Name, Space and Namespace

By: McTwist
February 21, 2018

Contents

1 Introduction

Functions may be a great asset and objects as well. But as mentioned in previous lesson, objects have methods, and here is where it will take its point. The document will also bring up names and namespaces for objects.

One would be recommended to start Blockland and singleplayer to be able to follow these instructions a bit easier.

2 Unique Identification

2.1 Object ID

First comes the object system in Torque Engine. If one already have noticed, an object in TorqueScript is really a number, or also called identifier(ID for short). That means that any number that is on the left side of the dot will be evaluated as an object.

```
1 $obj = new ScriptObject(); // $obj now contains a number
2 $obj.getID().getName(); // Calls getName method on object
```

`$obj` will call `getID`, which will call `getName` in return. As when the object is created, it will actually return a number to be used to identify the object. As the number is a value in a variable, then it could easily be modified as such. Keep in mind that one should never do such a thing.

As objects are normally stored within variables, one should rarely have to use the object ID's to be able to modify them. However, Blockland actually uses this to be able to get real objects within the world by just looking at them. Go up to an object and look at it. Then type in /getid and in the chat there will be an id, class name and a distance. Using that info in the console for the server and one could easily modify it.

2.2 Names

Of course, when scripting, this is generally not needed and the above mentioned approach is only used for debugging purposes. Therefore TorqueScript is able to name its objects.

```
1 new ScriptObject(MyObject);
2 MyObject.getID();
```

Notice that there is no variable that stores the object this time. That is because that there is first a name set for the object: `MyObject`. This could then be used to access the

object. One could also set the name on the object with the method `setName`. Keep in mind that unless from the previous ID system, there is no one stopping having several objects with the same name. It will simply take the first object it can find. **It does not access all of the objects with the same name, only one of them.**

It is fully possible to send in a variable value in the first parameter to set the name.

Names are case insensitive. String comparisons is not.

2.3 Strings

One might wonder why strings suddenly come up here, but that is because it is important to know about a special usage one need to know about them. And that is that these examples are working exactly the same.

```
1 $a = "James";
2 $b = James;
```

That is, this is valid syntax and both *aand*b contain the same string. This is because TorqueScript allows for strings to be created if they either: is not syntax; does only contain characters that can be used within variables. This is also reversible.

```
1 new ScriptObject("MyObject");
2 $obj = MyObject;
3 $obj = $obj.getID();
4 "MyObject".getID();
```

As previously mentioned, variables in TorqueScript is either number or strings. But in fact, it's almost always strings, unless otherwise specified. That is, when a normal arithmetic operation is performed, then it will convert the value to a number, if it is not already. In this case, as the name might not contain quotation marks, it will still be converted to a string. It only needs to follow the set rules for a variable name: the first character can only be any character from the latin alphabet or an underscore; contain numbers. If these rules are not followed, then one need to use quotation marks. Keep in mind that this should only be used for object names. Not to create a string.

3 Object Initializer

There is actually one more way to create an object and all its variables.

```
1 $obj = new ScriptObject()
2 {
3     name = "James";
```

```
4     age = 56;
5 };
```

This will make it easier to initialize objects without having to refer to the object itself.

4 Namespaces

The most used functionality within TorqueScript is *namespaces*. It is a way to box in parts of a code and then link it to an object. One then uses the operator of **namespace** (::), which is two colons. The name on the left is the name of the namespace and the one on the right is the function. This concatenation is what previous mentioned is called *methods*.

```
1 function ScriptObject::set(%obj, %value)
2 {
3 %obj.value = %value;
4 }
```

This could easily be called like a regular function, as the namespace operator is part of the name. However, as `ScriptObject` already exists as a name, then this method is now added to the object.

```
1 $obj = new ScriptObject();
2 $obj.set(4);
```

It will try to call the method of set, which is part of the namespace of `ScriptObject`. The first variable of all the methods always contains the value that was used to call the method. That means that if a value of an object ID is sent in, the variable will contain that ID; if it actually a name, then that will be used. One could of course call it like a normal function.

```
1 ScriptObject::set($obj, 4);
```

4.1 Variables

One quick thing that is useful to know when it comes to namespaces and global variables, is that namespaces can be used with them. One just simply separate sections of a variable with the operator and then accesses it like so.

```
1 $Pref::OneMod::value = 4;
```

There is some rules around them, and some is a bit complicated to bring up here. But as a rule of thumb, is to use the normal variable convention for each section.

4.2 Classes

Of course, one should try to avoid adding methods to existing objects as that will easily clog it. Therefore one could use classes, which basically a mix of names and namespaces. First off, a new unique namespace is created whenever a new name is used left of the operator.

```
1 function OneClass::set(%obj, %value)
2 {
3     %obj.value = %value;
4 }
```

Here is OneClass declared as a new namespace that could then be used to identify it, like ScriptObject was used. If one put it as a name for a new object, it creates a new powerful tool.

```
1 $obj = new ScriptObject(OneClass);
2 $obj.set(4);
```

This will link the object to both ScriptObject and the new namespace of OneClass. As this might look bothersome as one now forces the object to use an object name, and therefore will make that object reachable from everywhere by using that name. In comes a special member variable that will make use of even more namespaces and limit the object to be local.

```
1 $obj = new ScriptObject()
2 {
3     class = OneClass;
4 };
5 $obj.set(4);
```

This variable will do exactly the same thing, but will leave the object unnamed. On top of this, if one needs to add more than one namespace, there is a second variable named `superclass`. This brings up the complicated process of *packaging* and *parents*.

4.3 Parents

Now is where most struggles and therefore end up with horrible code. This is the most important part of TorqueScript and it is really powerful. There is two aspects that one need to think about. The first is the process to *override* a function. That means that

if one wrote a second function of `OneClass::set`, it will actually remove the previous declaration and replace it with the new one. In some cases, one could use this to replace previous declared functionality, but then one needs to rewrite the previous functionality if one still wants that to exist. In comes *overload*, which will wrap around the previous declaration, and then call it in the way needed. It is easily described with an example.

```
1 function TwoClass::set(%obj, %value)
2 {
3     Parent::set(%obj, %value + 2);
4 }
```

This will create the method of the same name but different namespace. When it is called, it will call on the *parent* method that this object has. These are called in a specific order, testing all of the ways that is possible: Name, class, superclass and lastly the head object.

```
1 $obj = new ScriptObject()
2 {
3     class = TwoClass;
4     superclass = OneClass;
5 };
6 $obj.set(4);
7 $obj.value; // 6
```

As previously mentioned, in this example the first one called is `TwoClass::set` and then `OneClass::set` is called from there as its parent. This could easily get confusing, but playing around with it will make it easier and eventually one might wrap around it.

Parents also works for normal functions. Keep in mind that one normally does not use parent this way, the biggest usage is in next section.

4.4 Packages

On this arises a question: What if one would like to overload an existing function instead of a method? Reusing the same name as the function will just override the previous declaration, rendering the whole process useless. Here is where *packages* is the best tool to use. One create a package by using the syntax `package` along with a name for it. Then scope around the functions that one want to overload.

```
1 function print(%text)
2 {
3     echo(%text);
4 }
5
```

```
 6 package OnePackage
 7 {
 8     Function print(%text)
 9     {
10         Parent::print("Hello" SPC %text);
11     }
12 };
```

Then one requires to activate the package. This is a way to easily deactivate functionality that will not be used in some parts of the code. An example is a client Add-On only used when online at a server.

```
1 activatePackage(OnePackage);
```

And then use it normally.

```
1 print("James");
```

This prints "Hello James" in the console.

Of course, this could also be used for methods.

5 Assignment

The best way of learning is by doing. That is especially true when programming. This assignment is a big one. Mostly because you already should know most of the things about how to code in TorqueScript.

It is split up in two parts. The first one is reading comprehension. That means that you will first read a piece of code and understand how it works. First of all, this code is used only in a client. Secondly, it is about mapping keyboard buttons to functions, which will be called when pressed down and up. Read and understand what it does. It is easier than it looks.

https://gist.github.com/McTwist/8bdd572fe31663d4a73ea534b5e261cd

The second part is about taking what you know and put it to the test. You basically will take the previous function and split it into several methods. One could then use that in an object to add, remove and check binds. Extra points if you manage to be able to sort the divisions.

This assignment will take at most one month to complete.