Module 2.0 Reference

Overview

QDL introduced modules as its encapsulation mechanism from the start. However, while the basic workings of them are largely unchanged, management of them turned out to be a serious issue. The major problem is that the original modules were imported into a register for the system and were therefore not really easy to manage – you were not sure if a module was imported into your specific scope and there was no real way to tell.

On top of this, modules were given their namespace alias, but as long as there were no name collisions, you did not need that. This confused many people again as to how they were scoped. In particular, this made using them in functions very hard. Since QDL is a functional language, that is unacceptable.

Improvements

The easiest way to fix all of this was to simply allow modules to be variables and to be treated as such. You import them (to create an instance), then can pass them around wherever you like. They must therefore always be scoped. If you want modules to be unscopes, *i.e.*, dropped into the current scope, then there is a specific command called **`use`** to do that. Making it explicit simplifies using them a great deal.

Example. Contrast the old and new

We load the standard cryptographic module and use it both ways.

```
     module_load('edu.uiuc.ncsa.qdl.extensions.crypto.CryptoLoader', 'java')
qdl:/tools/crypto
  module_import('qdl:/tools/crypto', cc)
cc
```

This requires you to load the module and import it. Then the functions are in your workspace though they may be prefixed with **`cc#`**.

Contrast this with using version 2.0

```
  crypto := j_load('crypto')
```

which has looked up the class path from the lib entry of info(), loaded it and created it. It is now stored in the variable **`crypto2`**. You cannot just list a module's functions in 1.0, you have to list all modules:

```
  // list the functions in the module
  )funcs -m
create_key([0,1,2])  decrypt([2,3])  encrypt([2,3])  export_jwks([2])
import_jwks([1])  rsa_public_key([1])  s_decrypt([2])  s_encrypt([2])
8 total functions
```

though you can add the -fq switch and fully qualify them. In 2.0, you can list the functions in a given module either programmatically:

```
  funcs(crypto)
[create_key([0,1,2]),decrypt([2,3]),encrypt([2,3]),export_jwks([2]),import_jwks([1]
),rsa_public_key([1]),s_decrypt([2]),s_encrypt([2])]
```

or using the workspace command with the -m switch

```
  )funcs -m crypto
create_key([0,1,2])  decrypt([2,3])  encrypt([2,3])  export_jwks([2])
import_jwks([1])  rsa_public_key([1])  s_decrypt([2])  s_encrypt([2])
8 total functions
```

To list the module level help, you can either give the variable or, if the module has been loaded, the uri:

```
  )help -m crypto2
  module name : CryptoModule
    namespace : qdl:/tools/crypto
default alias : crypto
   java class : edu.uiuc.ncsa.qdl.extensions.crypto.CryptoModule
QDL's crypto graphic module. This has a variety of operations possible…
```

Furthermore, unlike 1.0, you can pass the module along as an argument to a function:

```
 f(x)->x#create_key()
  f(crypto)
{
 p:AMIWAWnFsHk…
```

This allows you to have various instances of the module with various specific states. E.g.

```
transactions := j_load('db');
     clients := j_load('db');
// configure these so you can define a function like
   count(database, table_name)→
          database#read('select count(*) from ' + table_name);
```

You may also return a module as the result of a function. It is, in fact, just another variable.

Migration from 1.0 to 2.0

There is really not a lot to do except replace the load and import statements,. If you are running these in your workspace and have scripts that access them, you should decide what name you want to use and qualify any references in your scripts. You can do this still using 1.0 which does not care, then you can change your QDL configuration to load version 2.0 (you can even specify what variable to assign to) and it should just work.

Function reference

docs

Description

List the documentation for a module.

Usage

docs(uri | var)

Arguments

uri = the uri of the module (assumes it has been loaded, but not necessarily imported)

var = the variable name containing the imported module

Output

A stem of lines consisting of the module's documentation

funcs

Description

List the functions contained in a the workspace or a module.

Usage

funcs({null | var} {,regex})

Arguments

{no arg} - list all of the functions in the ambient (unqualified) scope.

null - same a no argument

var - the variable for a module

regex = optional regular expression to filter the results.

Output

This will return the functions in compact notation (so the name and arg counts) for either the workspace (no argument or null) or the module. The module may be given as either the (imported) variable name or a string for the module (if not imported).

The

Examples

```
   funcs()
// all the functions in the current workspace
   funcs(null)
// same as funcs()
   crypto := j_load('crypto')
   funcs(crypto)
[create_key([0,1,2]),decrypt([2,3]),encrypt([2,3]),export_jwks([2]),
 import_jwks([1]),rsa_public_key([1]),s_decrypt([2]),s_encrypt([2])]
// Now select only those that start with e.
  funcs(crypto, '^e.*')
[encrypt([2,3]),export_jwks([2])]
```

The reason that a null first argument is allowed is so you can get the functions in the current workspace and specify a regex. A string (which is allowed as the first argument) would be the namespace of a loaded but not necessarily imported module and might be used to, *e.g.*, determine if a function is present in the module before importing it.

import

Description

Take a loaded module and create a new instance of it. This must be stored in a variable.

Usage

import(uri {, import_mode})

Arguments

uri - string that is the namespace for the loaded module.

import_mode - either an integer or string for the import mode. Supported mode types are

Name	Integer code	Description
none	100	No shared state. The module is self-contained. This is the default
inherit	101	Inherits the current state, which is then copied and is hence decoupled from the ambient environment
share	102	Shares the current state with the ambient environment. Changes to the environment are shared in the module.

Output

The module. Make sure you capture in a variable for later use.

Examples

Here is a simple example of inheritance modes in action.

```
  a:=3 /// stick this in the current state.
  module['a:a'][f(x)->a*x;]
  inherit := import('a:a', 'inherit')
  inherit#f(2)
6
  shared:= import('a:a', 'share')
  shared#f(2)
6
```

Both w and z seem the same, but if we change a,

```
  a := 5
  shared#f(2)
10
  inherit#f(2)
6
```

This illustrates the point that **`inherit`** inherited the state when it was created by having it copied. On the other hand, **`shared`** is still coupled to the ambient state. Contrast with

```
  u:=import('a:a')
  u#f(2)
unknown symbol 'a' At (1, 20)
```

which is the default of **none** meaning there is no shared state of any sort. In this case, since a was not defined in the module, attempts to use it will result in an error.

j_load

Description

Load and then import a java class.

Usage

j_load(string)

Arguments

string - *either* the fully qualified class name or an entry in **`info().lib.`**

Output

The module. Be sure to capture it in a variable or some place else.

Examples

Loading and assigning several modules at once. QDL does allow for multiple assignments of stems, so if you wanted to make a bunch of modules at once you could issue something like this.

```
[p,q] := [j_load('crypto'), j_load('crypto')]
```

so p and q are now initialized to the give modules (which can be anything). And yes, you could have issued

```
a. := [p,q] := [j_load('crypto'), j_load('crypto')]
```

and **a.** would now be a list of modules.

Note that the lib_load value in the configuration file (see the configuration manual) allows you to add entries to the **info().'lib'** stem. If you do add entries, then the search path is relative to that, so let's say you added **info().'lib'.'my_lib'.'math'** where the value of this is the fully qualified Java class name. Then if you issue

```
j_load('my_lib.math')
```

it will be located and loaded. There is a special case that a simple string (no dots) is resolved against the system tools, so the following are the same

j_load('cli')
j_load('tools.cli')
j_load(info().'lib'.'tools'.'cli')
j_load(lib_entries().'tools'.'cli')

j_use

Description

Load and import a java module into the current scope.

Usage

j_load(path)

Arguments

path - either the fully qualified name of the Java class or a relative path to a **info().'lib'** entry.

Output

Boolean true if it was loaded, false otherwise.

Examples

See note at the end of j_load for how library paths are resolved.

lib_entries

Description

List the library entries in the system. These are class paths of modules that may be easily loaded.

Usage

lib_entries({key, stem.})

Arguments

(none) - list all of the library entries for the system. This is equivalent to issuing **`info().'lib'`**.

key - the name of this library

stem. - a stem of name : classpath pairs.

A typical stem would look like (this is the default for the system tools)

```
          cli : edu.uiuc.ncsa.qdl.extensions.inputLine.QDLCLIToolsLoader
      convert : edu.uiuc.ncsa.qdl.extensions.convert.QDLConvertLoader
       crypto : edu.uiuc.ncsa.qdl.extensions.crypto.CryptoLoader
           db : edu.uiuc.ncsa.qdl.extensions.database.QDLDBLoader
  description : System tools for http, conversions and other very useful things.
         http : edu.uiuc.ncsa.qdl.extensions.http.QDLHTTPLoader
         mail : edu.uiuc.ncsa.qdl.extensions.mail.QDLMailLoader
```

Note that it does contain a description, which is, of course, not loadable, but sure helps with the readability.

Output

The new lib entry.

Examples

This just spits out the standard tools for a basic QDL install. If you have more libraries loaded, it may be considerably different.

```
lib_entries()
{tools:{cli:edu.uiuc.ncsa.qdl.extensions.inputLine.QDLCLIToolsLoader,
 convert:edu.uiuc.ncsa.qdl.extensions.convert.QDLConvertLoader,
 crypto:edu.uiuc.ncsa.qdl.extensions.crypto.CryptoLoader,
 db:edu.uiuc.ncsa.qdl.extensions.database.QDLDBLoader,\
 description:System tools for http, conversions and other very useful things.,
 http:edu.uiuc.ncsa.qdl.extensions.http.QDLHTTPLoader,
 mail:edu.uiuc.ncsa.qdl.extensions.mail.QDLMailLoader}}
```

load

Description

Loads (i.e. reads into the current workspace) either a QDL or Java module. Note that for QDL modules this exact same effect can be gotten from

1. typing in a module[…] statement and hitting enter
2. reading in a text file that contains such a module statement
3. using this.

The main use of this is therefore loading java implementations of modules.

Usage

load(path {, type})

Arguments

path - (QDL modules) the full path to the file containing the module statement. Note that this function is FVS aware. (Java) the full path to the class.

type - either 'qdl' or 'java'. Note that the system tries to figure it out so normally this is not needed, but you can add it if you insist.

Output

The namespace of the loaded module, suitable for use with the **`import`** function.

Examples

```
   load(info().'lib'.'tools'.'cli')
qdl:/tools/cli
```

This means that the java module for CLI (command line utilities) tools is now in the workspace and can be used.

loaded

Description

List all of the currently loaded modules in the workspace or check if a specific module is loaded.

Usage

loaded({arg | arg.})

Arguments

(none) - return a list of all loaded module in this workspace

arg - a string that is the namespace of a module

arg. - a stem of strings that are namespaces

Output

(none) - A list of namespaces of the loaded workspaces.

arg. - a conformable stem whose values are true if the module is loaded and false otherwise.

You can remove loaded modules with the **unload** function. You can rename them with the **rename** function.

Example

```
  loaded()
[my:/cli,qdl:/tools/crypto]
```

There are two modules loaded in this workspace.

```
   loaded(['qdl:/tools/crypto', 'qdl:/tools/cli','qdl:/tools/convert'])
[true,false,false]
```

In this case, only the first module is loaded. A typical use might be

```
   cli := loaded('qdl:/tools/cli')?cli:j_load('cli');
```

where a script might have the cli already imported at this point, but if not, just load it.

rename

Description

Rename a loaded module's namespace. This can be useful in some specialized cases. It does nothing to currently active modules, it just changes behavior from that point forward.

Usage

rename(old_ns, new_ns)

rename(stem.)

rename(old_list., new_list.)

Arguments

old_ns - string that is the old namespace

new_ns - string that is the new namespace

stem. - this has the structure that stem.old_ns := new_ns,

old_list., new_list. - list of strings such that old_list.k and new_list.k are the old and new namespace

Output

A conformable result of true if the rename worked, false if it failed.

Examples

Example of loading a module, then renaming it. We check the name with the `loaded()` function.

```
  load(info().'lib'.'tools'.'cli')
qdl:/tools/cli

  rename('qdl:/tools/cli', 'qdl:/tools/cli2')
true
  loaded()
[qdl:/tools/cli2]
```

Note that if we try to rename it again, we get a false (since the name has changed)]

```
    rename('qdl:/tools/cli', 'qdl:/tools/cli2')
false
```

use

Description

Import a module into the current ambient scope. This means you do not need to qualify references to the members.

Usage

use(path{,type})

Arguments

path - (QDL) the full path to the file containing the module. (Java) The full class path to the implementation.

type - 'qdl' or 'java'. The system will attempt to figure out the type automatically, so this is usually not needed.

Output

A true if it worked, false otherwise.

Examples

```
   load(info().'lib'.'tools'.'cli')
qdl:/tools/cli
   use('qdl:/tools/cli')
true
   )funcs
to_stem([0,1,2,3])
1 total functions
```

This shows that the single function for this module has been loaded into the current workspace and is just another function. You do not need to qualify it to use it.

unload

Description

Remove a loaded module from the workspace.

Usage

unload(uri | stem. | list.)

Arguments

uri - the string for a single uri

stem. - a stem of uri strings to drop

list. - a list of uri strings to drop

Output

A conformable stem or list of booleans, with a true if the item is no longer in the workspace and a false otherwise. Note that this does *not* remove instances from the workspace, it merely prevents you from importing more of these modules.

Examples

```
   unload('oa4mp:/tools/claims')
true
```

In this case, a single module is being unloaded.

```
   unload({'claims':'oa4mp:/tools/claims', 'jwt' : 'oa4mp:/tools/jwt'})
{claims : true, jwt : true}
```

In this case, two modules are unloaded and the returned stem preserves this information.

vars

Description

print the variables inside a module. This follows the same syntax as the funcs() function.

Usage

vars({null | var | uri {,regex})

Arguments

(none) - get all of the variables in the current scope.

null - same as no arguments.

var - module variable

uri - the fully qualified namespace for a loaded, but not necessarily imported module.

regex - optional second argument to filter the names

Output

A list of variable names as strings.

Examples

```
   vars()
[a,  b,  phi]
```

This is a list of the variables in your workspace (results will vary).

```
  mm := import('qdl:/ext/math')
  vars(mm)
[primes.]
```

This is the standard math library and it contains a stem of the primes < 10,000.

```
  vars(mm, '^q.*')
[]
```

This filters for all of the variables in the module that start with q – there are none.

Do note that this function can be used with the **`apply`** function that takes names of variables.