core.lua functionality

- base subclass
- defining new classes
- creating/deleting objects
- adding/removing/testing a single tag.
- single tag queries
- defining new global methods.

All these functions (and virtually all others) are methods of the un-named base class which is accessed via the global *system.* Some are marked ‘private’ which means only the system should use them.

<Class Ref>,<Superclass Ref> :createClass(<class name>,[<superclass name>])

Creates a new class with the given name, using the given superclass - the latter is optional, and if undefined it is the base class. Class names are case insensitive alphanumeric separated by full stops (e.g. system.support.timer). Returns the Class Reference, and the Superclass reference. If no superclass is provided, the superclass reference returned is that of the root object.

Class files should be held either in java style (e.g. system.support.timer is in system/support/timer.lua) or together in a single file with all classes in that group (e.g. system.support.timer is in system/support.lua and so is everything else in system.support.*)

Having a duplicate class, or an unknown superclass causes an error.

<Instance Ref> :new(<class name>,[<constructor table>])

Creates a new instance of a class with the given name, which is case insensitive and should already be defined. The second parameter is optional, and if not provided an empty table is used. The first parameter must be a string and a known class, and the second parameter, if defined must be a table. The method creates a new instance of the class using the class reference as a metatable (decorated objects are no longer supported). Once created it calls the *constructor* method of the class passing the constructor table, or the default empty, as a parameter. A default constructor is provided that does a *shallow* copy of the constructor parameter (e.g. references are copied as references) by default. This default constructor is not callable using the SuperClass provided by createClass().

A count is kept of all instances of a class, so to help track instances that are created and not deleted. It sets an internal member (__isAlive) of the object to mark the object as alive. All instances are automatically tagged with “__object”.

void :dispose()

Disposes of the instance. It calls the *destructor* method of the class (a default is provided which does nothing) with no parameter. After this, the __isAlive flag is set to false, all tags featuring the object are removed, any singleton name is removed, and the instance count is adjusted.

boolean :isAlive()

This returns the __isAlive flag, so this will be false if the object has been disposed. This is to allow whether objects have been disposed or not. The problem is that if a reference is kept to the object, then it will 'still be there' even though it will have no functionality (e.g. sending delayed messages to an object which has been destroyed in the interim).

string :getClassName()

Returns the class name of the instance (debugging support), obtained by accessing its metatable and searching for it in the class name:class instance table. This is a slow search.

void :analyseClassUsage()

Prints the total instances currently defined for each class, in alphabetical order on stdout. This is for debugging support *only.*

<instance> :singleton([<name>])

Declares the given object to be a singleton. It does so by accessing the class via the metatable and replacing its new method with one that reports an error. It should be noted that a consequence of this is its subclasses cannot be instantiated by calling the SuperClass.new(self…) approach. If this is required for some wierd reason, put the constructor code in a seperate method, called from the constructor.

Singletons can be named ; the name should be alphanumeric and case is ignored, as ever. This should be used for semi-permanent instances, system, music players and so on that exist through the life of the app.

<instance> :getNamed(<name>)

Finds a named singleton object and returns a reference to it, throws an error if the reference is not found.

<Instance Hash>,<Instance Count> :__getAllObjects() private

Returns a table (instance:?) of all currently defined objects and a count of the number of objects in that table. This uses the tag list for "__object" and the instance count, thus the return value should be considered read only.

void :__defineGlobalMethod(<method name>,<function>) private

Adds a new function to the base object. The method should not already exist, and both parameters are required. This allows new methods to be added to the global object.

boolean :__setTagState(<tag name>,<is Set>) private

Sets the state of the tag for the current instance to set or unset (e.g. it is tagged with it, or it isn't). Returns the previous state of the tag for this object. Tag names should be case insensitive alphanumeric strings, and the set flag should be boolean. This method does not care whether a tag is currently set or not (save for the return value) but must of course maintain the tag indices correctly.

<Instance Hash>,<Instance Count> :__getAllTaggedObjects(<tag name>) private

Provides a similar function to __getAllObjects(), except it does so for a given tag. If the tag is unknown, it should return {} , 0. As with __getAllObjects, the hash returned should not be modified as it is a live index.

The tag methods should not be used, as there are methods in the tag library which call them to provide normal tag functionality.

<boolean> :__execute(method <string>, [4 parameters]) private

This is an error trapped method executor, used so (for example) when a timer or update event causes an error, it stops the system rather than repeatedly fires the same method. It returns true if the method was successfully executed on the object.

First version 24 Nov 2014
Added __execute 26 Nov 2014
createClass returns Root as SuperClass of classes without 27 Nov 2014