Writing an Extension in Java

Introduction

QDL has its own system for writing modules, but some times you need to add functionality that is not in the base system. You may write your own custom Java code and import it. It will function like any other QDL module. It is really quite simple. There are two interfaces the you need to implement, one for functions and one for variables and over-ride a single method to load it all.

Modules

A module the basic “encapsulated unit of execution” in QDL, *and* it has namespace qualification too. That means that it has its own state and when you execute functions from a module, they execute there. All modules have a URI (namespace) that identifies them uniquely.

Anatomy of the module definition

Consider the following QDL module definition:

```
module[namespace]
  body[
        === comments...
      functions and variables…
     ];
```

The corresponding Java classes that model this are:

`JavaModule` (class) corresponds to the `module[..]` statement and `=== comments`. It is also charged with creating new copies of the module when the **`import`** function is called.

`QDLMetaModule` (interface) corresponds to the contents of the `body`. This has two (optional) methods if you need to save state on workspace saves. See section below. As a mater of fact, if you are not saving state, you don’t even need to implement this.

`QDLLoader` (class) corresponds to hitting enter in the workspace which causes the module to be interpreted or, if this is in a file, issuing a **`load`** command (which parses the statement, sets up state &c.).

`QDLFunction` (interface) a QDL function

`QDLVariable` (interface) a QDL variable.

Lifecycle of a module

A module is loaded , which makes the workspace aware of it, then imported as needed. Importing a module means essentially making a working copy, aka an instance of it. This is typically assigned to a

a variable which is then referenced. Modules instances need not be unloaded, simply delete the variable that refers to it. If you **`unload`** a module, then it is completely expunged from the workspace and you cannot import it any more. This means from the perspective of writing an extension that you need the actual module and a loader.

How the java classes are used. Note that we are being pedantic when we say that an instance of **`JavaModule`** is returned, this means (since it is an abstract class) an extension of it. An "instance of" literally means the Java **`instanceof`** would return true,

load is called

- **`QDLLoader.load`** is executed, returning at least one instance of **`JavaModule`**.
- **`JavaModule.newInstance`** is called with a null State object. This creates a *template* or shell of the module which is then used to create stateful copies. It is at this point that the function and variable implementations are processed.

import is called

The existing template is retrieved and its **`newInstance`** with a clean state object is called.. Your implementation should detect a non-null state object and call the **`Javamodule.init`** method. The action of the **`init`** method is to populate the state object with any variables, functions (possible other modules too) and hand back a fully functional module instance.

System State object vs. class state

The system has a State object which contains all of the current variables, etc. for the workspace. Each module has its own local state object with just the module's values. On import, it is populated with your functions and variables by the **`JavaModule.init`** function. The system state mutates rapidly under use, and hence each call to your function it is passed in. You can retrieve values from the workspace or set them as needed, though generally that is a special case. Moreover, when the workspace is saved, this state object is pickled as needed, so you don't really have to do anything. While you can certainly manipulate the state object in your class, generally you won't need to.

Java classes have their own state (defined as the values of the class variables, etc.) which may be need to be persistent. There is a mechanism for this in the **`QDLMetaModule`** interface. That has two methods for pickling (*aka* serializing) and unpickling. A full example is below. In summary, you design and create a JSON object which is then automatically managed for you.

Example code

There are two complete examples in the source code. These may be loaded at any time and the source code is available too. There are two examples.

The basic example.

This is in the package **`org.qdl_lang.extensions.examples.basic`**. This shows how to make a module whose implementing functions and variables are independent of each other – so no shared Java state. It consists of a loader, module class and a separate class for each function and variable. Since it is an example, it shows how to create an extrinsic variable and function and the lifecycle of that.

Included classes

Here is a table of the classes by way of summary.

Class	Extends	In WS	Description
EGLoader	QDLLoader	--	The loader for this module.
EGModule	JavaModule	The module	The module class
ConcatFunction	QDLFunction	**`concat(x,y)`**	Simple concatenation function
ExtrinsicFunction	QDLFunction	**`$$identity(x)`**	Function whose name starts with $$
ExtrinsicVar	QDLVariable	**`$$EG`**	Variable whose name starts with $$
FEvalFunction	QDLFunction	**`f_eval(@f,x)`**	Example of a function that takes a function reference and how to resolve that and execute the function
StemVar	QDLVariable	**`eg.`**	A stem of various values so you can see how to set them
StemEntryVar	QDLVariable	**`a.5`**	A single entry of a stem, showing fine-grained control.

The extrinsic objects are simply that by virtual of the naming convention that they start with a $$ in their name. When the system detects this *on load*, it will add them to the workspace. Since you don not need to import the module to have access, you can put bootstrapping values in them, for instance.

Sample use

This is to illustrate how this works. The examples are included in the base distro so you can always do this

```
  X := j_load('tools.eg.basic')
  )funcs X
concat([2])  f_eval([2])
2 total functions
  )vars X
a.  eg.
  X#a.
{5:test value}
  X#f_eval(@cos, pi()/3)
0.500000000000002
```

This shows how to invoke a couple of sample functions. Don’t forget there is help availble too!

```
  )help X#f_eval 2
f_eval(@f, x) - simple basic that evaluates f at x using a function reference.
E.g.
    eg#f_eval(@cos, pi()/7)
0.900968867902419

This is the same as issuing cos(pi()/7)
```

Sample showing how extrinsics work

In this case, we just load the module and check that the extrinsics are availble.

```
  load(info().lib.tools.eg.basic, 'java')
ex:eg
  )vars -extrinsic
$$EG
  )funcs -extrinsic
$$echo([1])
1 total functions
  // Show that the function, which just echos the input, works.
  $$echo('mairzy doats and does eat stoats')
mairzy doats and does eat stoats
```

One of the basic problems with writing complex systems if the “bootstrapping” problem, meaning how to initialize and configure the system. If a module needs information specific to its import, then an extrinsic variable or function allows a way to do this. You do not need to include these ever, but the example is included since it would otherwise not be clear how to solve a bootstrapping issue. This is the Java analog of the standard construct in a module file of setting any extrinsics in the header before the module. So if this example were written as a QDL module the file would start as follows:

```
// extrinsic variables
$$echo(x)→x;
$$EG := 42;
module['ex:eg']
// … rest of module definition
```

The stateful example

This is in the package **`org.qdl_lang.extensions.examples.stateful`** and consists of a loader, module class and implementation class. The implementation contains the function and variables as subclasses. These share Java state and show the serialization mechanism.

Included classes

Class	Extends	In WS	Description
StatefulExample	QDLMetaModule		Implementation class
GetS	QDLFunction	**`get_string()`**	Inner class, sets a class variable
ImportTimestamp	QDLVariable	**`import_ts`**	Inner class, with the timestamp the module was imported

SetS	QDLFunction	`set_string(x)`	Inner class, gets the class variable,
StatefulLoader	QDLLoader	--	Loads the module
StatefulModule	JavaModule	The module	The module itself.

Example of using it.

Again, this module is included in the standard distribution, so you can load it and explore it as follows:

```
     Y := j_load('tools.eg.stateful')
  Y#set_string('Baby shark do-do-do-do-do-do')
null
  Y#get_string()
Baby shark do-do-do-do-do-do
```

The contract is that set_string returns the previous value. In this case, the previous value was unset, so it is null.

Saving and restoring

In this case, the module as a Java class has an internal variable. We first show the user experience since this is what we want. From the previous point, we save it.

```
  )save /tmp/temp.ws
saved: '/tmp/temp.ws'
  on: Sun Sep 08 07:41:34 CDT 2024
 compressed size: 1066
  elapsed time: 0.019 sec.
   )off y
system exit
```

Now we restart QDL and load the workspace:

```
$temp>qdl
  )load /tmp/temp.ws
  )vars
Y
  Y#get_string()
Baby shark do-do-do-do-do-do
```

Showing that the module was serialized by the system and on loading it, the state of the Java class was restored automatically.

Java class reference

Each of the above classes or interfaces is discussed here in detail. There are two basic ways to approach writing the functions and variables. The first if they are all independent, simply have them as separate

classes and add them to the module in your **`JavaModule`** extension. The second is to have them as inner classes of a meta class, where they can all share state. QDL provides an interface for this case, **`QDLMetaModule`**

QDLMetaModule

This interface is used if you have an enclosing class for all of your functions and variables that typically share some state. It has two methods that allow you to store your state on workspace save, so you may persist it.

```
JSONObject serializeToJSON();

void deserializeFromJSON(JSONObject json);
```

Example

These are taken from the HTTP implementation for QDL. That module tracks the host (the internet address the module is talking to) and the headers to be used.

```
@Override
public JSONObject serializeToJSON() {
    JSONObject json = new JSONObject();
    if (!StringUtils.isTrivial(host)) {
        json.put("host", host);
    }
    if (headers != null) {
        json.put("headers", headers);
    }
    return json;
}

@Override
public void deserializeFromJSON(JSONObject json) {
    if (json.containsKey("host")) {
        host = json.getString("host");
    }
    if (json.containsKey("headers")) {
        headers = json.getJSONObject("headers");
    }
}
```

The essential point is that the JSON object can be anything you like, since it is simply returned to you and then you unpack it. It would also be possible to have some other re-initializatio method(s) called in the deserializeFromJSON method.

JavaModule

This is the class that QDL thinks of as being the module. You extend this class and the only four methods that are required to override are

the no argument and URI argument constructors, which should call their respective **`super`** and

```
public Module newInstance(State state)
```

which will create a new instance of this module. If you want to supply a description for the module that can be displayed after loading the module, override

```
public List<String> getDescription()
```

The newInstance method

There is a method you must override in a module called

```
public Module newInstance(State state);
```

This will be called whenever **`import`** is called on your module, *i.e.*, to create a new instance. Points to remember are

1. create all **`QFLFunction`**s and **`QDLVariable`**s here. Add them to the module
2. Call **`init(State)`** as well as **`setupModule(Module)`** on your new module. This sets up the by creating the actual functions, variables and find it. If you do not call this, your module will not function once imported.
3. If your implementation of the module is QDLMetaModule, you must set that to enable serialization of the state.

Here is an annotated example. This is for the case that the implementing class is a QDLMetaModule

Java	Exegesis
`public class MyModule extends JavaModule{`	
`  public MyModule(){}`	required
`  public MyModule(URI namespace){super(namespace);}`	required
`@Override`	
`  public Module newInstance(State state) {`	
`    MyModule myModule =` `     new MyModule(URI.create("qdl:/examples/java"));`	This sets the namespace of the module
`    MyImpl myImpl = new MyImpl();`	In this case, the functions and variables of the class are inner classes of MyImpl
`    myModule.setMetaClass(myImpl);`	Set the *instance* of myImpl to be the meta module of this instance.
`  // Now add functions`	
`    List<QDLFunction> funcs = new ArrayList<>();`	
`    funcs.add(myImpl.new MyFunc());`	Remember this instantiates an inner class named MyFunc
`    myModule.addFunctions(funcs);`	
`// Add variables`	
`    List<QDLVariable> vars = new ArrayList();`	

vars.add(myImpl.new MyVar());	
myModule.addVariables(vars);	
myModule.init(state);	// adds the functions etc. to the state
setupModule(myModule);	// finishes accounting
return myModule;	
}	
} //end class	

The getDescription method

The method with signature:

```
public List<String> getDescription();
```

is optional in the sense that if you do not implement it, nothing is wrong. Its function is to give the module-level description that the user can peruse. This is available right after loading the module. So for instance

```
  )help ex:stateful
  module name : StatefulModule
    namespace : ex:stateful
default alias : null
   java class : org.qdl_lang.extensions.examples.stateful.StatefulModule
A stateful module example. This is intended for programmers who
are learning how to write their own QDL modules in Java. It shows
how to create an implementation class the contains inner classes which
are the functions and variables for the modules. The assumption is that
all of there share some state in the Java class (which is a priori unknown
to QDL) and must be serialized.
See the documentation https://qdl-lang.org/pdf/qdl_extensions.pdf
functions:
  get_string() - get the current string value.
variables:
  import_ts
```

The list of strings (here in italics) is inserted in the middle of the generated module description, here in italics. The rest of this description is from introspection on the module.

QDLLoader

This class is charged with telling the system what the modules are. Now there is an issue with the number of modules to return. Generally a loader returns a single class, since if you call it with the load function, that is unambiguous. However, it is also sometimes useful to return several loaders at once, if for instance you are intending that they be in the QDL configuration file and you want them just loaded but not somehow imported.

Complete example from the toolkit

The stateful example has this as the entire implementation:

```
public class StatefulLoader implements QDLLoader {
   @Override
   public List<Module> load() {
      ArrayList<Module> modules = new ArrayList<>();
      modules.add(new StatefulModule().newInstance(null));
      return modules;
   }
}
```

The QDLFunction interface

Functions must implement the `org.qdl_lang.extensions.QDLFunction` interface. This has a few methods (refer to the Java documentation). The basic way it works is that you tell how many arguments this may accept and when called, you will be given an array of objects which you must use. Note that in order to keep them straight you should specify in the function documentation what you expect the user to supply. You may name the class anything you like.

The getName method

```
public String getName();
```

This returns the name of the function. This is the name the QDL user sees.

The getArgCount method

```
public int[] getArgCount();
```

This returns an array of integers for the number of arguments the function accepts. Any non-neative argument may be used. The system will invoke your function when it encounters a call with one of these as the argument count.

The getDocumentation method

When the user involes)help *function_name arg_count* help is displayed. This isread directly from this method and printed. Remember that in many cases the system will take only the first line of the help and display it as a short form, so it is always a good idea to have the first line be concise.

```
List<String> getDocumentation(int argCount);
```

The evaluate method

The workhorse method of QDLFunction is

```
public Object evaluate(Object[] objects, State state);
```

In which an array of objects is passed as is the current state. Note that the objects are either constants or function references. You must check what the types are.

Example. A complete QDLFunction implementation

Here is one from the toolkit. It is very simple but shows all the parts.

```
public class ConcatFunction implements QDLFunction {
    @Override
    public String getName() {
        return "concat";
    }

    @Override
    public int[] getArgCount() {
        return new int[]{2};
    }

    @Override
    public Object evaluate(Object[] objects, State state) {
      return objects[0].toString() + objects[1]; // call toString so it compiles. Can't add
objects
    }

    @Override
    public List<String> getDocumentation(int argCount) {
        ArrayList<String> docs = new ArrayList<>();
        docs.add(getName() + "(string, string) will concatenate the two arguments");
        docs.add("This is identical in function to the built in '+' operator for two
arguments. It is just part of the");
        docs.add("sample kit for writing a java extension to QDL that is shipped with the
standard distro.");
        return docs;
    }
}
```

Of particular note is that the arg count is restricted to 2 arguments, This means that if the user attempts to call it with the wrong number, an error is generated.

```
   eg#concat('foo')
undefine function concat with one argument
```

QDLVariable interface

This has two methods:

The getName method

Returns the name of the variable. Note that for stems, you may include (constant) indices, allowing you to set individual elements of a stem.

```
String getName();
```

The getValue method

This is the value that will be assigned to the variable. Note that it must be or contain only QDL data types as per the table above. This is computed on module load and will not be updated, so it sets the initial value of the variable.

```
Object getValue();
```

An example

Here is a complete example that returns a set

```
public class MySet implements QDLVariable{
    @Override
    public String getName() {
        return "my_set";
    }

    @Override
    public Object getValue() {
        QDLSet set = new QDLSet();
        set.add(new BigDecimal("-34.234443"));
        set.add(42L);
        set.add("Sphinx of Black Quartz, Judge My Vow");
        return set;
    }
}
```

In this case, a set named my_set will be in a variable in the module.

Do you really need to create variables this way?

Strictly speaking, you do not. In the **`newInstance`** call for the module, you could just manually set variables in the state, providing it is not null, so the above example could be done as

```
  public Module newInstance(State state) {
   // .. other required stuff
   if(state != null){
        QDLSet set = new QDLSet();
        set.add(new BigDecimal("-34.234443"));
        set.add(42L);
        set.add("Sphinx of Black Quartz, Judge My Vow");
        state.setValue('my_set', set);
   }
  // rock on!
} // end of newInstance method
```

If you have a lot of variables to set, this might be a better option. However, if you are setting extrinsic variables, this works quite differently, so it is far easier to have a **`QDLVariable`** to define that.

Other topics

Allowed QDL data types vs. Java

These are the citizens of the module and are what the user interacts with. Each is just an interface to be implemented. The main method for each returns a value and the value must conform to a QDL type as per this table

QDL	Java	Comment
null	QDLNull.getInstance()	There is exactly one QDL null in the universe. Use it.
boolean	Boolean	There are two Boolean values.
integer	Long	The java object is used and all integer in QDL are 64 bit
decimal	BigDecimal	Other type such as double, float, etc. will reliably cause errors
stem	QDLStem	This include lists, which are special cases of stems.
set	QDLSet	All elements must be QDL variable types.
string	String	

Variables may be these or contain these, *e.g.* a stem of integers (*aka* Java Longs).

Intrinsic variables and functions

In standard QDL there is a privacy mechanism for making the state of a module immutable. In Java-based- modules, however, everything is private (as far as QDL can tell) unless explicitly revealed to the user, so there usually is no reason to define intrinsic items. Said more plainly, in a Java module everything is intrinsic unless you expose it.

Extrinsic variables and functions

Extrinsic functions an variables by convention start with a $$. If you create a variable with that name, then on load that becomes part of the current workspace. This is quite useful, since you can put bits of state, constants needed for bootstrapping etc. there and have it available before loading the module proper.

What if you really need this module to be a singleton? Then this is a style issue. You have it set as an extrinsic variable, *e.g.*,

```
   $$connection_pool := import(connection_pool);
```

which makes it available to every module and function in the workspace as soon as it is set. The point is that it is a lot easier to do this than have the module itself setup an extrinisic module on load since there is a bootstrapping issue (the module has to already exist in the workspace to set itself to be an extrinsic reference.)

Are modules classes?

No, in QDL there is no inheritance of modules. The is the namespace mechanism and encapsulation though.

Exceptions

You should just either BadArgException when checking arguments or otherwise throw regular Java exceptions as needed (e.g. IllegalStateException).

Example of how to use BadArgException(String, int) in practice

A typical example is checking for arguments at the start of a method. Here, the assumption is that object[1] must be a stem:

```
public Object evaluate(Object[] objects, State state) {
   if (!(objects[0] instanceof QDLStem)) {
      throw new BadArgException("The first argument of " +
                                          getName() + " must be a stem", 0);
   }
   QDLStem leftArg = (QDLStem) objects[0];
// … rest of method
}
```

Throwing BadArgException in this case means you pass in a message and the index of the bad argument. This is caught be the system and all of the information for the stack trace is filled in for you (since passing all pending state from everything calling this method in to the function would be very messy, hence it is done for you). This lets you get a tidy stack trace with the correct line numbers. Note there are several constructors for BadArgException, only use

```
BadArgException(String message, int argNumber);
```

Note: Aside from BadArgException *never* throw a QDLException or its subclasses. The reason is that QDLException contains information about the internal workings of QDL (such as line numbers and parsing information) and is intended to be used by QDL to give a report on the issue. If you throw one, then you may get some very strange errors as the system tries to figure out the error. In short for a general QDLException there are “no user serviceable parts.”

Java Serialization Guidelines

QDL *can* save its workspace using Java serialization. For certain types of complex state this can save resources (e.g. you have a million high precisions numbers in a stem). Modules are automatically serializable, but be sure that any classes you write implement the Serializable interface. If a module you wrote is not serializable, then you cannot save a workspace that references it. Do not forget that changes to your class may make it impossible to deserialize a workspace later, so you really must handle the serialization right if you intend to save your work this way. The best way to test this is at some point when you are testing your module inside a QDL workspace is just try to save the workspace and see what messages are produced.

There are any number of criticisms of Java's serialization mechanism and most of these boil down to the fact that it was never intended to be used in web traffic, so if it is possible to intercept a serialized object, then malicious code can be injected that will be executed on deserialization. As long as it is used for local operations or storage only (exactly what QDL does with it) it is a fine thing. You should, however, consider uses that sending serialized objects over the network (standard fix is to require an SSL connection for all network traffic), is strongly discouraged, but that is far outside the scope of what we are doing here.