The Syntax of Functions

Functions are the heart of macros, functions are listed here and - much more so - in the API documentation. But how should we read something like this:

```
Void Group.StoreMenu.StoreOnButton(String group, Int32 index, AcwUserNumber userNumber)
```

Essentially functions are the part of the code where the system is instructed to do something.

- the function has a name, and most likely this name already describes (very briefly) what this function does. It is reasonable to assume that a function named `delete()` deletes something
- in our situation with hundreds of functions it's very helpful that functions are classified in namespaces. Thus you immediately see the difference between `playbacks.delete()` - which probably deletes a playback - and `fixtures.delete()` - which probably deletes a fixture.
- by convention, each function is denoted by a pair of brackets, even if there is nothing in them. `go()` is a function whereas go is something else, but not a function.
- most functions accept parameters (a.k.a. arguments) which tell the function what exactly to do, or with what to do it. In the above example of course we need to tell the delete() functions which playback or fixture exactly to delete. Hence we would write something like `fixtures.delete(29)` in order to have fixture no. 29 deleted. In this case, 29 is the parameter value.
- more generally, when describing a function's parameter, instead of the '29' you would want to denote something like 'Number of the fixture to delete, given as integer'. That's why in general descriptions we write not only the parameter (one word, maybe camelCase, which can be explained later), but also mention the type of value which this function expects. For our delete-fixture-example we would write something like `fixture.delete(Int32 fixtureId)` as our fixture.delete function needs the id of the fixture-to-delete which must be an integer value.
- if the function expects more than one parameter, the type/parameter pairs are separated by commas, like so: `function(type1 parameter1, type2 parameter2, type3 parameter3)`
- finally, while many functions on Titan just do something (without letting you know), there are others which give back a value: the return value. Again in an attempt to do it as generally as possible, we always denote the type of return value before the function itself - and for those functions which do not return anything the type of the return value is void. **When putting such functions to use in a step, simply omit the 'void' keyword. In turn, functions WITH a return value are usually used exactly BECAUSE of this value, and used accordingly.**

Putting this to use we can now at least make something out of the above mentioned function:

- `Void` – this function does something but doesn't return any value
- `Group.StoreMenu` – this is the namespace. Most likely this function is applicable in the Group Store menu (in console notation: <Group> [Store])
- `StoreOnButton` – this is the very function. Probably it stores something (a group, see the namespace) on a button.
- finally, inside the pair of brackets, we have three pairs of type/parameter `String group, Int32 index, AcwUserNumber userNumber`
    - a string which denotes the group

- an Int32 which denotes the index (which exactly needs to be explained separately)
- and finally the userNumber which needs to be given as AcwUserNumber which probably is an object

If you want to cross-check: http://api.avolites.com/10.1/Group.StoreMenu.StoreOnButton.html

StoreOnButton

The user has pressed a preset flash whilst in the store menu. Try and record the group to that handle

Namespace:	Group.StoreMenu
Parameters:	group (String) : Button group
	index (Int32) : Index into that group
	userNumber (AcwUserNumber) : The user number of the group to store

We have come quite close, won't you think?

some more examples

functions with return value

```
Single Math.Cast.ToSingle(Object value)
```

see http://api.avolites.com/10.1/Math.Cast.ToSingle.html

All the Math.Cast functions do explicitely change a value's type , and hence, by definition, DO have a return value. This function takes an object as input (parameter), and spits out a single precision value

functions as parameter - nested functions

The example changexfade takes this a step further:

```
ActionScript.SetProperty("Playbacks.Editor.Times.ChaseFixtureOverlap",
Math.Cast.ToSingle(1))
```

- the **outer** function `ActionScript.SetProperty` sets a property, in this case the property "`Playbacks.Editor.Times.ChaseFixtureOverlap`", to a value
- in this case, this value needs to be a single - and our value '1' is NOT a single value
- hence, we use this casting function `Math.Cast.ToSingle`...
- ... and use this function as **inner** function by putting it as parameter for the outer function

don't be confused by commas - INSIDE a string

```
Void CueLists.GoBack(Handle handle)
CueLists.GoBack("Location=Playbacks,2,1")
```

See cuelistgoback

At first glance this looks like more parameters in the brackets - but it isn't. The commas are inside the double quotes which make the entire part one string: “`Location=Playbacks,2,1`”. And the function definition shows that this is a handle.

---

further readings

- Introduction to macros
- Console and simulator - how actions on the consoles are described
- Recorded vs. coded macros - both kinds: Country, AND Western
- Macro file format - what to observe when creating macro files
- Macro Folders - where exactly are the macro files stored
- Deploying macros - how to import a macro file into Titan
- XML format - a veeeery basic introduction into the format macro files are written in
- The Syntax of Functions - understanding how functions are described in general
- Control Structures - conditions and other means to control the flow
- Action and Menus - when a menu needs to be toggled in addition to the action
- Step Pause - a little delay might sometimes be helpful
- Active Binding - highlighting a macro handle as active
- Namespaces - a way to keep order of the functions, properties and other stuff
- Datatypes - numbers, words, yes & no: the various types of values
- Properties list - the affected system variables of Titan
- Function list - the functions mentioned in this wiki
- Examples list - all the contributed macros. And where is yours?

2017/10/13 15:12 · icke_siegen