Writing an Extension in Java

Version 1.3.1

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. 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, but we don't mostly to avoid running afoul of the contract for that method in the future.) 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.

Evaluating functions

The basic signature of a 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. Evaluating a function as an argument

You may pass function function references or lambdas. These are pre-processed and arrive as function reference nodes. To evaluate them, turn the reference into an Expression. For instance, let us say that you had a function whose signature (in QDL) is

```
f_ref(@f, x)
```

which is to evaluate f(x) and return the result. The zero-th aregument is therefore a function reference and the 1st argument is passed. You would write something like:

```
import static edu.uiuc.ncsa.qdl.evaluate.AbstractFunctionEvaluator.getOperator;

// .. other stuff in the class

@Override
public Object evaluate(Object[] objects, State state) {
    ExpressionImpl expression =
        getOperator(
           state, // current state
           (FunctionReferenceNode) objects[0], // cast as needed
```

```
            1); // valence. Here the 0th arg is monadic.
        expression.getArguments().add( new ConstantNode(objects[1])); // Set the arguments
        return expression.evaluate(state); // run it and return.
    }
```

so in QDL you could easily invoke this as

```
  f_ref(@cos, pi()/7)
0.900968867902419
```

Note that we cannot simply hand back the expression, just the function reference node. The reason for this is the `getOperator` function, which looks up the function based on the number of arguments it has. Since there is no way to tell ahead of time what the arguments are, you must resolve this.

Nota Bene: This exact example is included in the sample function

```
edu.uiuc.ncsa.qdl.extensions.example.FunctionReferenceExample
```

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). These are evaluated at load time and the value in the workspace is set.

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 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.

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 implement the marker interface **`edu.uiuc.ncsa.qdl.extensions.QDLModuleMetaClass`** 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 implements QDLModuleMetaClass{
   // 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 should implement the Serializable interface. The way QDL saves its state is via Java serialization (if we did not, you would have to write your own serialization/de-serialization mechanism for any extension you write.) See the note below for a fuller explanation of this.

This example then 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.

Synopsis

- `JavaModule` is the java analog of the definition `module[. . .]`. It has the variables, functions and documentation.
- `QDLModuleMetaClass` is an actual implementing class. Each variable or function is an inner class of this class. You don't need to set your module up as inner classes.
- `QDLLoader` is the class that `module_load` calls. Since this is user-facing, a good name is in order.

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.) Also, in QDL there is no inheritance of modules.

Exceptions

You should just throw regular Java exceptions as needed (e.g. IllegalStateException) and *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 QDLException there are "no user serviceable parts."

Serializability

Java Serialization Guidelines

Make sure that java serialization is handled right for any modules you create. The way that new instances of classes are done for you on the fly are via serialization (they are serialized then every new

instance of the module is deserialized as its own instance.) This means that things (like loggers) that should not be serialized should be marked *transient*. For the vast majority of implementations, you don't really need to do anything.

Saving workspaces with serialization

QDL *can* save its workspace using Java serialization. For certain types of complex state where you do not want to write the XML serialization code, this can work 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. 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 send serialized objects over the network (standard fix is to require an SSL connection for all network traffic), but that is far outside the scope of what we are doing here.

XML Serialization Guidelines

To support serialization to XML (which you should do really), you need to override the following as needed.

- `readExtraXMLAttributes` read the additional attributes to the module tag
- `writeExtraXMLAttributes` write attributes to the module tag
- `readExtraXMLElements` read custom elements inside the module element
- `writeExtraXMLElements` write custom elements inside the module element.

Please read the javadoc for these methods. Also, always call super! These also allow you to create instances of things or read configuration you stashed and re-initialize your objects. The aim is that if a user serializes their workspace, deserializing it gives back a fully functional set of objects.

State

The State object from QDL will be injected at runtime and each function will have the state at that instant injected into it in the evaluate method. This means that the state object should *not* be part of the enclosing class, but processed in the evaluate method where it is passed. Do not try to store State objects.