Old Module Reference

Version 1.5.1

As of QDL version 1.5.2 modules are now first class objects and are assignable to variables. The old machinery for them still works, but should be considered deprecated and replaced. Below is the entire old reference manual entry for them. Generally there are referred to as version 1.0 modules as opposed to the newer 2.0 modules. Use 2.0 modules if at all possible!

Modules

Namespaces

A ***namespace*** is a set of names that refer to a set of objects. Using a namespace ensures that all objects have unique names with respect to that namespace. A very common example is a file system, where every file within a directory has a unique name. In QDL this is accomplished through modules. Each module has its own namespace and every object in the workspace is associated with a namespace. The namespace qualifier is a URN. It is profitable to think of namepsaces as a dictionary of resources.

Encapsulation

One of the most basic ideas in programming is *encapsulation* that is to say, a group of statements with their own state (so the variables and functions know about each other and their workings are independent of the rest of the environment). This is the concept of a ***module*** in QDL. Very simply, a module is defined using a unique identifier that **must** be a URN and an alias. The alias is just a regular name like a variable (same syntax). Aliases must be unique within the scope. One of the great evils of many scripting languages is that there is no encapsulation – every variable is global and the net effect is extremely hard to find bugs.

Things to do with modules

- encapsulate sets of functions to extend the workspace or QDL. E.g. a module that implements a set of specific Math functions. Typically you would want to dump these into the current scope with the **use** command.
- encapsulate specific functionality, e.g., a module to do database processing. You may import then various modules to talk to various databases.
- encapsulate specific utilities.

Module syntax

Modules are defined as

```
module[urn{, alias}]
```

```
{body}[
   === module level comments.
   statements];
```

The alias is a suggested default alias. It is optional. The statements have local scope, so inside the body of the module you do not need to qualify variables or functions. Modules may be nested. Modules may also reside in an external file (with the suffix .mdl) and may be loaded. Loading a module is little more than reading it into the current workspace.

Import, use

There are two ways to get access to the module.

`import` will create a new instance and hand it back. You must assign it to a variable to use it.

`use` create a new instance and dump the contents into the current scope. You do not need to qualify the functions or variables and they may be treated like any other QDL object.

Example

Let us say that you had loaded the standard mathx module that comes with QDL. You could then either import it with its own namespace

```
   mathx := import('qdl:/ext/math')
   mathx#cosh(3)
10.0676619957778
```

Alternately, you could simply use it and make it an extension to the current workspace

```
   use('qdl:/ext/math')
   cosh(3)
10.0676619957778
```

Another use is inside of an module, where you can make it local

```
   module['my:/ext/math']
         [load('/path/to/modules/mathx.qdl');
          use('qdl:/ext/math');
          versinh(x)→ 2*sinh(x/2)^2; // hyperbolic versine
         ];
```

More modules

Modules may defined inside other modules and they may be imported into other modules. If module A imports B then you would access the variable *u* in B as

A#B#u

If you import or define a module inside another module, it is completely local to that module.

Intrinsic elements

Modules may have variables and functions that are only available in that module. These are referred to as ***intrinsic*** (meaning “originating and included wholly within a part”). This is done by a convention of starting the name with a double underscore. Any such elements inside a module are only visible there.

```
module['a:a','X'
 ][
 __secret_number := 42;
 __f(x)->x/__secret_number;
  g(x)->__f(x+1);
];
```

Would define a module with a single extrinsic (the opposite of intrinsic) function `g(x)`.

```
   X:=import('a:a')
   )funcs -m
g(1)
   )vars -m
```

Meaning that __f and __ secret_number are not visible. Attempts to access or change them will result in errors:

```
   X#__secret_number := 1009;
cannot set an intrinsic variable
   X#__secret_number
unknown symbol
```

The aim is that the internal state of a module cannot be changed by users. This ensures integrity, so that nobody can change the expected state/behavior of a module willy nilly. Similar for intrinsic functions. You may, however list them in a module with the -intrinsic -fq and flags:

```
   )funcs -m -fq -intrinsic
X#g(1)  X#__f(1)
```

This say to show modules, fully qualify the names, show intrinsic functions too. If you define intrinsic functions and variables in the current workspace, they will display, since that is their natural scope.

Extrinsic Variables

These are also know as global variables in some languages. (***Extrinsic*** means “not part of the essential nature of someone or something; coming or operating from outside”) They are simply variables prefixed with a $$ and are available every where at all times to all functions, modules, etc. You access them like any other variable.

```
   $$Avogadro := 6.02214076E23
   define[moles(x)][return(x/$$Avogadro);]
   moles(1.3766E25)
22.858980798715173
```

Related Functions

jload

Description

Shorthand for loading a single java module by class name. This is the same as

```
module_import(module_load(class_name |  moniker, 'java'))
```

See also `lib_load` which is in the extensions module and will let you easily load one of the built-in system modules (like http or convert).

Usage

```
jload(class_name{, alias})
```

Arguments

class_name - Either the full class name of the module or the stem path in the info().lib stem. Make sure the class is available if you wrote the extension.

alias - (optional) an alias on load. If omitted, then the default alias for the module is used.

Output

The alias of the module imported

Examples

The `info()` function contains a handy list of all available standard extensions with their classes.

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

so to load the db module you would issue

```
   jload(info().lib.tools.db)
db
```

OR

```
   jload('db')
db
```

which tells you the module was loaded, imported and the alias for it is db. This would look up the module in the stem path starting at lib. Extensions to QDL (such as the OA4MP workspace) also have their own paths and an example there might be

```
   jload('oa2.util.jwt')
jwt
```

You could even supply an alias if you wanted.

module_import

Description

Import a module with a given alias.

Usage

```
module_import(urn{, alias});
module_import(arg.)
```

Arguments

Dyadic version with simple scalar arguments

`urn` = a Uniform Resource Name (as per RFC 2141 and 1737, if you track such things).

`alias` = (optional) a string that conforms to the requirements of a QDL variable name.

Note that both of these are passed in as strings and the URN is checked for form. Also, when a module is defined, it has an alias given. This is the default, so you may import the module with only the URN. If you specify another alias, that is used.

Monadic version.

`arg.` A stem of either urns or urns and aliases. If an alias is omitted, then the default issued.

E.g. A list

```
   arg. := [[urn0{,alias0},[urn1{,alias1}, . . . [urnn{,aliasn}]
```

Or

```
   module_import(urn0, [urn1, 'curves'], [urn2. 'complex'])
```

Or something like

```
   arg.0 := urn0; // assumed defined in the workspace somplace.
   arg.'complex' := ['qdl:/math/complex', 'c']
   arg.'curves' := ['qdl:/math/curves', 'curves']
   module_import(arg.);
```

The result would be conformable to the arguments.

Output

A logical (dyadic) alias as a scalar string, (monadic) conformable stem of aliases if the import succeeded or `null`.

Examples

```
   module_import('my:/thingie', 'ahab');
ahab
```

Would locate the module with URN *`my:/thingie`* and let you use *`ahab`* as the local alias. Remember that the module must be loaded first or you will get an error.

Another example of multiple imports

You may import a module multiple times with different aliases. This effectively creates new independent copies of it. In the following small session, a module is created (this effectively loads it, so you do not need the `load_module` command. It is then imported twice, first using the default name, then with a different alias. The values are set so you may see the they are independent. (This is very much like most object oriented languages which have a notion of a class, which is a user defined template from which instances are created. The instances are accessed using their alias. So if you do python or java, `import(a,b)` is the same as creating a new instance of a class.)

```
  module['a:/a','a']body[q:=1;];
  module_import('a:/a')
a
  module_import('a:/a','b')
b
  )modules imports
a:/a = [a,b]
  a#q :=5;
  b#q :=6;
  a#q
5
  b#q
6
```

module_load

Description

Load a module from an external source. Note that once a module is loaded, it is automatically available for import in any other module. Since namespaces are unique, there should *never* be a collision. Said differently, there there is a master list of all loaded modules in a given workspace.

Usage

```
module_load(arg{,type});
module_load(arg.)
```

Arguments

Dyadic version, using simple namespace and alias:

`arg` – Either the full path to the file that contains the module, when there is no second module *or* the fully qualified Java class name. The Java class may be either a class that extends the QDLLoader class (to load multiple modules) or extends the JavaModule class. This is loaded then run in its own environment which is then added to your session. *Be sure the class is included in your class path*.

`type` - If absent this implies file. The only other supported value at this point is `'java'` which means that the first argument is a class name.

Monadic version,

`arg.` - a stem of modules namespace or namespace, type pairs.

E.g.

```
arg. := [[arg0{,type0}, arg1{,type1}, . . . , argn{,typen}]
```

Same note about using a general stem holds here as for module_import.

Note that you can also load modules at start up via the configuration file. You should look in the documentation there for more information.

Output

A either a scalar (dyadic case) of the alias or a conformable list of aliases if it succeeds. Failures return a **`null`** otherwise (or an error message if there was an actual problem with loading the module).

Examples

```
   module_load('/home/bob/qdl/modules/mega_module.mdl');
```

Would load the given module.

Loading a Java module

The example loader ships with QDL, so you may always import that. To do so issue the following:

```
   module_load('edu.uiuc.ncsa.qdl.extensions.example.QDLLoaderImpl', 'java')
qdl:/examples/java
```

This returns the actual namespace that was created, not the class name. To check that it was imported, we ask the workspace, import it and then check again.

```
  )modules
qdl:/examples/java
  )imports
(no imports)
  module_import('qdl:/examples/java')
```

```
java
```

And import returns the (in this case default alias) of `java`. If it imported correctly, it should have brought in a single command.

```
  )funcs
java#concat(2)
```

module_path

Description

Returns or sets the list of paths used to resolve the loading of modules from relative paths. These are file system (including virtual) paths.

Usage

`module_path({arg})` - Query or set the module path.

Arguments

`arg` - if given, sets the path. It is a colon separated list, e.g. 'path0:path1' *or* it is a stem of paths.

Note that if relative path has a scheme for a VFS (e.g. vfs#ligo/v4) , then this restricts resolution to those paths. If the path is unqualified, every path is checked.

Output

If a query, the current module path.

Examples

```
    module_path()
[/home/ncsa/dev/ncsa-git/qdl/language/src/main/resources/modules/,
vfs#/scripts/modules/]
```

This means that loading a module from a relative path like `my/lib/module.mdl` would check against the first path, since it is not VFS qualified, but a relative path like `vfs#math/complex.mdl` would be resolved against the second.

module_remove

Description

Remove an imported module by alias

Usage

`module_remove(arg)`

Arguments

`arg` - A string or stem of strings that are aliases for currently imported modules. They will be removed *in toto* from the environment.

Output

A boolean or list of booleans showing success.

Examples

```
   module_remove('transactions')
true
```

This removed the module with alias transactions. Note that if you can remove a module you did not import. That means that if a module with an alias has been loaded before the current clode block and you simply remove it, it will be removed at the point it was added. This may or may not be exactly what you want to do (e.g. replace the current module with one of your choosing) so this is documented.

Use of modules

Since this is the basic encapsulation unit their use requires a bit of amplification. If you are familiar with classes, they behave a bit like them, but are not classes (which have internal private functions and variables, as well as some form of inheritance). For one thing, nothing is truly private and a lot of languages like Java or C++ with **`public`** or **`private`** keywords are really mostly syntactic sugar. In QDL, the convention is simple: If a function or variable begins with a double underscore, that is an indication that it should be left alone. But there may be multiple instances of modules and they are independent. For instance, if you have a database module that manages access to tables, say

```
 module[
   'my:/db/access/table', 'table'
  ][
    define[init_connection(table_name)][. . . ];
    define[select(query_string)][. . .];
   // other stuff
];
```

then you might want to create several tables like this, each is a module (this assumes that your module path is set and `db-table.mdl` is in it.)

```
db_table := module_load('db-table');   // make it available in the workspace
module_load(db_table, 'clients');
module_load(db_table, 'accounts');
module_load(db_table, 'transactions');
```

Then you would initialize each of these land use them like

```
clients#init_connection('client_table_name');
clients#select(my_query); // select on a give criterion.
```

and then each instance of the module would have its own state, independent of any other instances.

Why not have classes? Because QDL is an interpreted functional programming language (so functions can be created on the fly in context, rather than only residing in a class) and classes often badly interact with that model, resulting in very clunky, hard to maintain code. However, some form of an encapsulated unit is a must for clean programming. Therefore, the module was created to do this.

Appendix

The old module management system. Starting with QDL version 1.6, modules may be assigned to variables and used like any other variable – pass them to functions etc. Before that, modules had a much less intuitive and clunky management system. You either defined a module directly (required the alias) or issued a **`module_load`** command. Then to use it, you would issue a **`module_import`** command which would make it available. At that point in time, the assumption was that modules would be large syntactic units but usage showed that smaller code blocks were a much better idea. The next couple of bits are for people using the older module system. The newer system is documented in the manual. The old module system is still supported and will be for quite some time. It is, hwoever, strongly urged that you use the newer system and updates to the old system will not be made.

Creating modules

To use a module, you must import it. Very roughly import is the same as instantiating a copy it. Modules may be created and nested anywhere.

Let us say that we had the following script in the file my-module.qdl:

```
module[
   'a:a','A'
]body[
    === Sample module
    foo := 'abar';
    define[f(n)]body[return(n+1);];
]; // end
```

The URN puts this in a unique namespace. There can be no conflicts. The alias is used as a shorthand to access it. Think of the definition as a template and aliases as instances created from it, each with their own state. URNs many be quite complex. Generally though you should not put in query parameters and such. To access anything in a module explicitly you must ***qualify*** the name *i.e.*,

```
alias#name
```

It is easiest to see this in action, so in the interpreter (clear state)

```
    module_import(module_load('my-module')); // load and import
A
    )funcs

    )vars
```

Note that there is nothing in this session so these show nothing. Let's import our module above in a file called `my-module.qdl`. We are going to import it with another alias just to show we can

```
module_load('my-module.mdl');
module_import('a:a', 'b');

    )vars
b#foo

      )funcs -m -fq
A#f(1)
```

Loading the module makes the session aware of it, but you must also import it. You may import a module multiple times with different aliases. If we did not specify the second argument, then the default alias in the module definition, `'A'`, would be used. Now at this point, there is nothing else in the session and there are no name clashes possible, so we can use the unqualified name:

```
    foo
abar
```

And we can invoke the function too

```
    f(5)
6
```

For QDL modules, module_load is a convenience. You can exactly the same behavior by reading a file with module statements in it. It is, however, needed to to load a Java-defined module.

Local variables and functions vs. modules ones

Remember that the workspace is its own module and has no prefix. You may have like-named variables and functions in various modules. So if we define foo in the workspace then import the module

```
    foo := 5;
    mm: = module_load('my-module');
    module_import(mm, 'b');
b
    )vars -m
b#foo, foo
```

This shows us that there is a session variable with the same name.

```
   foo
```

```
5
   b#foo
abar
```

The most straightforward way to handle name clashes, *viz*., if there is any conflict, require the user resolve it rather than having the interpreter doing it and risking getting it wrong. Part of the philosophy with QDL is that is avoids a lot of the sorts of things that cause hard to find errors and tells you up front how to fix it. The same sort of qualifications hold for functions.

Help for modules

Just like functions, you can create help for a module by inserting documentation commands right before the body. You can access this in the workspace with

```
)modules name|alias -help
```

Here

```
   )modules b -help
a:a [b]
Sample module from the reference manual.
```

This prints out the namespace [alias0, alias1, . . . aliasn] and any module-leveldocumentation.

Note that all of the functions in a module (with their fully qualified names) are automatically found by the help system, so this command is just for the module itself.