Writing an Extension in Java

Version 1.0.3

Introduction

QDL has its own system for writing module, 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. That means that it has its own state and when you execute functions from a module, they execute there. All modules have a URI that identifies them uniquely and a human-readable alias that may be changed to suit the current user needs. Modules may not be nested in the sense that you cannot create a new one inside of one, but you can import them and use them.

Modules are also factories

There is a function you must override in a module called

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

This will be called whenever you need to create a new instance. Points to remember are

- create all QFLFunctions and QDLVariables here. Add them to the module
- If the state is not *null*, call init on your new module.

Here is an example (taken from the supplied toolkit) for such a method. The module, named *MyModule* is created

```
public class MyModule extends JavaModule{
  @Override
  public Module newInstance(State state) {
     MyModule myModule = new MyModule(URI.create("qdl:/examples/java"), "java");
     ArrayList<QDLFunction> funcs = new ArrayList<>();
     funcs.add(new Concat());
     myModule.addFunctions(funcs);

     ArrayList<QDLVariable> vars = new ArrayList<>();
     vars.add(new EGStem());
     myModule.addVariables(vars);
     if(state != null){
         myModule.init(state);
     }
```

```
    return myModule;
  }

}
```

The head scratcher here is why is this class creating a new instance of itself? Because this allows the module to produce copies of itself (could have had some sort of a *clone()* method instead too.) This method will be called whenever anyone *import* s this module with a different alias, allowing for multiple instances of this module.

Functions

Functions must implement the `edu.uiuc.ncsa.qdl.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.

Variables

Variables implement the `edu.uiuc.ncsa.qdl.extensions.QDLVariable` interface. This has two methods, one for the name and one for the value (as an Object).

Exposing the methods of a class as a function

A common construction is to have the module effectively be a single encapsulated class and the functions in it expose the underlying Java methods. In order to achieve this, you should have the class itself and then each QDLFunction will be a *non-static* inner class. Since non-static inner classes have access to the state of the enclosing class, this makes much coding quite easy. E.g.

```
public class MyClass{
   // whatever you need to make MyClass work. To expose the methods to QDL:
  public class MyQDLFunc implements QDLFunction{
       @Override
       public String getName(){
         return “my_func”;
   // ...all of the other implementations

  } // end myQDLFunc

}  // end MyClass
```

This means that this creates a function named *my_func* that QDL can use.

When you are loading this, you need to instantiate the class from the parent like this:

```
MyClass mc = new MyClass();
```

```
MyQDLFunc myFunc = mc.new MyQDLFunc();
```

Alternately you could have a each QDL function as its own class and a set of them could share a single instance of the class. This can be done in the QDLLoader for the module.

Are modules classes?

No. Not quite. They are encapsulated units. The difference is that a class you can pass around with a reference, modules you cannot (though you can pass along anything inside one.)

Serializability

Make sure that java serialization is handled right for any modules you create. This means that things (like loggers) that should not be serialized should be marked *transient*. QDL saves its workspace using Java serialization and this was chosen because otherwise every additional module needs to have a serialization mechanism designed and implemented – which can be a tremendous amount of work. If a module you wrote is not serializable, then you cannot save a workspace that references it. 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 it is possible to intercept a serialized object and inject malicious code that will be executed on deserialization. As long as it is used for local storage only (exactly what QDL does with it) it is a fine thing. You should, however, consider uses that send serialized objects over the network (standard thing is to require an SSL connection for all network traffic), but that is far outside the scope of what we are doing here.