Types

All data - in particular properties and functions' parameters - need to be of a certain type. Essentially, this is a convention you know from everyday's life. There are some information which are always a number - e.g. the phone number. Then there are other information which are always words, e.g. a person's name.

Furthermore, there are some sub-types. E.g. a phone number will always follow a specific pattern (per country), a zip code - which in many countries is also a number - will follow another pattern. If you, by accident, dialed a person's zip code instead of their phone number, most likely you will get no connection (which at least gives you immediate feedback that something is wrong), but in rare cases you might even get connected. But most likely not to the person you intended to speak to. Hence, the most important lesson is: types matter. Passing the wrong type of a parameter to a function will lead to wrong results, and most likely to errors (in which case usually the macro does nothing).

And: there are more complex types as well. Imagine the data type 'address'. An address in real life comprises of a street name, a house number, a zip code, a city, and maybe a few more details. in general, such complex data types are refered to as objects or structures, and it is important to know the details of how they are composed.

Here, we only very briefly try to explain the various types you might come across when writing macros. Please feel free to refer to countless programming literature if you are interested in more details.

Casting - type conversion

As types are that important to adhere to, and as functions are very picky about what they want to get as input and what they maybe return, there are means to convert data from one type to another. This is called type casting. While this is a good thing in order to provide the correct data type, this invokes the next challenge: conversion rules. Some are easy to understand, e.g. casting an integer value to a float. Others aren't: what rule would you suggest to cast the string “Moving Light” into an integer value? But usually you shouldn't need to bother. Just use the appropriate casting function, e.g. in the example Chase - Change a chaser's overlap.

Another example about when and when not to reference types:
http://forum.avolites.com/viewtopic.php?f=20&t=5766#p20783

> The ActionScript.SetProperty function can be used to set a value of any type, as the parameter type Object is the root class for all types including non-reference types (e.g. integers, floats). The ActionScript.SetProperty.Integer function only accepts an integer as the second parameter which can have the advantage when parsing input this can be assumed and so the correct type is chosen (particularly in WebAPI). However you can force a conversion by using casts which can work where automatic conversion is not possible for example:
>
> ```
> ActionScript.SetProperty("Windows.Playbacks.FixedColumns", int:"5")
> ```
>
> will work where

```
ActionScript.SetProperty.Integer("Windows.Playbacks.FixedColumns", "5")
```

will not.

The ActionScript.SetProperty.Enum function takes a string as the value parameter and internally converts that string to the correct enumeration type based on the current value of the property being set. WebAPI does something similar when initially parsing values but uses the function parameter type to determine this however as there are many enumeration types there is not a SetProperty function for each one as this would be impractical. You could do the following:

```
ActionScript.SetProperty("Windows.Playbacks.ButtonSize",
  Math.ToEnum(
    "Avolites.Titan.Controllers",
    "Avolites.Titan.Controllers.HandleButtonSize",
    "Fixed")
  )
```

but that is very long-winded in comparison to the `ActionScript.SetProperty.Enum` function.

Datatypes in this Wiki

Here is a list of the primitive types covered in this wiki:

- Boolean
- Double
- Enum
- EventPropergation
- Flag
- Float
- IEnumerable
- Int32
- List`1
- Null
- Object
- Single
- String
- Void

And the more complex object types:

- AcwColour
- AcwProgrammerIdPair
- AcwRecordMask
- AcwTimecodeSource
- AcwTimeSpan

- AcwUserNumber
- DmxAssignment
- FrameRate
- Leveladjust
- TimecodeTime
- Timestamp

---

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