module initializer. The Object Create of the main module serves as the module instance initializer.

Procedure Create and Destroy

Procedures are in many aspects the same as types. To initialize the static members of procedures. Procedure Create is called on creation of the procedure’s object. In the Procedure Create you can only call static members of the procedure, because the instance members of procedure only exist during a call to the procedure.

Procedure Create is stuck right at the end of the object’s Object Create.

Procedure Static members of a procedure are like part of the object instance, rather than the procedure instance.

***Question***

Where are type static procedure members initialized?

Type Static procedure members are like part of the module instance. They are module instance *specific*, but this is also called type specific, because a type belongs to a module instance. They should be initialized along with the module instance, so along with the type creation. Type Create can access Type Static procedure members and are initialized there.

Type Static members of both the type and its procedures are initialized in the Type Create procedure and terminated in the Type Destroy procedure.

Module Static members of both module, its type and procedures are initialized in the module’s Type Create procedure and terminiated in its Type Destroy procedure.

***Old***

**Type Static**

The basic things about a static procedure:

* + Callable even when the object symbol isn’t created
  + Data mutual for every object

The first point is just a matter of tagging it static.

The second point:

The type should contain objects that store data. How do they belong to the type rather than the individual objects? Make all system procedures static? What does that render? That makes the procedures accessible when the descendant isn’t created.

Aleen als de naar hetzelfde geheugen verwijst.

Ik volg het niet. Ik zie het niet.

Static Members

There is a way to call procedures of a symbol that isn’t created: you can make procedure’s static. This makes them callable even when the symbol is not created. They are the type’s members rather than the object’s members. An object symbol becomes static by making one or more of its system procedures static, for instance the Symbol Get, which makes you able to link to an object symbol. You can call the type’s static members through any symbol that has that type. Static procedures aren’t passed the hidden object argument.

Type Create and Type Destroy

You may want static objects to always be created. Remember that static objects are objects some of whose system procedures are static. Those static objects can store static data for the type. Or you may want to reserve ‘static’ memory space for the type yourself. There’s no such thing as static memory space, but you can allocate memory and store a reference to it in a static object. To be able to have sub objects created as soon as the *type* is running, a type, apart from an object, needs a Create procedures. Therefore, object symbols have a Type Create procedure. Object symbols also have a Type Destroy procedure to be able to destroy static objects and deallocate static memory.

Type Create and Type Destroy are Static nor Non Static

The Type Create and Type Destroy are considered static nor non static, because because Type Create is called before the type is created, so before static members can be called.

**Module Static**

Other Issues

Modules need to be created too.

Modules can also have static members. Those are members of the module that can be called even when the module is not created. The module type create procedure of all sub modules is called as soon as the module is loaded in Sigma.

Types in a non created module can usually not be referred to.

There can also be static members of a module. Those are mutual to all module instances and can be referred to even when the module isn’t created. Then you can refer to procedures (and objects and types) of a non created module. The procedures must be made Module Static in order to do that. Module Static symbols’ Type Create procedures are called even when the module isn’t created. To have objects run as soon as the module type runs, the module symbol has a Type Create and Type Destroy procedure as well, in which you can create objects to run staticly.

Types of modules can be

Then module static type creates should be called when the sigma module loads. Right. The module only exists when. So modules must also have their own (Module) Type Create and Type Destroy procedures that are runned when the Sigma module is loaded, respectively unloaded.

So now there’s not only static symbols, but module static symbols, which can be members of ancestors at any depth inside the module. Only module static members (including system procedures) can be called when a module is not created.

Sub module’s type creates are also called on module load. If you don’t define module type create of syb modules, then nothing is called ofcouse. Only if you choose to have a module with globally static data, their module type create procedure is called.

Sub module’s type creates are also called on module load. Sub module type creates are called in the super module type create. In a module type, Type Create procedures that are declared Globally Static of object symbols are called when the the module type is created.

The creation of a module must be initiated by code, because a module symbol can also become a module reference, in case of which Create must not be called.

Dus globally static procedures kunnen worden aangeroepen nadat module is geload. Alle globally static procedures van alle.

* Globally static procedures
* Type Create procedure of modules
* Static procedures
* Type Create procedures of object symbols
* Object Create procedures of modules
* Object Create procedures of object symbols
* Static procedures directly in modules

Symbols always contain some data: their line targets. Those are static data, so when you create a module there’s always memory reserved: for each symbol memory that stores the line targets. (by the way: doesn’t a flag need to be stored to on if the symbol is created?). Or should I store line targets in the Core’s object table? Hmmm…

Static bij procedure members.

< How can you make a type: module static? >

**Other ideas**

<Also discuss the way to call a type’s members directly, while they are ‘object static’ (I haven’t introduced the term object static yet, I used to call it non-static, but object static seems to fit well in the system). So, discuss calling a type’s object static members directly, filling the *this* argument with a reference to the object. >

If one diamond’s internal lines change, lines of all procedure symbols pointing to the diamond change too. If a static procedure line is changed, all procedure symbols, also all calls with that definition will change.

< You have to mention here somewhere what the normal kind of static is and that when explaining situations, you are always seeing members in their normal type of static, and the other types of static are not considered. That way in other explanations lateron, I don’t have to add to every conclusion ‘unless it is static’>

Static,

2004

<

a symbol apart from an object access procedure also has a class access procedure giving you access to the class members in a static form. The object includes its type-static members…(I’m explaining static here)

>

JJ

Static & Classes,

2010-05-18

The class defines the *characteristics* of an object. This is a vague way to put it, because the distinction between the data that is part of the class and the data that is part of the object is arbitrarily chosen by the programmer. But since the class data does not change, the

JJ