Seminar Paper

Beyond JavaScript

Adding ooRexx and other JSR-223 Scripting Languages to the JavaFX WebView Control.

by

Maximilian Wannemacher

Matriculation Number: 01354328

Supervisor: ao.Univ.Prof. Dr. Rony G. Flatscher

Submission Date: 19.06.2019

Contents

1. Introduction

Software development has become a jungle of various programming languages and frameworks. For any given task, the best suited language can be chosen. If need be, more than one can be used to utilize the strengths of each. In web development however, this jungle has become a single tree. JavaScript is the predominant language on the client-side for everyone seeking to build a dynamic website. This prevalence of a single language creates various drawbacks and problems.

This paper seeks to provide a proof of concept for utilizing various scripting languages for web development. It is an attempt at diversifying the selection of languages. To this end, a simple browser application will be created using JavaFX WebView. Code contained within HTML <script> tags will be parsed and handled by Java ScriptEngines, which are part of the Java Scripting Framework proposed in JSR 223.

The application created will serve as a proof of concept, not a final product. It is meant as a nudge in a direction of varied languages within the web scripting context.

2. Background

2.1. The prevalence of JavaScript / ECMAScript

JavaScript was originally developed by Netscape in 1995 with the purpose of making HTML dynamic. Nowadays, it is used within various frameworks for all manners of applications. The primary purpose, however, remains the client-side scripting of websites. It has long been part of the trifecta of the web – HTML, CSS and JavaScript. All major web-browsers support JavaScript and as such it has become the de-facto agreed-upon scripting language of the web. Only Microsoft supported VBScript in addition until recently *(Windows Blogs: Disabling VBScript, 2019)*.

At the time of this writing, JavaScript is used by 95.2% of all websites for client-side scripting. *(W3Techs Usage Statistics, 2019)* As JavaScript is based on ECMAScript, development depends on the standards set by the Ecma Standards Organisation. This fact places a large amount of responsibilities in the hands of Ecma members, many of which are trying to further their own interests, among them tech giants like Google or Microsoft *(Ecma Ordinary Members, 2019)*.

2.2. Flexibility in Coding

2.2.1. Capabilities

No single language will always be the ideal choice for every task in a given project. Each language comes with its own ways of doing things, its own opportunities and challenges. Already there is a push to extend the capabilities of JavaScript. TypeScript for example seeks to create a JavaScript compatible language with strong typing. Adding more tools to a developer's toolboxes furthers their ability to create efficient and powerful code. Limiting them to just one language takes away the possibility of choosing the best-suited language for any given task.

2.2.2. Libraries

Modern coding paradigms and practices such as object-orientation are built around the idea, that elements of code can be reused. A problem that has been solved once, does not need to be solved again. Libraries are essentially many such code pieces are bundled together to allow for easy implementation and usage. There are extensive libraries available for JavaScript, the most-used among them being jQuery with over 90% of the most visited websites using it *(builtwith: jQuery, 2019)*.

Languages like Python offer their own set of libraries with sometimes different specializations. Access to these libraries would vastly extend the toolkit available for client-side web programming.

2.2.3. Preference

Another argument for language diversity is the simple factor of preference. A developer only familiar with Python would need to waste both time and effort on familiarizing himself with JavaScript in order to build websites. Furthermore, even a developer perfectly capable in JavaScript may prefer the syntax or certain elements of other scripting languages. Some languages are also easier to learn than others and can be used as an entry point into software or web development.

2.2.4. Drawbacks

This flexibility does not only come with advantages. Maintaining a code base with a lot of variety requires developers who are familiar with all the languages used.

The prevalence of JavaScript has also created a sizeable community, increasing the amount of cooperation and available support among web developers. There is also a large amount of documentation available for any web development task, created by multiple different organizations.

2.3. Java Scripting Framework (JSR 223)

The Java specification request 223 defines a standard framework for communication between Java and a number of scripting languages *(JSR 223, 2006)*. The API allows Java developers to implement scripts into their Java applications. This gives them the flexibility to make use of various advantages and features of scripting languages, such as dynamic typing, automatic type conversions etc.

The Scripting Framework uses so-called ScriptEngines to run script code. By default, the Java Scripting Framework come with Nashorn, a JavaScript engine. For this project, Jython (a Python Engine) as well as ooRexx are used in addition. However, the standards defined within JSR 223 should make it possible to implement any scripting language defined therein.

The application developed in this paper uses the framework to evaluate code directly and to invoke functions from Java at a later time. This allows for handling of different scripting languages, used here for displaying webpages with script tags defining languages other than JavaScript. The logic behind the scenes is done in pure Java, as it is easy to communicate between Java and the various scripting languages.

2.4. The <script> Tag in HTML

HTML denotes scripts to be executed within the context of a webpage by using the `<script>` tag. This tag provides the ability to differentiate between programming languages through the `type` attribute. The `type` attribute's value holds a MIME-type, with the media type being either `text` or `application`, and the subtype being the desired programming language. In the past the `language` attribute was used for this purpose, but it is now considered deprecated for use with the `<script>` tag *(MDN web docs, 2019)*.

In HTML5 the type attribute is not required, as it defaults to `application/javascript` *(W3org: HTML 5.2, 2017)*. Setting it to `application/python` causes the respective code to not be executed at all. Even if the browser attempted to run the script, it would at best only produce errors, since the browser is missing a built-in python interpreter.

The application developed for this paper will use the `<script>` tag to denote the language used. The IANA maintains an extensive list of registered MIME types and subtypes, however, this does not include other scripting language subtypes such as `application/oorexx` or `application/python` *(iana.org: MIME types, 2019)*. This is not a necessity, as the type attribute can hold any value and the handling of these new subtypes is controlled within the application itself.

```
<script type="text/python">
print("Hello world from python")
</script>
```

Excerpt of testfile.html

If any given `<script>` does not specify a `type`, the application will pass it on to the JavaScript engine, to ensure compatibility with preexisting `.html` files and parity with the HTML5 standard.

```
<script>
    print("Hello world! I defaulted to JavaScript");
</script>
```

Excerpt of testfile.html

Do note that the `print()` function is specific to Nashorn, the JavaScript engine provided with the Java Scripting Framework . The function converts its argument to a string and prints it to `stdout`.

2.5. JavaFX WebView

WebView is an embedded browser as part of JavaFX. It is based on the open source WebKit rendering engine and offers the following HTML 5 features:

- Canvas,
- Media playback,
- Form controls (except for `<input type="color">`),
- Editable content,
- History maintenance,
- Support for the `<meter>` and `<progress>` tags,
- Support for the `<details>` and `<summary>` tags,
- DOM,
- MathML,
- SVG,
- CSS,
- Javascript
- Support for domain names written in national languages.

(Wikipedia: JavaFX, 2019)

For the purpose of this paper, the native JavaScript support will be disabled in favor of an implementation through Nashorn. The main feature used for this application is DOM, as it allows for quick traversal of the target `.html` file and the underlying DOM tree.

3. Proof of Concept

3.1. Dependencies and Prerequisites

This application requires a set of dependencies and prerequisites to run correctly. The most important among them being a Java with both JavaFX and JSR 223 functionalities included. For DOM handling the `org.w3c.dom` package is used. For reading external files `java.io` is used.

The complete import statements of `Browser.java`:

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.concurrent.Worker;
import javafx.concurrent.Worker.State;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.layout.VBox;
import javafx.scene.web.WebEngine;
import javafx.scene.web.WebView;
import javafx.stage.Stage;
import org.w3c.dom.*;
import java.util.ArrayList;
import java.util.List;
import java.net.*;
import javax.script.*;
import java.io.*;
```

Excerpt of Browser.java

Furthermore, since this application relies heavily on the script engine functionality provided by JSR 223, several script engines for the desired languages must be included. This proof of concept provides support for the following scripting languages:

- Python via `jython 2.7.0`
- ooRexx via `Open Object Rexx (ooRexx) (100.20170923)`
- JavaScript via `Oracle Nashorn (1.8.0_191)`

To list all available scriptEngines on a given system, this simple application can be used. It retrieves all available engines and prints details about them.

```
import java.util.List;
import javax.script.*;
public class ListEngines {
    public static void main(String[] args)
    {
        System.out.println("Available ScriptEngines:");
        ScriptEngineManager sem = new ScriptEngineManager();
        List<ScriptEngineFactory> factoryList =
sem.getEngineFactories();
        for (int i = 0; i < factoryList.size(); i++)
        {
            ScriptEngineFactory sef = factoryList.get(i);
            String version = sef.getEngineVersion();
            String lName = sef.getLanguageName();
            String lVersion = sef.getLanguageVersion();
            System.out.println(sef.getEngineName());
            System.out.println("Version: "+version);
            System.out.println("Language: "+lName+
                               "     |     Version:     "+lVersion);
            System.out.println("-----");
        }
```

```
    }
}
```

Excerpt of ListEngines.java

3.2. Creating a simple Browser Application with Java WebView

Normal browsers do not come with engines for scripting languages other than JavaScript. The Java Scripting Framework alone cannot display web pages. It is therefore necessary for this application to provide browser functionality. To this end, JavaFX will be used to provide a user interface, whereas the browser functionality will be provided by WebView. This is by no means a complete browser, it merely provides the bare minimum needed to facilitate this proof of concept.

```
public class Browser extends Application {
    public void start(final Stage stage) {
        //parameters for browser window
        stage.setTitle("Browser");
        stage.setWidth(500);
        stage.setHeight(500);
        Scene scene = new Scene(new Group());
        VBox root = new VBox();
```

Excerpt of Browser.java

The first step is using JavaFX for the interface by creating a new `stage` as well as setting a few parameters for it. The `stage` is a container that represents the UI window. `setTitle()` changes the text within the title bar, while `setWidth()` and `setHeight()` set the horizontal and vertical measurements of the UI respectively. Creating a `Scene` provides us with another container for all future UI elements we want to add to our application. When using `Scene` one must also specify a `root` property. Here a `VBox` is used to align all future children vertically. This also prevents any clipping by the scene's `width` and `height` parameters. Later, the scene needs to be attached to

the stage, so the window has content to display. Since the stage's visibility is by default set to false, it needs to be toggled to display the elements.

```
stage.setScene(scene);
stage.show();
```

Excerpt of Browser.java

In order to provide browser functionality, a `WebView` with a `WebEngine` are used. To facilitate the display of larger websites, the contents of the `WebView` are added to a `ScrollPane`.

```
ScrollPane scrollPane = new ScrollPane();
scrollPane.setContent(browser);
```

Excerpt of Browser.java

JavaScript is then disabled on the WebEngine. While it would be perfectly acceptable to use, this application attempts to show off correct handling of various scripting languages. As such, native JavaScript support is turned off to later be provided by the application itself. Doing both would result in the code being run twice.

```
//disable webEngine JS to prevent scripts from being run twice
webEngine.setJavaScriptEnabled(false);
```

Excerpt of Browser.java

In order to run code within `.html` files, the website needs to be fully loaded first. The WebEngine `LoadWorker` has a `state` property, to which a `ChangeListener` is attached. With the listener in place, `getScripts` is only called once the `.html` file is fully loaded.

```
webEngine.getLoadWorker().stateProperty().addListener(
        new ChangeListener<State>() {
            @Override public void changed(ObservableValue ov, State
                                          oldState, State newState){
                if (newState == Worker.State.SUCCEEDED) {
                    stage.setTitle(webEngine.getLocation());
                    getScripts(webEngine.getDocument());
                }
            }
        });
```

Excerpt of Browser.java

To test functionality, a local test file is loaded. First, the URL is retrieved and stored using a relative resource name. It is then loaded in the webEngine using a string representation of the URL. Alternatively, a non-local file could have been loaded. The entire `scrollPane` is then added to the `VBox` and the scene is set.

```
//loads local testfile
URL url = getClass().getResource("testfile.html");
System.out.println(url);
webEngine.load(url.toExternalForm());

// it is also possible to load websites using
// webEngine.load("https://www.iana.org/");

root.getChildren().addAll(scrollPane);
scene.setRoot(root);

stage.setScene(scene);
stage.show();
```

Excerpt of Browser.java

Finally, we create an instance of the application on the JavaFX application thread by calling `Application.launch()` within the main method.

```
public static void main(String[] args) {
    launch(args);
}
```

Excerpt of Browser.java

3.3. Context and Bindings

Web developers rely on various preexisting bindings that exist within the browser context. Any scripts executed within the ScriptEngine context will not have access to things such as the `document` object, which serves as the entry point to the document object model.

In order to access the aforementioned objects within the ScriptEngine context, bindings have to be created. All ScriptEngines are handled by the ScriptEngineManager. Therefore, any binding set with `ScriptEngineManager.put()` will be within the Global Scope and therefore accessible by all future ScriptEngines. `ScriptEngineManager.put()` uses a key/value pair, so whenever `document` is accessed in the future, the stored value is retrieved.

```
getScripts(webEngine.getDocument());

...

private void getScripts(Document doc) {
    ScriptEngineManager sem = new ScriptEngineManager();
    //make document available to all scriptEngines
    sem.put("document", doc);
```

Excerpt of Browser.java

3.4. Using ScriptEngines to Evaluate Code

With the ScriptEngineManager already created, the next step is to gather all `<script>` tags into a collection. Using a for loop, every element of this collection is then evaluated using the `runScript` function. The return value of this function is a ScriptEngine which will later be used to properly handle DOM-events. As arguments, the function receives both a node and the pre-existing ScriptEngineManager `sem.`

```
//get all scripts
NodeList scripts = doc.getElementsByTagName("script");
...
for(int i = 0; i < scripts.getLength(); i++){
     engineList.add(runScript(scripts.item(i), sem));
}
```

Excerpt of Browser.java

The node is then searched for the `type` attribute. Should no type attribute exist, `getNamedItem` will return `null`. In this case, handling for this `<script>` node defaults to JavaScript.

```
private ScriptEngine runScript(Node script, ScriptEngineManager sem)
{
    NamedNodeMap attributes = script.getAttributes();
    Node type = attributes.getNamedItem("type");
    String textContent = "";

    if(type == null){
        textContent = "text/javascript";
    }
    else{
        textContent = type.getTextContent();
    }
```

Excerpt of Browser.java

If a type attribute is found, the `textContent` is retrieved and then split between MIME-type and subtype using a simple regular expression. As it is assumed that either `application` or `text` are used in a `<script>` tag, only the subtype is relevant for correct handling of the script. `split` returns an array of strings of which only index 1 (the subtype) is used.

```
textContent = textContent.split("/")[1];
```

Excerpt of Browser.java

The application then prints out a short string for debugging purposes, stating the language of the script found. This language is equal to the content of the type attribute. Do note that Java ScriptEngines can be created by multiple aliases. For example: passing either `JS` or `javascript` to the `getEngineByName` function would both be equally functional.

The preceding `if` statement is an attempt at filtering out scripts that cannot be handled by the application. While theoretically possible, this would be outside of the scope of a

simple proof of concept and is therefore left for future iterations of this application. Besides `json` there are also `template` which has caused problems on live websites.

To run or evaluate the `script`, the ScriptEngineManager creates a new ScriptEngine, using the isolated type from before. Every engine created by this ScriptEngineManager can also access the `document` binding created earlier and can use it as an entry point to the document object model. Finally, the `eval` function is used to run the script, catching any `ScriptException` in the process. The `ScriptException` is created by the corresponding `ScriptEngine` and the structure of the content varies between implementations which makes handling of specific errors difficult.

```
System.out.println("Script of type " + textContent + " detected.");
if(!textContent.contains("json")) {
    ScriptEngine se = sem.getEngineByName(textContent);
    String scriptCode = script.getTextContent();

    ...

    try {
        se.eval(scriptCode);
    }
    catch (ScriptException e) {
        System.err.println(e);
    }
    return se;
}
```

Excerpt of Browser.java

Lastly, the `ScriptEngine` is returned, to later provide the ability to invoke any functions within that script.

3.5. External Files

Script are often not written inline within `.html` files. In such cases, these scripts come in external files. This application, for example, contains an external JavaScript file with the name `external_file.js`. To access those files from within HTML, the `src` attribute is used within `<script>` tags.

```
<script type="text/javascript" src="external_file.js"></script>
```

Excerpt of testfile.html

To check for any external files, the script node gets searched for any `src` attributes before evaluating the scripts inside. If any are found, a FileReader is used to evaluate the entire file, throwing an exception if the file could not be found.

```
//handling for external files
    Node src = attributes.getNamedItem("src");

    ...

    if(src != null) {
        try {
            se.eval(new FileReader(src.getTextContent()));
        }
        catch(FileNotFoundException|ScriptException e){
            System.err.println(e);
        }
    }
```

Excerpt of Browser.java

3.6.Handling DOM-Events

One common practice of web development has been the usage of DOM-events. In this case, a (usually short) piece of JavaScript code is executed on a specific event attached to a DOM-element. Most commonly, these pieces of code call functions.

```
<button id=JS onclick="testFunctionJS(this)">0</button>
```

Excerpt of testfile.html

In this case, every time the button is clicked, `testFunctionJS` is called from within a different `<script>` node. To handle these events, all DOM-events need to be identified first. This program currently only handles `onclick` events to demonstrate that it is indeed possible. Other events should be possible to implement using this method, as long as a corresponding Java EventHandler exists.

As every element within the DOM tree can have an event attached, all nodes need to be checked for `onclick` attributes. If such an attribute has been found, a Java `org.w3c.dom.events.EventListener` is attached to it. The fully qualified name is used to avoid conflict with other `EventListeners`, such as the ones provided by JavaFX.

Just like the HTML and JavaScript equivalent, this listener handles all click events and is attached to the same DOM-element.

As soon as the condition for the event triggers, `handleEvent` is called.

```
//get all dom nodes
NodeList allNodes = doc.getElementsByTagName("*");

...

for(int i = 0; i < allNodes.getLength(); i++) {
    Element element = (Element) allNodes.item(i);
    if (element.hasAttribute("onclick")){
        //TO-DO: add other DOM events
        //create new eventlistener
        org.w3c.dom.events.EventListener listener = new
        org.w3c.dom.events.EventListener() {
            public void handleEvent(Event ev) {

            ...

            }
        };
        //add eventlistener to element
        org.w3c.dom.events.EventTarget et =
        ((org.w3c.dom.events.EventTarget) element);
        et.addEventListener("click", listener, false);
}
```

Excerpt of Browser.java

3.6.1. Isolating Functions within DOM-Elements

`handleEvent` first isolates function name and any arguments. Currently, only one argument is supported. However, implementation of further arguments should be a trivial task for future iterations.

Furthermore, only function calls in the format of `functionName()` with optional arguments are possible at the moment. Other scripting languages also allow for different syntax for function calls. In ooRexx for example functions can be called with `call functionName`. While it is possible to invoke functions from any language, special syntaxes have not yet been considered. This is currently the only language-specific element of the program, that could theoretically prevent the addition of any language. This could be circumvented by strictly requiring the `functionName()` syntax in the future.

To isolate argument from function, once again a split with a simple regular expression is used. Using the bracket as a separator requires escape characters. The argument still has a closing bracket at the end of the string, which is removed with `substring`.

```
public void handleEvent(Event ev) {

    //isolate function name
    String event = element.getAttribute("onclick");
    String func = "";
    //isolate arguments
    //TO-DO: multiple arguments
    //TO-DO: other language syntax
    //TO-DO: multiple functions
    String arg = "";
    if(event.contains("(")) {
        String[] seperated = event.split("\\(");
        func = seperated[0];
        arg = seperated[1].substring(0, seperated[1].length() - 1);
    }
```

Excerpt of Browser.java

To allow for multiple languages inside the DOM-events, the correct `ScriptEngine` needs to be used. This presents a problem, as the `type` attribute is specific to certain

elements. A button element for example has three different types: `button`, `submit` and `reset`.

With the `type` being unreliable, the deprecated script attribute `language` could be used instead, but a button for example does not allow for this attribute. It is therefore necessary to return every `ScriptEngine` when running a script and saving it in an `ArrayList`, to iterate through them later.

```
List<ScriptEngine> engineList = new ArrayList<>();
...
engineList.add(runScript(scripts.item(i), sem));
```

Excerpt of Browser.java

3.6.2. Invoking Functions

By iterating through this list of engines, the program checks if it can invoke a function carrying the name isolated before. This uses the `Invokable` interface defined within the Java Scripting Framework. The `Invocable` interface allows for script function calls from within Java.

This solves another problem. Normally, invoking functions from other languages would not be possible. By utilizing `invokeFunction` a function call can come from any language, as long as it is recognized as such and will be handled accordingly.

```
for(int en = 0; en < engineList.size();) {
    //try invoking the code first
    Invocable invocable = (Invocable) engineList.get(en);
    try {
        if(arg.isEmpty()){
            invocable.invokeFunction(func);
        }
        else{
            //make element referencable via "this"
            if(arg.contains("this")){
                invocable.invokeFunction(func, element);
            }
            else{
                invocable.invokeFunction(func,
                                         engineList.get(en).get(arg));
            }
        }
        break;
```

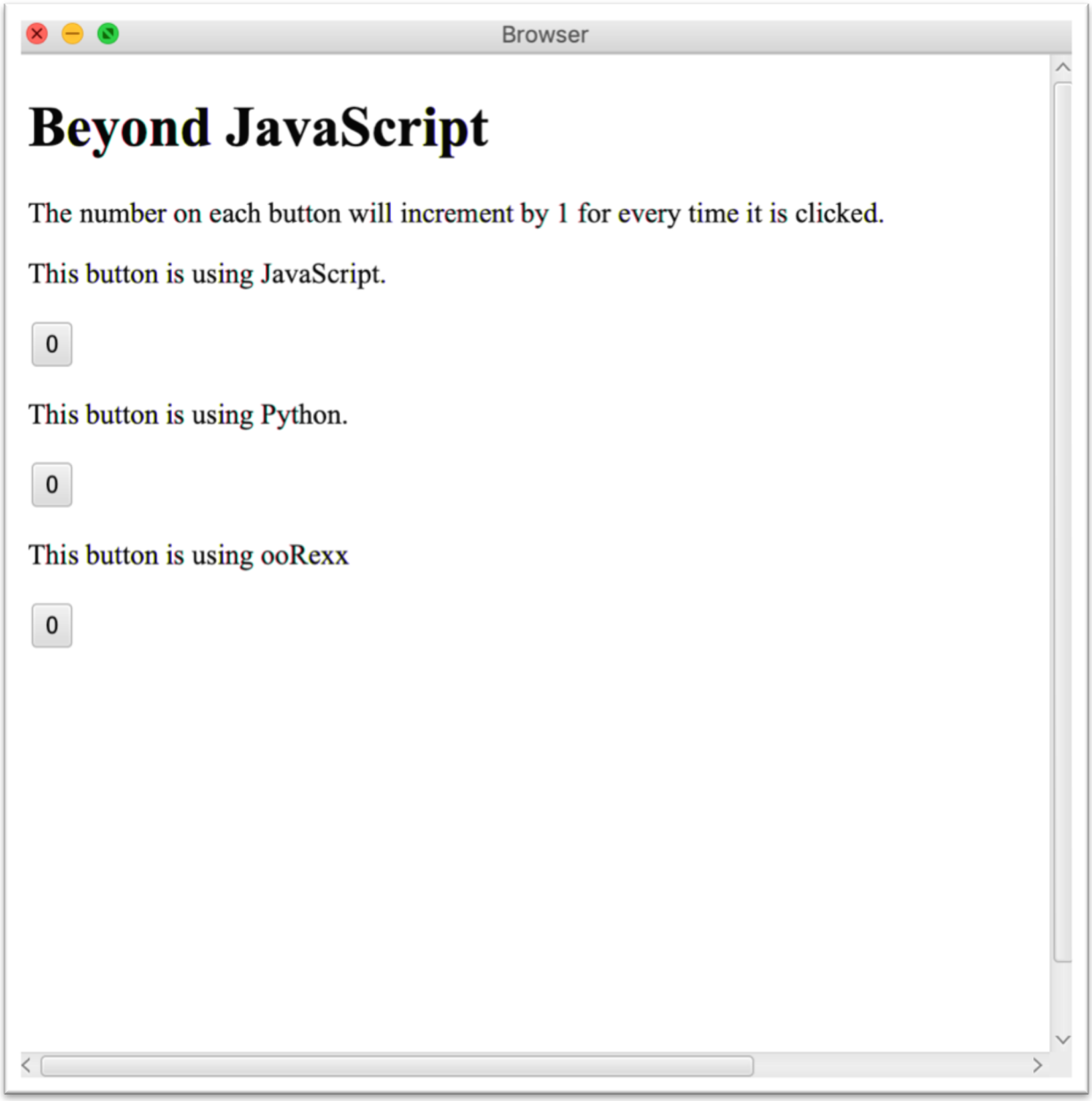
Excerpt of Browser.java

In many cases, these functions also pass on the `this` keyword as an argument. Here, the `this` keyword references the HTML element to which the DOM-Event is attached, in this case a `<button>` *(W3Schools: This, 2019)*.

As every script is handled by a different `ScriptEngine`, passing the `this` object to other engines becomes necessary. This is done by giving the element to the invoked function as an argument. If the argument does not contain `this`, the text content gets passed on instead.

4. Testing Functionality

For the purpose of testing of the features described within this paper, a simple .html file was created. The following image shows the applications user interface.



Screenshot of the Browser.java UI

The .html contains three buttons and five main `<script>` tags.

4.1. Defaulting to JavaScript

The first script attempts to test the correct handling of <script> tags when the type attribute is missing.

```
<script>
    print("Hello world! I defaulted to JavaScript");
</script>
```

However, due to the ambiguity of languages, this could be both JavaScript as well as Python. Even if the console shows the correct output, it is not guaranteed to actually be evaluated by the JavaScript engine. Adding an output for the engine name in `Browser.java` shows that the code is indeed run with the Nashorn Engine.

```
se.eval(scriptCode);
//output engine name for testing purposes
System.out.println(se.getFactory().getEngineName());
```

Excerpt of Browser.java

```
Script of type javascript detected.
Hello world! I defaulted to JavaScript
Oracle Nashorn
```

Output of Browser.java

4.2. Testing Languages and Invoking Functions

4.2.1. JavaScript

Once again, adding an output of the engine name whenever a function is invoked (as opposed to a script evaluated) shows the correct engine in `stdout`.

```
Script of type javascript detected.
Hello world from JavaScript!
Oracle Nashorn
```

Output of Browser.java

```
//output engine name for testing purposes
System.out.println(engineList.get(en).getFactory().getEngineName());
```

Excerpt of Browser.java

Upon evaluation of the script, the correct output is shown in `stdout`.

```
<script type="text/javascript">
    print("Hello world from JavaScript!");
    var clickCount = 0;
    function testFunctionJS(arg){
        clickCount += 1;
        arg.textContent = clickCount;
    }
</script>
```

Excerpt of testfile.html

```
Script of type javascript detected.
Hello world from JavaScript!
Oracle Nashorn
```

Output of Browser.java

Furthermore, every time the corresponding JS button is clicked, the number as expected increments by one and `Oracle Nashorn` is written to the output.

It is worth noting, that Nashorn does not understand `console.log`, as it does not evaluate the script within a browser context. It would be possible to create a binding

redirecting `console.log` to `stdout`. For testing purposes the `print` function emulates this behavior.

4.2.2. Python

The same test can be repeated for Python with slightly different syntax. Here the `global` keyword needs to be used to make `clickCount` available outside of the function scope.

```
<script type="text/python">
print("Hello world from Python!")
clickCount = 0
def testFunctionPy(arg):
    global clickCount
    clickCount += 1
    arg.textContent = str(clickCount)
</script>
```

Excerpt of testfile.html

Worth noting is, that this is where the syntax highlighting of the development environment failed, as no language other than JavaScript is expected within a `<script>` tag.

Again, this test concludes with the correct results.

```
Script of type python detected.
Hello world from Python!
jython
```

Output of Browser.java

And clicking the button increments by one and correctly prints.

```
jython
```

Output of Browser.java

4.2.3. ooRexx

Repeating this step for ooRexx requires another change in the syntax. For clickCount the local environment object is used, so it is accessible within the routine. The keyword public needs to be added to the routine, as the application tries to access it from the outside via Java. The message based nature of ooRexx requires sending messages to the button object. Thanks to the ScriptEngine, the object understands these messages and produces the correct output as expected.

```
<script type="text/oorexx">
say "Hello world from ooRexx!"
.local~clickCount = 0
exit
::ROUTINE testFunctionRexx public
    button = arg(1)
    .local~clickCount += 1
    button~textContent = .local~clickCount
    return
</script>
```

Excerpt of testfile.html

```
Script of type oorexx detected.
REXXout>Hello world from ooRexx!
Open Object Rexx (ooRexx)
```

Output of Browser.java

Do note that as part of the implementation of the ooRexx `ScriptEngine`, all outputs written to `stdout` through ooRexx scripts are preceded by `REXXout>`. Clicking the corresponding button increments the number by one and prints the name of the engine.

```
Open Object Rexx (ooRexx)
```

Output of Browser.java

4.3. Testing External Files

For testing if external scripts are correctly evaluated, a file called external_file.js is created. It only has one line of code.

```
print("Hello from external_file.js");
```

external_file.js

It is run via the `src` attribute within the last `<script>` tag.

```
<script type="text/javascript" src="external_file.js"></script>
```

Excerpt of testfile.html

The output correctly reads.

```
Hello from external_file.js
Oracle Nashorn
```

Output of Browser.java

5. Discussion

5.1. Limitations and Future Improvements

There exist many oddities in the implementation of JavaScript in web development and covering all of them would be no easy undertaking. Further improvements should focus on eliminating the missing features from JavaScript, one of more important among them being the implementation of additional DOM events. Modifying the application in such a way that it would be usable as a browser replacement would reveal most of the missing features in live testing.

A non-exhaustive list of features not yet implemented:

- The `javascript:` prefix, i.e.
  `<a href="javascript:func(this)">here</a>`

- Handling of other file types such as JSON

- Passing multiple arguments when invoking functions, i.e.
  `<button onclick="func(1, 2)"></button>`

- Calling multiple functions on events ,i.e.
  `<button onclick="func1();func2();">`

- Adding other language syntax for function invocation, i.e.
  `<button onclick="call func1">`
  or alternatively finding a way that is not language specific.

Future research could also touch upon topics such as security and efficiency.

5.2. Conclusion

Browser.java serves a minimal browser application that can reliably evaluate script code of multiple languages defined within JSR 223. As such, it serves as a platform for language independent web scripting. The current implementation is not perfect, but it can serve as a stepping stone for future improvement.

This should serve as a proof of concept that JavaScript is far from being the only language potentially suitable for web scripting. This opens up new opportunities for developers to use preexisting toolkits in a new environment.

It also opens up web development for developers without knowledge of JavaScript. This is especially useful for beginners, with websites also offering a way to instantly see what you created. This is where languages with simple syntax such as ooRexx could come into play. While this project may not topple JavaScript as the undisputed king of web scripting, maybe it can bring a bit of diversity.

6. Bibliography

builtwith: jQuery. (2019, June 10). Retrieved from https://trends.builtwith.com/javascript/jQuery

Ecma Ordinary Members. (2019, June 10). Retrieved from Ecma International: https://www.ecma-international.org/memento/ordinary.htm

iana.org: MIME types. (2019, June 10). Retrieved from https://www.iana.org/assignments/media-types/media-types.xhtml

JSR 223. (2006, July 31). Retrieved from https://jcp.org/aboutJava/communityprocess/final/jsr223/index.html

MDN web docs. (2019, June 10). Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/lang

W3org: HTML 5.2. (2017, December 12). Retrieved from https://www.w3.org/TR/html5/scripting-1.html#attr-script-type

W3Schools: This. (2019, June 10). Retrieved from https://www.w3schools.com/js/js_this.asp

W3Techs Usage Statistics. (2019, June 10). Retrieved from W3Techs.

Wikipedia: JavaFX. (2019, June 14). Retrieved from https://en.wikipedia.org/w/index.php?title=JavaFX&oldid=901776939

Windows Blogs: Disabling VBScript. (2019, June 10). Retrieved from Windows Blogs: https://blogs.windows.com/msedgedev/2017/04/12/disabling-vbscript-execution-in-internet-explorer-11/

Appendix A: Browser.java

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.concurrent.Worker;
import javafx.concurrent.Worker.State;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.layout.VBox;
import javafx.scene.web.WebEngine;
import javafx.scene.web.WebView;
import javafx.stage.Stage;
import org.w3c.dom.*;
import org.w3c.dom.events.Event;
import java.util.ArrayList;
import java.util.List;
import java.net.*;
import javax.script.*;
import java.io.*;

/*
 * This application was developed during the 4201 Seminar aus BIS
 * at the Vienna University of Economics and Business. It serves
 * as a proof of concept, that in theory JavaScript is not the only
 * language capable of web scripting.
 *
 * It does so by utilizing the Java Scripting Framework to load testfile.html
 * evaluating code from three different Scripting languages in the process.
 *
 * Thanks to everyone who helped me along the way, but especially
 * - ao.Univ.Prof. Dr. Rony G. Flatscher for the idea and additional help
 * - Thomas Weber for valuable input and moral support
 * - the many creators of ScriptEngines and the Java community in general
 */

public class Browser extends Application {
    public void start(final Stage stage) {
        //parameters for browser window
        stage.setTitle("Browser");
        stage.setWidth(600);
        stage.setHeight(600);
        Scene scene = new Scene(new Group());
        VBox root = new VBox();

        final WebView browser = new WebView();
        final WebEngine webEngine = browser.getEngine();

        ScrollPane scrollPane = new ScrollPane();
        scrollPane.setContent(browser);

        //disable webEngine JS to prevent scripts from being run twice
        webEngine.setJavaScriptEnabled(false);

        webEngine.getLoadWorker().stateProperty().addListener(
                new ChangeListener<State>() {
```

```
                    @Override public void changed(ObservableValue ov, State
oldState, State newState) {
                        if (newState == Worker.State.SUCCEEDED) {
                            getScripts(webEngine.getDocument());
                        }
                    }
                });

        //loads local testfile
        URL url = getClass().getResource("testfile.html");
        System.out.println(url);
        webEngine.load(url.toExternalForm());

        // it is also possible to load websites using
        // webEngine.load("https://www.iana.org/");

        root.getChildren().addAll(scrollPane);
        scene.setRoot(root);

        stage.setScene(scene);
        stage.show();
    }


    private void getScripts(Document doc) {
        ScriptEngineManager sem = new ScriptEngineManager();
        //make document available to all scriptEngines
        sem.put("document", doc);

        System.out.println("Document loaded");

        //get all scripts
        NodeList scripts = doc.getElementsByTagName("script");
        //get all dom nodes
        NodeList allNodes = doc.getElementsByTagName("*");

        List<ScriptEngine> engineList = new ArrayList<>();

        for(int i = 0; i < scripts.getLength(); i++){
            engineList.add(runScript(scripts.item(i), sem));
        }


        for(int i = 0; i < allNodes.getLength(); i++) {
            Element element = (Element) allNodes.item(i);
            if (element.hasAttribute("onclick")) //TO-DO: add other DOM events
            {
                //create new eventlistener
                org.w3c.dom.events.EventListener listener = new
org.w3c.dom.events.EventListener() {
                    public void handleEvent(Event ev) {

                        //isolate function name
                        String event = element.getAttribute("onclick");
                        String func = "";
                        //isolate arguments
                        //TO-DO: multiple arguments
                        //TO-DO: other language syntax
                        //TO-DO: multiple functions
```

```
                        String arg = "";
                        if(event.contains("(")) {
                            String[] seperated = event.split("\\(");
                            func = seperated[0];
                            arg = seperated[1].substring(0, seperated[1].length() -
                                                        1);
                        }

                        for(int en = 0; en < engineList.size();) {
                            //try invoking the code first
                            Invocable invocable = (Invocable) engineList.get(en);
                            try {
                                if(arg.isEmpty()){
                                    invocable.invokeFunction(func);
                                }
                                else{
                                    //make element referencable via "this"
                                    if(arg.contains("this")){
                                        invocable.invokeFunction(func, element);
                                    }
                                    else{
                                        invocable.invokeFunction(func,
                                                    engineList.get(en).get(arg));
                                    }
                                    //output engine name for testing purposes

System.out.println(engineList.get(en).getFactory().getEngineName());
                                }
                                break;

                            } catch (NoSuchMethodException nsm) {
                                if (en == engineList.size()-1){
                                    System.err.println(nsm);
                                }
                                en++;
                            } catch (ScriptException e) {
                                System.err.println(e);
                                break;
                            }
                        }
                    }
                };

                String script = element.getAttribute("onclick");

                //add eventlistener to element
                org.w3c.dom.events.EventTarget et =
((org.w3c.dom.events.EventTarget) element);
                et.addEventListener("click", listener, false);
            }
        }
    }


    private ScriptEngine runScript(Node script, ScriptEngineManager sem) {
        NamedNodeMap attributes = script.getAttributes();
        Node type = attributes.getNamedItem("type");
        String textContent = "";
```

```
        if(type == null){
            textContent = "text/javascript";
        }
        else{
            textContent = type.getTextContent();
        }

        //search for scriptengine according to type attribute
        //regex splits MIME-type and subtype
        textContent = textContent.split("/")[1];

        System.out.println("Script of type " + textContent + " detected.");
        if(!textContent.contains("json")) {
            ScriptEngine se = sem.getEngineByName(textContent);
            String scriptCode = script.getTextContent();
            Node src = attributes.getNamedItem("src");
            //handling for external files
            if(src != null) {
                try {
                    se.eval(new FileReader(src.getTextContent()));
                }
                catch(FileNotFoundException|ScriptException e){
                    System.err.println(e);
                }
            }

            try {
                se.eval(scriptCode);
                //output engine name for testing purposes
                System.out.println(se.getFactory().getEngineName());
            }
            catch (ScriptException e) {
                System.err.println(e);
            }
            return se;
        }

        return null;
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Appendix B: ListEngines.java

```
import java.util.List;
import javax.script.*;

public class ListEngines {
    public static void main(String[] args)
    {
        System.out.println("Available ScriptEngines:");
        ScriptEngineManager sem = new ScriptEngineManager();
        List<ScriptEngineFactory> factoryList = sem.getEngineFactories();
        for (int i = 0; i < factoryList.size(); i++)
        {
            ScriptEngineFactory sef = factoryList.get(i);
            String version = sef.getEngineVersion();
            String lName = sef.getLanguageName();
            String lVersion = sef.getLanguageVersion();

            System.out.println(sef.getEngineName());
            System.out.println("Version: "+version);
            System.out.println("Language: "+lName+" | Version: "+lVersion);
            System.out.println("-----");
        }
    }
}
```

Appendix C: testfile.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<br>

<h1 id="test">Beyond JavaScript</h1>

<p>The number on each button will increment by 1 for every time it is clicked.</p>
</br>
<p>This button is using JavaScript.</p>
<button id=JS onclick="testFunctionJS(this)">0</button>

<p>This button is using Python.</p>
<button id=PY onclick="testFunctionPy(this)">0</button>

<p>This button is using ooRexx</p>
<button id=OR onclick="testFunctionRexx(this)">0</button>

<script>
    print("Hello world! I defaulted to JavaScript");
</script>

<script type="text/javascript">
    print("Hello world from JavaScript!");
    var clickCount = 0;
    function testFunctionJS(arg){
        clickCount += 1;
        arg.textContent = clickCount;
    }
</script>

<script type="text/python">
print("Hello world from Python!")
clickCount = 0
def testFunctionPy(arg):
    global clickCount
    clickCount += 1
    arg.textContent = str(clickCount)
</script>

<script type="text/oorexx">
say "Hello world from ooRexx!"
.local~clickCount = 0
exit
::ROUTINE testFunctionRexx public
    button = arg(1)
    .local~clickCount += 1
    button~textContent = .local~clickCount
    return
</script>

<script type="text/javascript" src="external_file.js"></script>
```

```
</body>
</html>
```

Appendix D: external_file.js

```
print("Hello from external_file.js");
```