Path To QA Experience

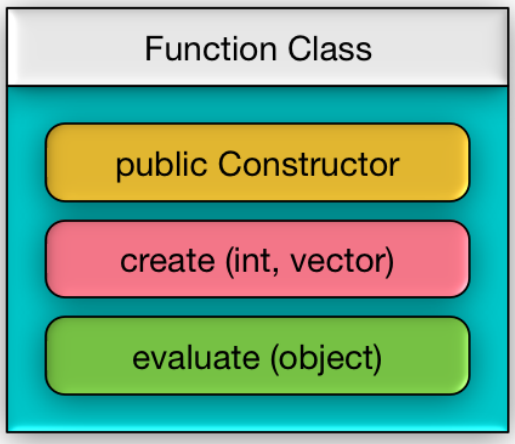
SEPTEMBER 10, 2015SEPTEMBER 14, 2015 / HARINIVAS GANAPATHY

Demystifying RIT Custom Function – Part 2: Putting the gears together

In **Part 1 (https://harinivasganapathy.wordpress.com/2015/09/06/demystifying-custom-functions/)** of the series, we discussed the list of pre-requisites and steps to set them up to begin developing Custom Functions. In Part 2 we would explore in detail about understanding the architecture of the Function class which is the backbone for any Functions in RIT.

Function Class Architecture

Any Function whether it is built-in or custom made – they are subclasses of Function Class available in *'com.ghc.ghTester.expressions'* package that we added to the target platform and manifest dependencies in **Part 1 (https://harinivasganapathy.wordpress.com/2015/09/06/demystifying-custom-functions/)**. It is important to understand the design of this Class because RIT lays down few rules that we have to follow for it to recognize, understand, load and execute our Function. Function Class has many methods, but three of them are very important and significant – the public constructor, create and evaluate method.



public Constructor : The way functions are designed in RIT mandates a default public constructor with no parameters. This is needed for loading our Custom Function class into RIT JVM. We don't need to add any computation inside this constructor. However it can be used to initialize instance variables.

create (int, vector) : This is a factory method that creates the instance of the particular Function with specific arguments we send during runtime. This method is invoked when RIT encounters the Custom Function syntax while evaluating a function action in tests. It is the responsibility of the *create* function to create, initialize and instantiate the specific Function object with matching parameters mentioned in the Function call in tests.

> Note that we can design and implement more than one Custom Function inside single Java Class file. And so it is the responsibility of the *create* method to initialize and instantiate the right Function with right parameters.

evaluate (Object) : This is the actual method that performs the computation of the Function we design and considered brain of our Custom Function Class. It is invoked by RIT after *create* method returns the second Function instance.

Rules of extending Function Class

1. Custom Function class should have a public default constructor.
2. Custom Function class should override *create (int, vector)* method.
3. Custom Function class should *evaluate (Object)* method.

Flow of Control

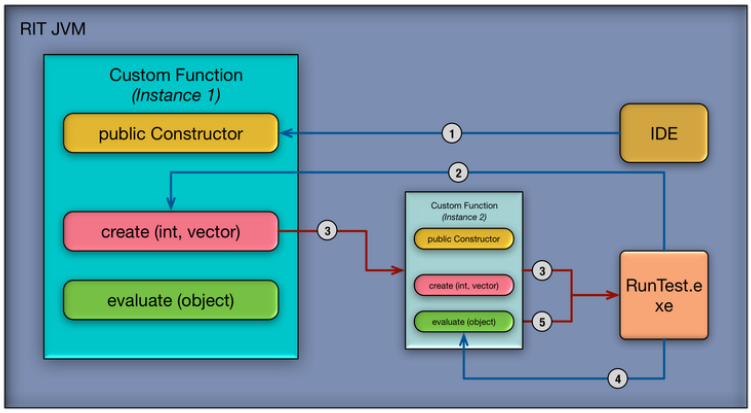
Step 1: When RIT starts up, it creates an instance of our Custom Function class using the default constructor and loads it to RIT JVM. For our reference we call it Instance 1. Once the Function successfully loaded, it will be available for us to use inside function action in tests.

Step 2: When RIT encounters the Custom Function syntax, while evaluating the function action in tests, it invokes the *create* method of the Instance 1 and sends two arguments in the call – (1) number of arguments in test function call as *int* and (2) all the arguments as a *vector*. *Create* method then creates another instance of the Custom Function and initializes it by calling the constructor with matching parameters based on the arguments received.

Step 3: Then it returns the newly created instance back to RIT with all data and parameters. For our reference we call this instance as instance 2.

Step 4: After receiving the instance 2 Custom Function object, RIT invokes the *evaluate* method of Instance 2 object and sends an Object as argument.

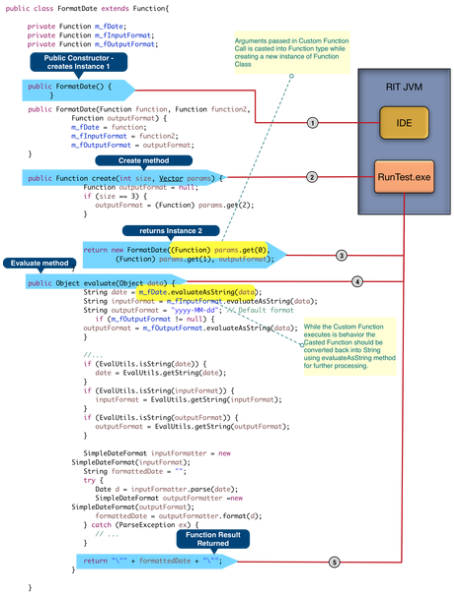
Step 5: On receiving the call, *evaluate* method has to discover the String data wrapped as an Function during the constructor call and assign it to the local variables for processing. Based on the received arguments, the evaluate method decides what to do. After performing all the computations it sends back the computed result back to RIT.



(https://harinivasganapathy.files.wordpress.com/2015/09/blog_rit_1.png)

Understanding using Code

Let us understand the architecture and flow of control we learnt so far by reviewing a sample code. For this purpose i have taken the example provided by IBM in its product documentation so it would easier when folks use the product documentation for further reference.



(https://harinivasganapathy.files.wordpress.com/2015/09/code_control_flow1.png)

Understanding the notion: *'Everything is Function'*

If you would take another look at the code, if not already noticed, you would realize that everything inside the Custom Function class either takes Function as argument or returns a Function object. Even the data members are type Function. The vector of parameters passed are also casted as Functions. Hence to use them for processing inside *evaluate* method the parameters as Function must be evaluated to discover the String value. For this purpose we use *evaluateAsString(Object)* method available in the Function Class.

Key Items for Consideration

1. There should be a public no argument constructor.

2. Class level variables should be created for each possible arguments the Custom Function is designed to take and each of the variables should be declared as type Function.

3. Class level Function variables must be initialized inside the corresponding constructor with parameters. Remember each parameters in Constructor are Functions so that they can be assigned to corresponding Class Variables.

4. There should be as many constructors with possible arguments a Custom Function call can have. For e.g. if a Custom Function is designed to take 2 or 3 or 4 arguments, then there should be 3 constructors with 2, 3 & 4 arguments respectively, so that create method would be able to use and instantiate the second instance. Or like in the IBM example single constructors can be used, however optional parameters must be evaluated and check for NULL like below –

```
Function outputFormat = null;
 if (size == 3) {
    outputFormat = (Function) params.get(2);
 }
```

5. *create* method should return only a new Instance of Custom Class with the String arguments stored as vector casted as Function before invoking corresponding constructor.

6. In *evaluate* method the actual String passed in Custom Function call in RIT test must be retrieved from the Function variables using *evaluateAsString* method available in the Function class.

7. *evaluate* method should return a String.

Now that we have got our core Custom Function Java Class ready, next step is to package it as a plug-in Jar and import it in RIT. We cover just these steps in our final Part – 3.

Continue to Part 3 (https://harinivasganapathy.wordpress.com/2015/09/13/demystifying-rit-custom-function-part-3-wrapping-up-for-deployment/)

Posted in **Rational Integration Testing**, **RIT** /Tagged **Custom Function**, **RIT** /**2 Comments**

2 thoughts on “Demystifying RIT Custom Function – Part 2: Putting the gears together”

1. Pingback: **Demystifying RIT Custom Function – Part 1: Gear up | Path To My Experience**
2.

Pingback: **Demystifying RIT Custom Function – Part 3: Wrapping up for Deployment | Path To My Experience**