

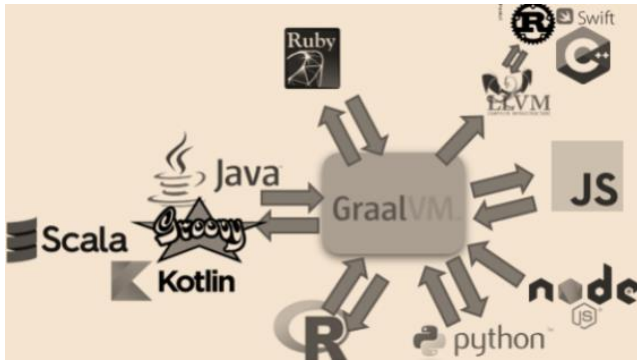


Workshop Graal VM Polyglot Interoperability

Adnan Drina
Lucas Jellema

INTRODUCTION

This workshop provides an introduction to GraalVM's Polyglot Interoperability features. GraalVM can run Java applications – and also JavaScript, Python, Ruby, R and many other types of applications. Additionally, GraalVM allows languages written in one of these languages to interact with code written in the other languages. At runtime, the logic executed in a single application process can originate from several different sources.



The labs in this document lead you through these main areas:

- Java applications executing JavaScript
- Java applications interacting with other “Truffle” languages
- Creating a native image (stand-alone executable) for a Java polyglot application
- Node & JavaScript applications interacting with Java code
- Polyglot multi-directional interoperability

The documents and sources can be found on <https://github.com/AMIS-Services/graalvm-polyglot-meetup-november2019>.

PREPARATIONS

You can of course compose your own custom workshop environment – installing the GraalVM platform and all additional tools. However, much easier and therefore the assumption for the remainder of this document is that you work in a predefined VirtualBox VM. This VM contains everything you need to run the labs – configured in a way that has been found to work.

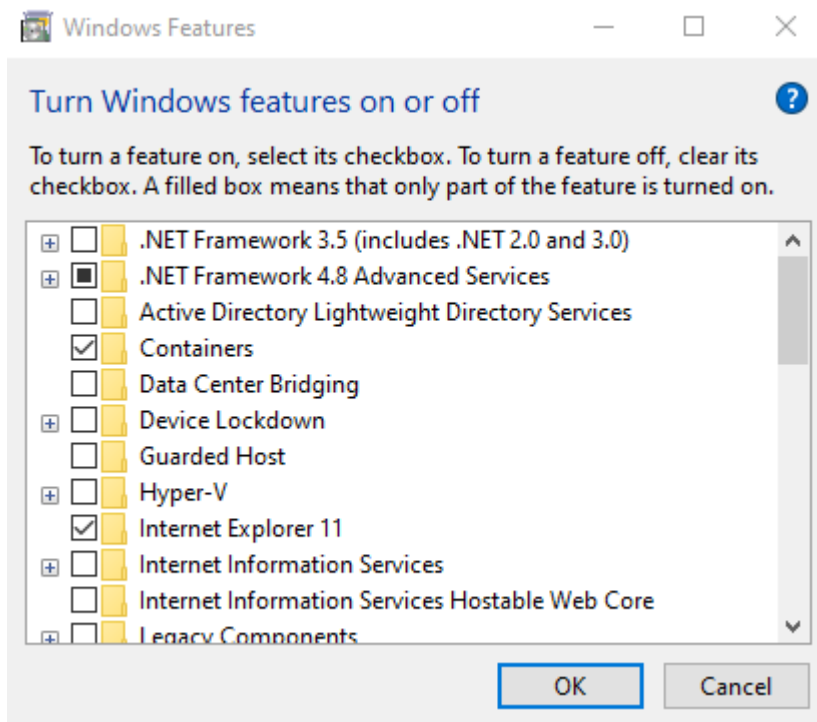
You can get your hands on a running VM in two ways:

- Build the VM – using Vagrant and the provisioning files provided in the GitHub repo for this workshop
- Import the VM from the file provided to you by the Lab instructors (or download it from the URL provided to you)

In both cases, you need to locally install Oracle VirtualBox, at least to run the VM.

INSTALL VIRTUALBOX

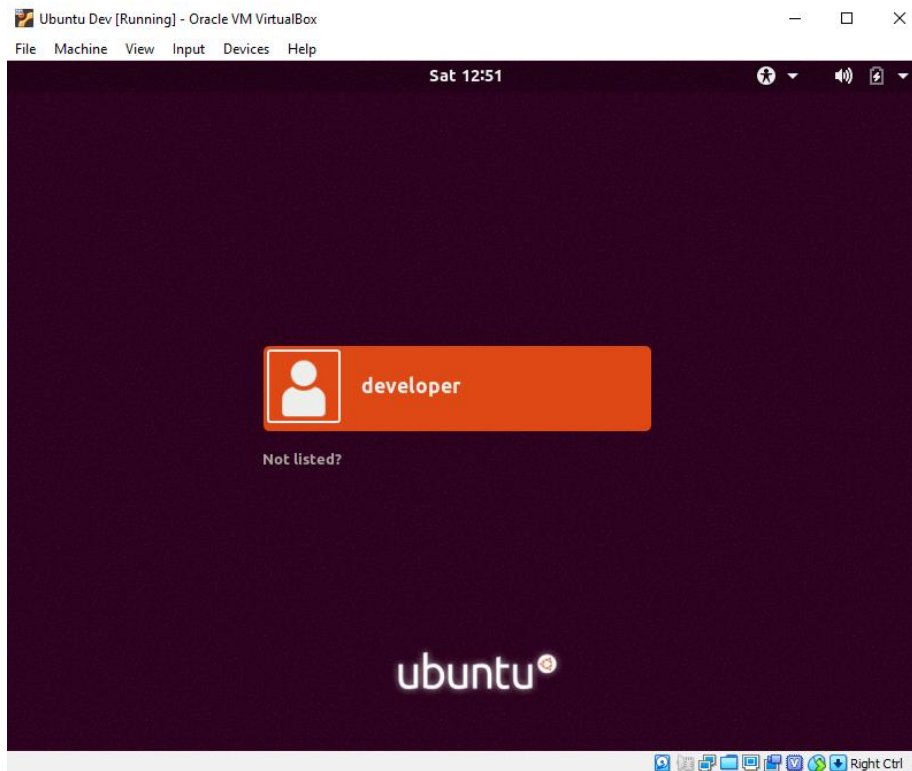
Before installing VirtualBox, make sure hardware virtualization support is enabled in the BIOS and make sure Hyper-V is disabled.



Download and install VirtualBox. If you already have Virtualbox installed and are importing the VM, make sure you have at least version 6.0.10 installed.

VirtualBox: <https://www.virtualbox.org/wiki/Downloads>

There are two ways to get started with the VM: download it and import in into VirtualBox or build it yourself using Vagrant. The second takes more time. Choose one of the options described below. After you have completed this, you can login using user *developer* password *Welcome01* as shown below.

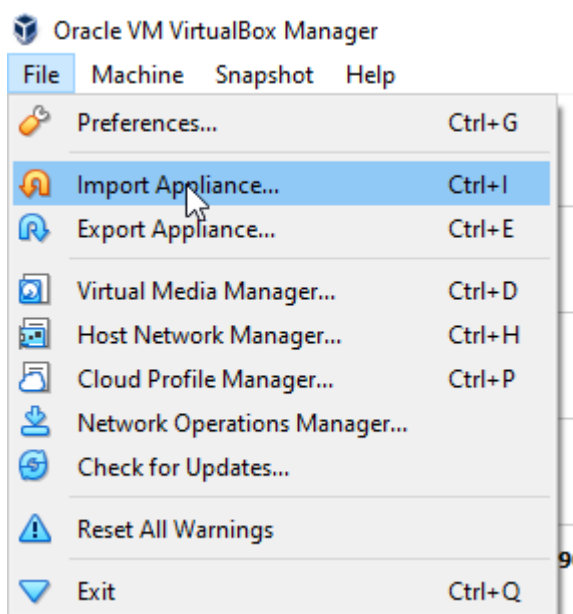


DOWNLOAD AND IMPORT THE VM IN VIRTUALBOX

Copy the appliance from the location or USB drive provided or download the appliance from https://conclusionfutureit-my.sharepoint.com/:u:/g/personal/lucas_jellema_amis_nl/ETMh8SgsueFHgU8CWingJ2MBI_QpdvBSaBy-TO5R6XljZA?e=U6cMPi.

This results in a 3GB file called *GraalVM Polyglot workshop.ova*.

Start VirtualBox and import the appliance.



Select the downloaded OVA file.

Make sure to select the correct MAC address policy else your VM will not have internet access. Also check the available CPUs and RAM of your laptop and adjust the settings accordingly.

Now you can start the imported VM. If you are curious as to how it was provisioned, you can take a look at the file `provision.sh` in the `provision` folder in the GitHub Repo: <https://github.com/AMIS-Services/graalvm-polyglot-meetup-november2019>.

BUILD THE VM YOURSELF USING VAGRANT

If you have downloaded and imported the prebuilt VM in the previous step, you should skip this step!

Download and install Vagrant: <https://www.vagrantup.com/>. If you already have Vagrant installed, make sure you have at least version 2.2.5.

Start a CLI terminal window. Git clone the workshop repo (<https://github.com/AMIS-Services/graalvm-polyglot-meetup-november2019>) to your local file system. Change the directory to `./graalvm-polyglot-meetup-november2019/provision`.

The `Vagrantfile` file in this directory will provide instructions for the Vagrant VM provisioner. You may want to update the IP address, CPU count or memory settings in this file.

The file `provision.sh` performs the hard work once the VM is provisioned according to the Ubuntu 18.04 base image. It installs the Ubuntu Desktop, GraalVM, Eclipse, Google Chrome, Visual Studio Code and the GitHub Repository for the workshop.

Start a command prompt in the directory and do:

```
vagrant up
```

After 10-15 minutes (primarily depending on download speeds) your VM will be provisioned.

You will have to wait for many things to happen now:

- downloading the Ubuntu image
- creating the VM through VirtualBox API
- provisioning the Ubuntu environment inside the VM with:
 - Ubuntu Desktop
 - Java 8
 - Eclipse
 - Firefox and Chrome
 - GraalVM 19.2.1 plus extra features (R, Ruby, Python, native image)
 - Visual Studio Source and the GraalVM extensions for Visual Studio Source
 - the GitHub Repo for the workshop resources

You could now run it from the command line (with `vagrant up`) but instead let's use the main entrance through the Ubuntu desktop that has been installed. Switch to the VirtualBox client. Locate the new VM - called *GraalVM Polyglot workshop* - and start it.

The VM starts and you will be able to enter the desktop environment.

TROUBLESHOOTING

DISKSIZE OR VBGUEST

if `vagrant up` fails with an error message that looks like this:

* Vagrant:

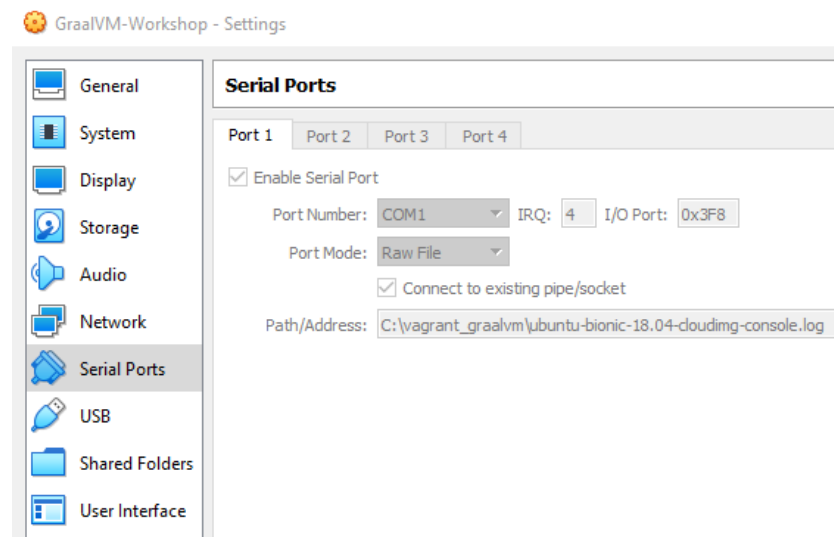
* Unknown configuration section 'vbguest'.

* Unknown configuration section 'disksize'.

then please run *vagrant up* again; there may be an issue with installing the Vagrant Plugins *disksize* and *vbguest*.

SERIAL PORT

Should the imported or generated VM hang during boot, you might have to configure the Serial port. Make sure a serial port is available and it can send output to a file;



INTERNET

Some of the steps require internet availability. One of the issues you might encounter is that the MAC address is hardcoded in the VM in the file `/etc/netplan/50-cloud-init.yaml`. The MAC might change due to importing the VM. The MAC address listed there should be the same as the output of: `cat /sys/class/net/enp0s3/address`

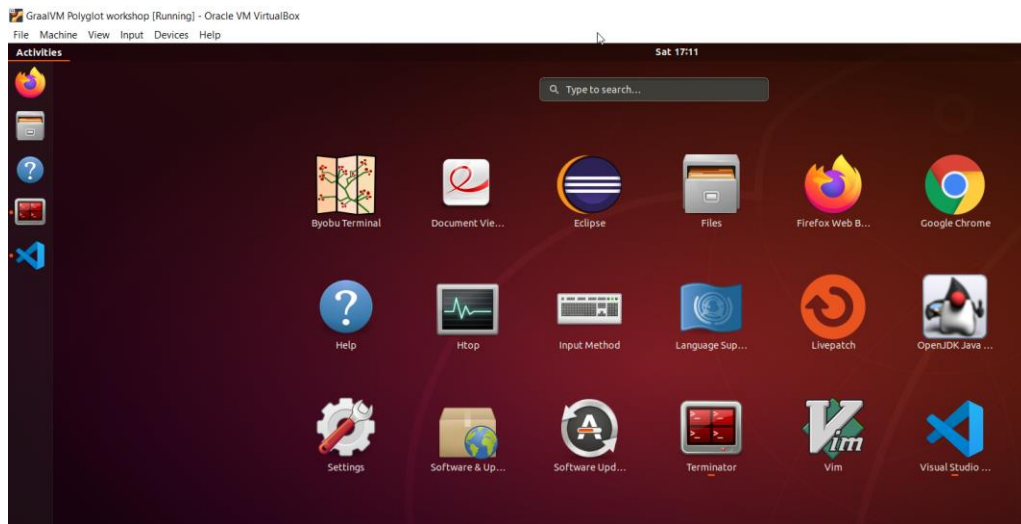
In order to fix this, execute the following:

```
sudo sed -E -i "s/([[:xdigit:]]{1,2}):([[:xdigit:]]{1,2}):([[:xdigit:]]{1,2}):([[:xdigit:]]{1,2}):([[:xdigit:]]{1,2}):([[:xdigit:]]{1,2})/\`cat /sys/class/net/enp0s3/address`/" /etc/netplan/50-cloud-init.yaml
sudo netplan apply
```

EXPLORING THE VM

Once you have entered the Virtual Machine, you can go for a little stroll to see what is there.

The grid icon in the bottom left hand corner gives you quick access to an overview of all applications that have been installed in the VM, as shown below.



Using the Terminator – you can start terminals. With the Files (explorer) you can inspect the file system.

JAVA APPLICATIONS CALLING OUT TO JAVASCRIPT

Java interoperating with JavaScript. Let's dive right in.



HELLOWORLD APPLICATION

Here the HelloWorld example – the gentle introduction into Java code interacting with JavaScript. Your first taste of polyglot interoperability.

From folder `/home/developer/graalvm-polyglot-meetup-november2019/java2js`, Generate a `.class` with the `javac` command and then check the code outputs as expected:

```
javac nl/amis/java2js/HelloWorld.java
java nl/amis/java2js/HelloWorld
```

It may feel as if you have executed just some Java code. However, the Java Class engaged JavaScript to do some of the work. Nothing spectacular as such – but two languages as brothers in arms at run time. That in itself is quite something!

Inspect the contents of the Java Class definition:

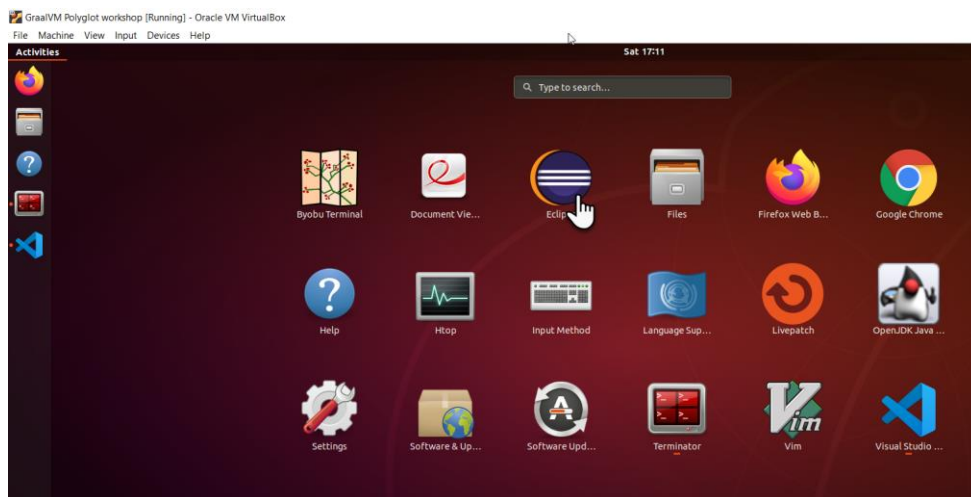
```
cat nl/amis/java2js/HelloWorld.java
```

The polyglot context object is the bridge between Java and JavaScript. Not only can we execute JavaScript snippets and get simple values returned, the JavaScript code can also return a function – that we can execute later on – and many times – from Java.

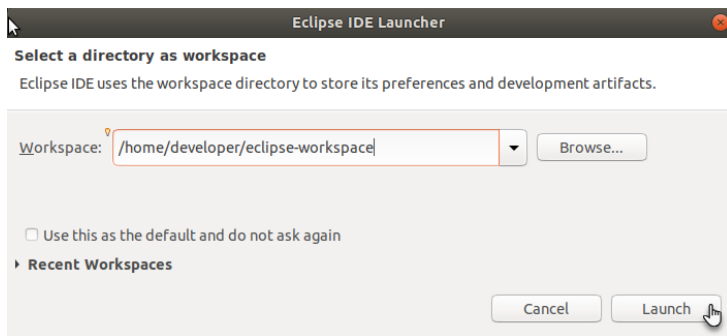
YOUR FAVORITE IDE

You are absolutely welcome to continue working with Java on the command line. Or use Visual Studio Code. This section discusses the use of Eclipse as the IDE of choice – even though we will not do any IDE intensive work.

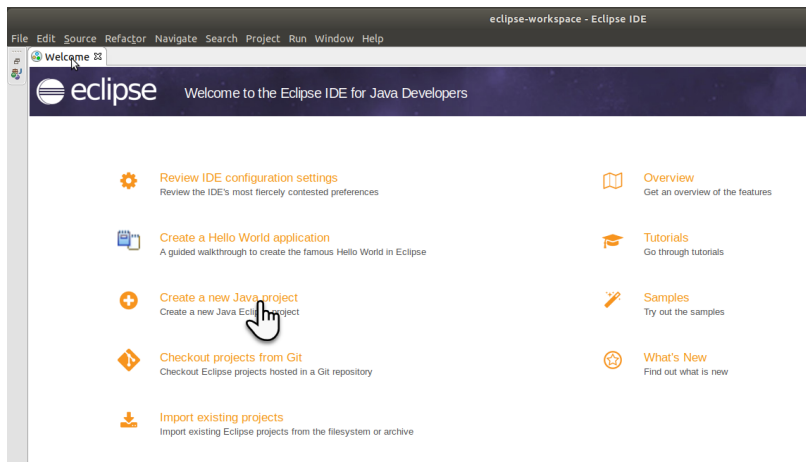
Start Eclipse from the icon.



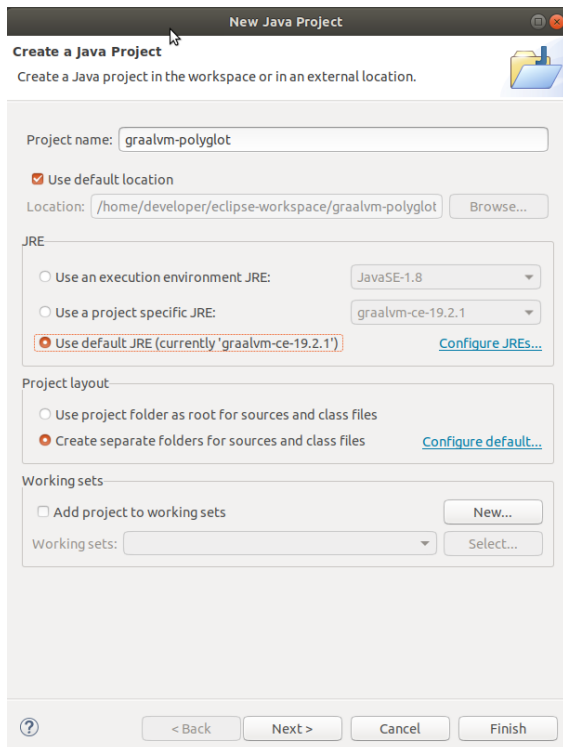
The Launcher Dialog appears.



Accept the proposed default location. Click on Create new Java project.

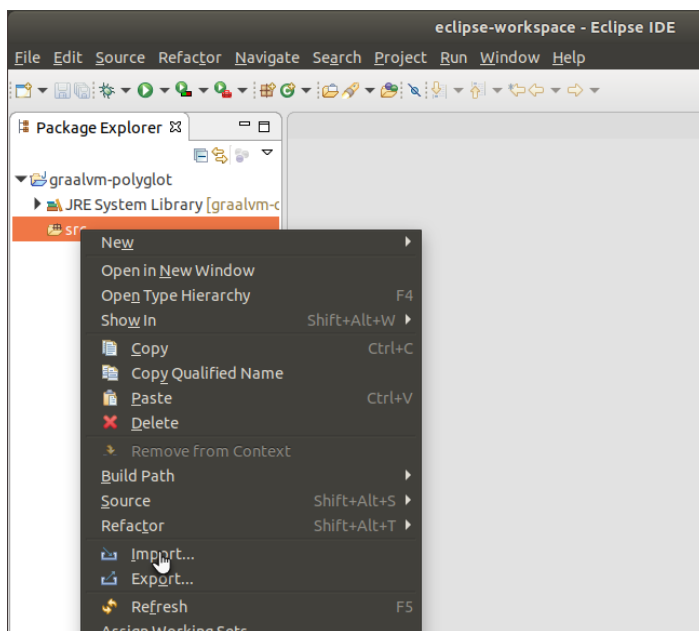


Provide the details for the project. The name is up to you. The JRE to use is important: make sure to select an option that uses graalvm-ce-19.2.1 as the run time.

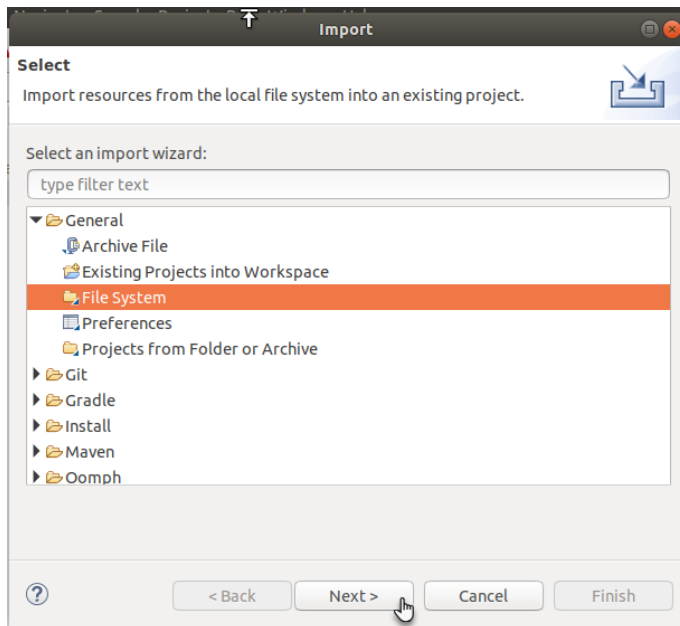


Click on Finish.

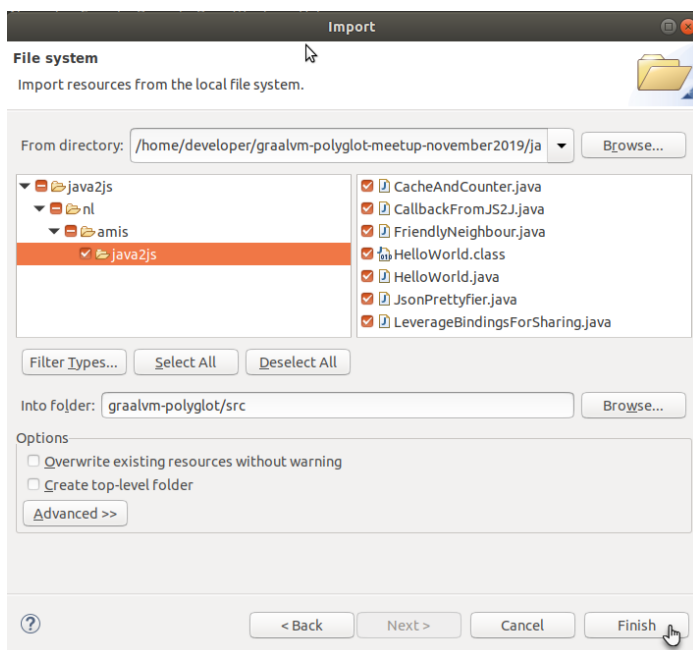
Open the context menu on the *src* folder in the new project and select the *Import* option – for importing Java sources from the GitHub repo into our Eclipse project.



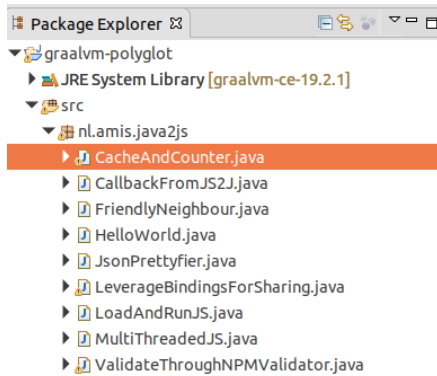
Select option General | File System and click Next.



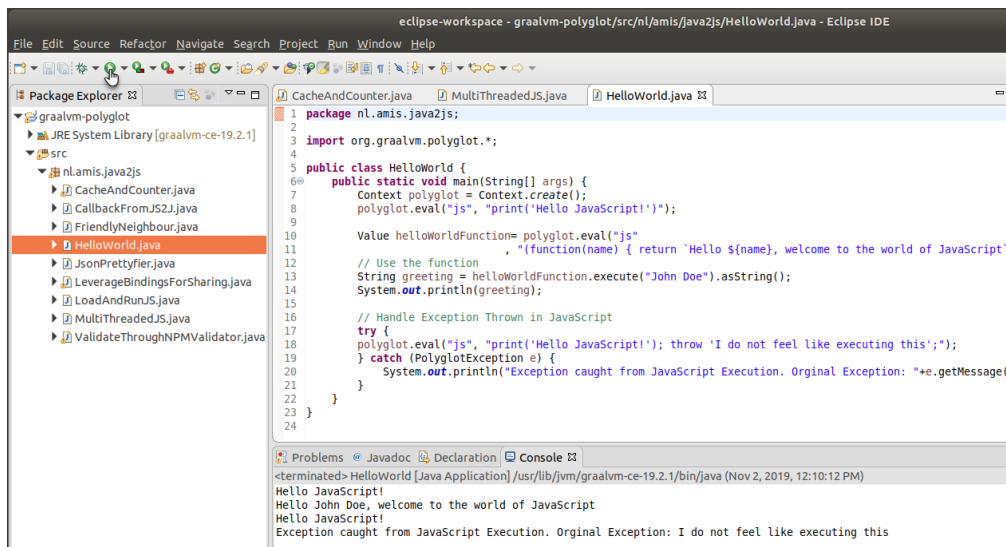
Select folder `/home/developer/graalvm-polyglot-meetup-november2019/java2js` as the source to import files from. Select all (sub) folders under this directory:



Press Finish to perform the import. Files are copied from the source directory to the Eclipse workspace:



Open the HelloWorld.java file. Then press the Run button to compile and execute this file:



The output from running the class is shown in the Console window.

LOAD JAVASCRIPT RESOURCE FROM FILE

JavaScript code does not need to be inline or embedded as Strings in the Java source. In fact, it is probably a much better idea to keep Java and JavaScript in separate files as much as possible.

Check class LoadAndRunJS. In this class you will find a simple example of how the JavaScript is not embedded in Java, but is loaded from a separate file – calculator.js - instead.



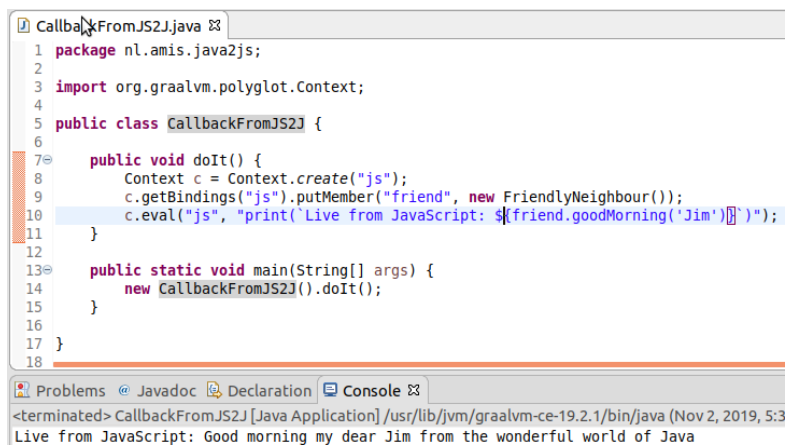
Note: you need to copy the files calculator.js to the bin directory of the Eclipse project:

```
cp /home/developer/graalvm-polyglot-meetup-november2019/java2js/calculator.js  
/home/developer/eclipse-workspace/graalvm-polyglot/bin
```

Run the Java Class and see if the calculations are performed correctly in JavaScript.

CALLBACK FROM JAVASCRIPT TO JAVA

Check class CallbackFromJS2J. In this class, a call is made to JavaScript – after a Java object has been instantiated and put in the bindings map in the Polyglot context. The JavaScript code can retrieve this object and invoke methods on it.



```
1 package nl.amis.java2js;  
2  
3 import org.graalvm.polyglot.Context;  
4  
5 public class CallbackFromJS2J {  
6  
7     public void doIt() {  
8         Context c = Context.create("js");  
9         c.getBindings("js").putMember("friend", new FriendlyNeighbour());  
10        c.eval("js", "print('Live from JavaScript: ${friend.goodMorning('Jim')}')");  
11    }  
12  
13    public static void main(String[] args) {  
14        new CallbackFromJS2J().doIt();  
15    }  
16  
17 }  
18
```

Problems Javadoc Declaration Console

<terminated> CallbackFromJS2J [Java Application] /usr/lib/jvm/graalvm-ce-19.2.1/bin/java (Nov 2, 2019, 5:3
Live from JavaScript: Good morning my dear Jim from the wonderful world of Java

You could try additional things, such as adding variables or methods to the FriendlyNeighbour class and see if you can execute and access methods and data from both sides of the polyglot divide.

EXPERIMENTING WITH BINDINGS

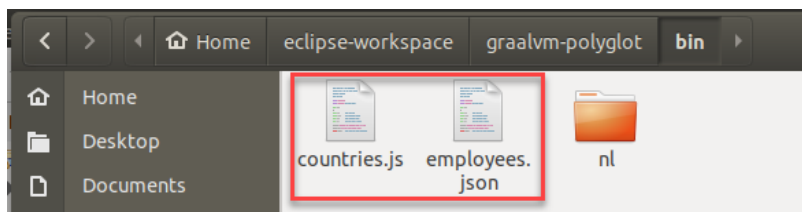
Check class LeverageBindingsForSharing. In this class you will find various examples of how objects are shared across language context using the bindings map. One example found in this class is how Java could leverage the JSON parsing capabilities of JavaScript by passing JSON data loaded from file to a JavaScript polyglot object for processing. The JavaScript object that is returned can fairly easily be read in Java.

```

31 Map<String, Object> backingMap = new HashMap<>();
32 backingMap.put("myKey", "myValue");
33 backingMap.put("myQuestion", "2*3");
34 c.getBindings("js").putMember("hostObject", ProxyObject.fromMap(backingMap));
35 // access the Java Map turned JavaScript object in bindings from JavaScript:
36 Integer answer = c.eval("js", "print('your key = ${hostObject.myKey}');");
37 + "hostObject.yourAnswer = eval(hostObject.myQuestion) ; eval(hostObject.yourAnswer)")
38 .asInt();
39 // the answer is available from the evaluation of the JS snippet
40 System.out.println("The Answer to " + backingMap.get("myQuestion") + " = " + answer);
41 // and also from the updated hostObject/backingMap in the bindings map
42 System.out.println("The Answer to " + backingMap.get("myQuestion") + " = " + backingMap.get("you
43 // creating new objects in JavaScript adds them to the bindings object - and
44 // makes them accessible in Java
45 c.eval("js", "var PI = 3.141592");
46 System.out.println("Current contents of Bindings object: " + c.getBindings("js").getMemberKeys());
47 Double pi = c.getBindings("js").getMember("PI").asDouble();
48 System.out.println("PI according to JavaScript = " + pi);
49
50 Map atlas = c.getBindings("js").getMember("atlas").as(java.util.Map.class);
51 System.out.println("Message from JavaScript Object: " + atlas.get("message"));
52
53 // get hold of JavaScript Array Object and iterate through its constituent
54 // elements
55
56 File countriesJS = new File(getClass().getClassLoader().getResource("countries.js").getFile());
57 c.eval(Source.newBuilder("js", countriesJS).build());
58 Value value = c.getBindings("js").getMember("countries");
59 if (value.hasArrayElements()) {
60     for (int i = 0; i < value.getArraySize(); i++) {
61         Map country = value.getArrayElement(i).as(java.util.Map.class);
62         System.out.println(country.get("name"));

```

Note: you need to copy the files countries.js and employees.json to the bin directory of the Eclipse project:



```

cp /home/developer/graalvm-polyglot-meetup-november2019/java2js/countries.js
/home/developer/eclipse-workspace/graalvm-polyglot/bin
cp /home/developer/graalvm-polyglot-meetup-november2019/java2js/employees.json
/home/developer/eclipse-workspace/graalvm-polyglot/bin

```

Run the Java Class and see if the processing is correct.

VALIDATING POSTAL CODES IN JAVA – LEVERAGING AN NPM JAVASCRIPT MODULE

Our challenge: implement validations of postal codes, across multiple countries, in our Java application. Our frustration: having to write such code while we know it already exists in the form of an NPM module called *Validator* – freely available in the open source domain. However: it is in JavaScript.

GraalVM to the rescue: we can now from the comfort of our Java application enlist the help of modules written in JavaScript. This challenge is discussed in detail in [this article](#).



The most important thing we need to find a workaround for is: how to make GraalJS – the JavaScript implementation on GraalVM – work with the module structure in the NPM Validator module. GraalJS does not

support `require()` or CommonJS. In order to make it work with NPM modules – they have to be turned into ‘flat’ JavaScript resources – self-contained JavaScript source file. This can be done using one of the many popular open-source bundling tools such as Parcel, Browserify and Webpack. Note: ECMAScript modules can be loaded in a Context simply by evaluating the module sources. Currently, GraalVM JavaScript loads ECMAScript modules based on their file extension. Therefore, any ECMAScript module must have file name extension `.mjs`.

The steps to turn an NPM module into a self contained bundle that GraalVM can process are these:

- check GraalVM compatibility of NPM module at <https://www.graalvm.org/docs/reference-manual/compatibility/#validator>
- install webpack and webpack-cli
- install npx (executable runner – complement to npm which is not included with GraalVM platform)
- install validator module with npm
- produce self contained bundle for validator module with webpack

In the interest of time and because the resulting `validatorbundled.js` file is already included in the workshop resources –there is no absolute need for you to perform these installations at this moment. So let’s skip them (if you are interested in the details, please refer to the blog article).

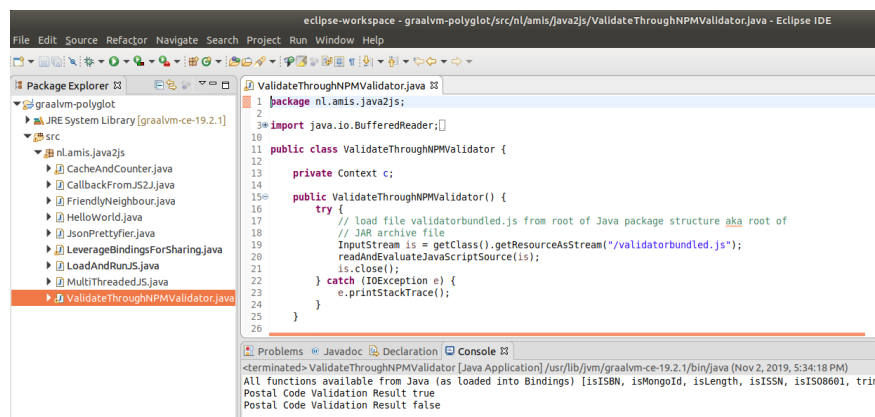
The outcome is a file called *validatorbundled.js* that contains the sublimation of all modules that contribute to the Validator module. We can now use this file as JavaScript resource in our Java application.

In Eclipse, take a look at Java Class `ValidateThroughNPMValidator`. See how this class loads the `validatorbundled.js` using the `ClassLoader` (for example from the JAR file) – or from an absolute path on the file system (used when we create a native image of this Java application, see below). The source loaded from file is evaluated against the Polyglot JavaScript context. As a result, all top level functions in the Validator module are now executable from within the Java application.

Note: you need to copy file `validatorbundled.js` to the `bin` directory in the Eclipse project – because that is root location for the compiled Java Class to load the bundle from. In my case, the Eclipse project name is `graalvm-polyglot` and the copy statement is this:

```
cp ./validatorbundled.js /home/developer/eclipse-workspace/graalvm-polyglot/bin
```

Run the Java Class and see if the validations are performed correctly.

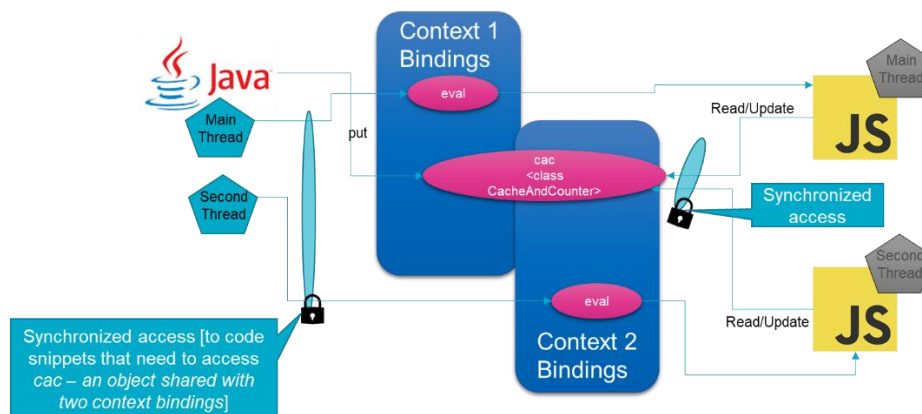


You can extend the Java Class to validate using some of the other validations in the Validator module (see: <https://www.npmjs.com/package/validator>).

More details in this article: <https://technology.amis.nl/2019/10/25/leverage-npm-javascript-module-from-java-application-using-graalvm/> .

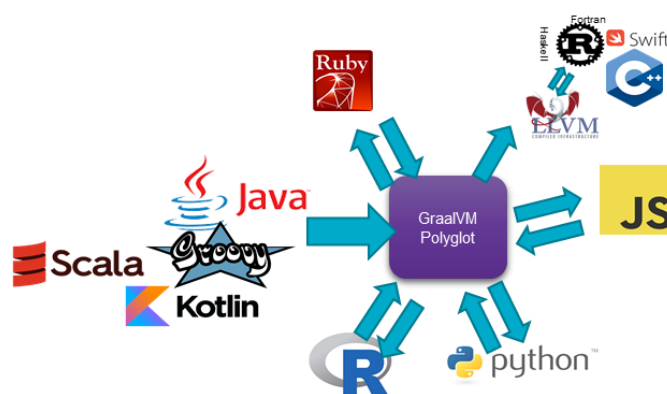
MULTI THREADED JAVA – LEVERAGING JAVASCRIPT

Check class MultiThreadedJS. In this class you will see how two Polyglot JavaScript contexts are created that are each used on a different Java Thread. A Java Object is created and associated with both JavaScript contexts. The consequence: the two Java threads each run a single threaded JavaScript context – so there are two JavaScript context running concurrently. These two JavaScript contexts – and the Java threads – all share the same Java object. For this to be handled correctly, we leverage Java synchronization – a mechanism not available in the world of JavaScript.



JAVA APPLICATIONS INTERACTING WITH OTHER “TRUFFLE” LANGUAGES

Java Polyglot applications not only can interact with JavaScript, they can just as easily evaluate and execute snippets and sources in other languages – provided there is a Truffle interpreter for that language. Some of the languages currently supported (to some extent) are R, Ruby, Python and LLVM (C, C++, Rust, Swift and others).



The GraalVM Documentation [on Embedding](#) shows a number of examples of Java code with embedded snippets of code in some of these languages. Browse through the document. Add one of the examples of calling R or Python from Java to class HelloWorld. Run the class' main method. Check whether HelloWorld in addition to speaking JavaScript now also can converse in these other languages.

CREATING A NATIVE IMAGE (STAND-ALONE EXECUTABLE) FOR A JAVA POLYGLOT APPLICATION

A Java application can be converted to a native image – a stand-alone binary executable that executes the logic of the Java application from a Ahead of Time compile code base. A polyglot Java application can also be converted into a native binary image. See for a detailed example this article:

<https://technology.amis.nl/2019/10/28/create-a-native-image-binary-executable-for-a-polyglot-java-application-using-graalvm/>.



Note: the GraalVM native image utility has been installed in the Virtual Machine. However, the AOT compilation requires a fair amount of memory. You may have to stop the VM and increase the allocated RAM memory if you find the native image generation unsuccessful.

A first quick attempt you could make to create a natively executable image of the HelloWorld class with embedded JavaScript:

```
$GRAALVM_HOME/bin/native-image -cp ./application-bundle.jar --language:js --verbose -H:Name=hello -H:Class=nl.amis.java2js.HelloWorld
```

Even this extremely simple example takes more than five minutes to create the executable file. The closed world analysis of all dependencies and the production of the executable need time and memory.

The result should be a natively executable file of moderate size (compared to the Java Runtime environment you would need to run the HelloWorld class with JIT compilation)

```
[hello:6956] analysts: 82,695.23 ms
[hello:6956] (clinit): 1,408.89 ms
8711 method(s) included for runtime compilation
[hello:6956] universe: 4,923.46 ms
[hello:6956] (parse): 10,232.35 ms

[hello:6956] image: 14,366.27 ms
[hello:6956] write: 1,364.50 ms
[hello:6956] [total]: 201,568.83 ms
developer@ubuntu:~/graalvm-polyglot-meetup-november2019/js2java$ ls -l
total 91568
-rw-rw-rw- 1 root root 52118 Nov 2 10:28 application-bundle.jar
-rwxrwxr-x 1 developer developer 93675688 Nov 2 20:20 hello
```

Run this executable with the following command. No Java, no classpath. Only this one file (that contains a JavaScript runtime engine as well as the as yet unparsed, unprocessed JavaScript snippets):

```
./hello
```

```
developer@ubuntu:~/graalvm-polyglot-meetup-november2019/js2java$ ./hello
Hello JavaScript!
Hello John Doe, welcome to the world of JavaScript
Hello JavaScript!
Exception caught from JavaScript Execution. Original Exception: I do not feel like executing this
```

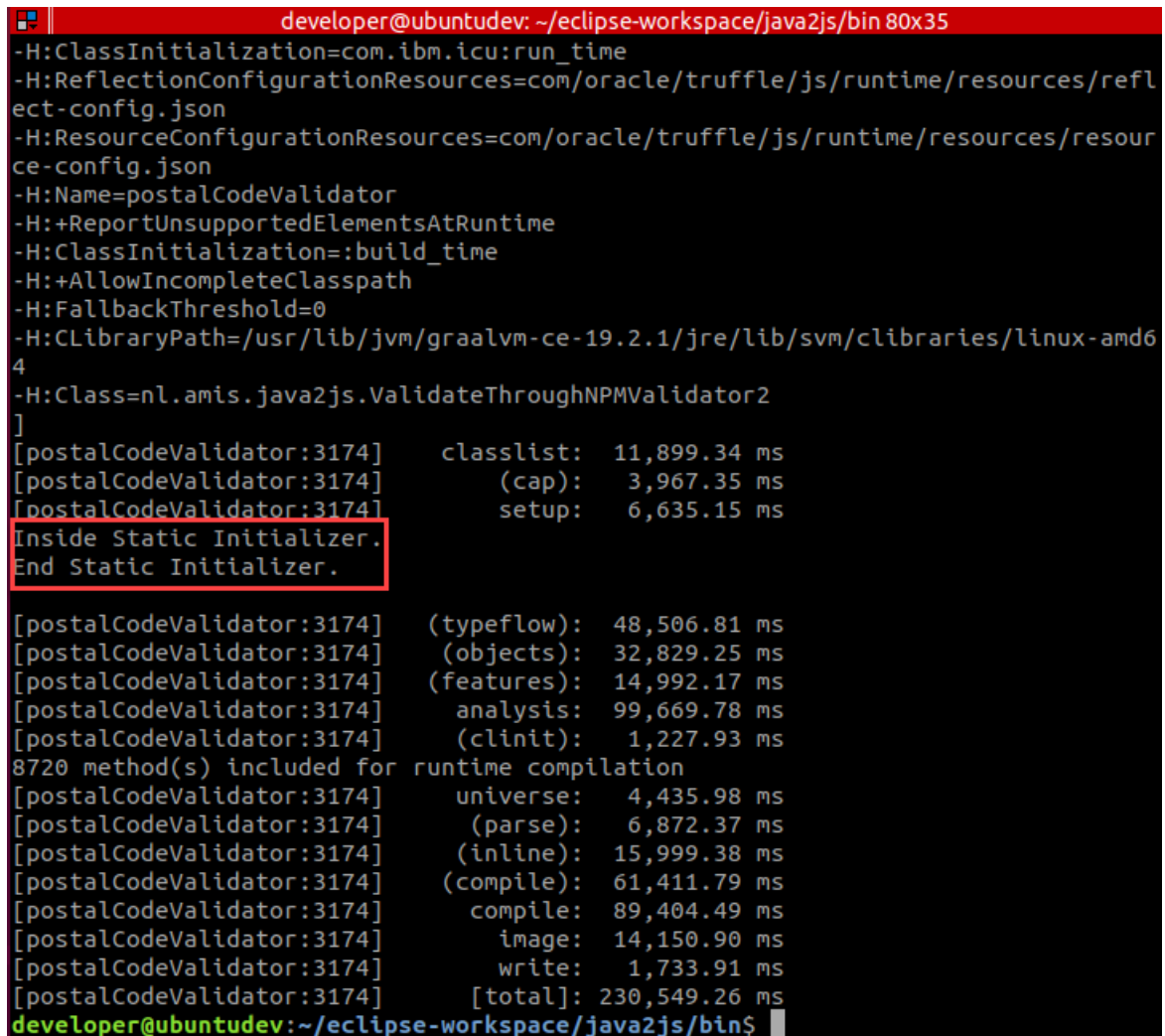
CREATING A NATIVE IMAGE FOR A POLYGLOT APPLICATION

Static Class Initializers can be executed by the native image generator at AOT time – instead of at runtime (see [this article](#) for an extensive explanation). At the time of building the native image, the class can be initialized, the JS resources can be loaded and they are embedded in the image, rather than loaded at runtime from an external file. Objects created during build time are available at run time in the so-called image heap.

Use this command to generate the native image. The crucial parameter here is `--initialize-at-build-time` that used to indicate which class[es] should be initialized at build time – to create objects for the image heap and prevent a need to do so at runtime.

```
$GRAALVM_HOME/bin/native-image -cp ./application-bundle.jar --language:js --verbose -
H:Name=postalCodeValidator -
H:Class=nl.amis.java2js.ValidateThroughNPMValidator --verbose -
H:+ReportUnsupportedElementsAtRuntime --initialize-at-build-time=
nl.amis.java2js.ValidateThroughNPMValidator --allow-incomplete-classpath
```

Here is the output from this command (that will take minutes to run):



```
developer@ubuntu16: ~/eclipse-workspace/java2js/bin 80x35
-H:ClassInitialization=com.ibm.icu:run_time
-H:ReflectionConfigurationResources=com/oracle/truffle/js/runtime/resources/refl
ect-config.json
-H:ResourceConfigurationResources=com/oracle/truffle/js/runtime/resources/resour
ce-config.json
-H:Name=postalCodeValidator
-H:+ReportUnsupportedElementsAtRuntime
-H:ClassInitialization=:build_time
-H:+AllowIncompleteClasspath
-H:FallbackThreshold=0
-H:CLibraryPath=/usr/lib/jvm/graalvm-ce-19.2.1/jre/lib/svm/libraries/linux-amd6
4
-H:Class=nl.amis.java2js.ValidateThroughNPMValidator2
]
[postalCodeValidator:3174]      classlist:  11,899.34 ms
[postalCodeValidator:3174]      (cap):    3,967.35 ms
[postalCodeValidator:3174]      setup:    6,635.15 ms
Inside Static Initializer.
End Static Initializer.

[postalCodeValidator:3174]      (typeflow): 48,506.81 ms
[postalCodeValidator:3174]      (objects): 32,829.25 ms
[postalCodeValidator:3174]      (features): 14,992.17 ms
[postalCodeValidator:3174]      analysis: 99,669.78 ms
[postalCodeValidator:3174]      (clinit):  1,227.93 ms
8720 method(s) included for runtime compilation
[postalCodeValidator:3174]      universe:  4,435.98 ms
[postalCodeValidator:3174]      (parse):   6,872.37 ms
[postalCodeValidator:3174]      (inline): 15,999.38 ms
[postalCodeValidator:3174]      (compile): 61,411.79 ms
[postalCodeValidator:3174]      compile: 89,404.49 ms
[postalCodeValidator:3174]      image:    14,150.90 ms
[postalCodeValidator:3174]      write:    1,733.91 ms
[postalCodeValidator:3174]      [total]: 230,549.26 ms
developer@ubuntu16: ~/eclipse-workspace/java2js/bin$
```

The resulting executable is called `postalCodeValidator`. It can be executed simply with `./postalCodeValidator`. It evaluates the JavaScript from `validatorbundled.js` at runtime – reading it from the image heap that was created at image build time.

```

[postalCodeValidator:3174]      image: 14,130.90 ms
[postalCodeValidator:3174]      write: 1,733.91 ms
[postalCodeValidator:3174]      [total]: 230,549.26 ms
developer@ubuntu:~/eclipse-workspace/java2js/bin$ ls -l
total 92072
-rw-rw-r-- 1 developer developer 42173 Nov 6 21:09 application-bundle.jar
drwxrwxr-x 3 developer developer 4096 Nov 6 20:48 nl
-rwxrwxr-x 1 developer developer 94072992 Nov 6 21:47 postalCodeValidator
-rw-rw-r-- 1 developer developer 153628 Nov 6 20:34 validatorbundled.js
developer@ubuntu:~/eclipse-workspace/java2js/bin$ ./postalCodeValidator
All JavaScript functions now available from Java (as loaded into Bindings) [isISBN, isMongoId, isLength, isISSN, isISO8601, trim, toBoolean, isURL, isInt, isDecimal, isFQDN, isMACAddress, isVariableWidth, isAlpha, isISRC, ltrim, whitelist, version, matches, isDataURI, isLatLong, isMobilePhone, contains, isAscii, isFloatLocales, isCreditCard, isLowercase, isNumeric, isJSON, isEmail, normalizeEmail, isAfter, isFullWidth, isJWT, isUppercase, isSurrogatePair, isHexadecimal, isUID, isISO31661Alpha2, isISO31661Alpha3, isWhitelisted, isBase32, isHalfWidth, isBoolean, default, isMagnetURI, isAlphanumeric, isPostalCodeLocales, isCurrency, isRFC3339, isISIN, isIdentityCard, isIn, isAlphanumericLocales, isIP, toInt, stripLow, isHexColor, isMultibyte, isMobilePhoneLocales, isMimeType, toDate, toFloat, isEmpty, isPostalCode, rtrim, isMD5, isByteLength, blacklist, isFloat, isAlphaLocales, isPort, isBase64, isBefore, equals, isIPRange, toString, isDivisibleBy, isHash]
Postal Code Validation Result true
Postal Code Validation Result false
developer@ubuntu:~/eclipse-workspace/java2js/bin$

```

Note: to create a JAR file with all required Java Classes – actually, only one is really needed – as well as other resources – notably the validatorbundled.js – execute this next statement in the *bin* directory of the Eclipse project:

```
jar cvf application-bundle.jar .
```

This will create the application-bundle.jar with all classes as well as the validatorbundled.js.

BONUS: LOADING THE JAVASCRIPT RESOURCES AT RUNTIME

If you are especially interested, follow the steps described in [this article](#) to create a native image for a Polyglot Java application that loads JavaScript resources - such as the NPM Validator Module - from external files at run time.

NODE & JAVASCRIPT APPLICATIONS EXECUTING JAVA CODE

Node and JavaScript applications interoperating with Java.



JOKER APPLICATION

Open Visual Source Code and open folder `/home/developer/graalvm-polyglot-meetup-november2019/js2java`. Open file `Joker.js`. It is monoglot – and utterly dull. The Joker does not have a single Joke. Very unfortunate.

Run the application:

```
node joker.js
```

You will not be dazzled, no tricks up anyone's sleeves.

Now open the file `joker2.js`. Things start to look more interesting. The joker still does not have any jokes – but he has a friend. A Java Class, called `Joker`, that may help out.

Run the application with this command

```
node --jvm --vm.cp application-bundle.jar joker2.js
```

Now there should be jokes cracked left and right. They must be produced by the Java Joker. Take a look at the file `Joker.java` in folder `nl/amis/js2java` to see how that class is coded. And to see that is not aware of the fact that it is used in a polyglot context. This is just a regular Java Class, doing its thing.

File `joker3.js` takes another step. It shows how we can post parameters and exchange more complex objects – such as an Array and a Map – between JavaScript and Java..

Run the application with this command

```
node --jvm --vm.cp application-bundle.jar joker3.js
```

VALIDATOR APPLICATION

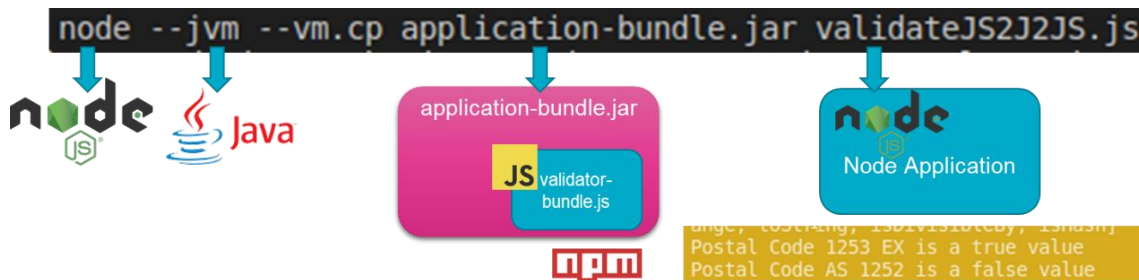
Open file `validateJS2J2JS.js`. The JavaScript application wants to validate a Postal Code. The developer knew about the Java Class `ValidateThroughNPMValidator` that we created a little earlier on, so she thought she might as well make use of it.

Run the application with this command

```
node --jvm --vm.cp application-bundle.jar validateJS2J2JS.js
```

and find that two postal codes are validated.

The remarkable thing here is that what is actually taking place is JavaScript executing Java code that in turn is executing JavaScript code. Not an obvious thing to do – but not a problem on a technical level.



GRAALVM TOOLS

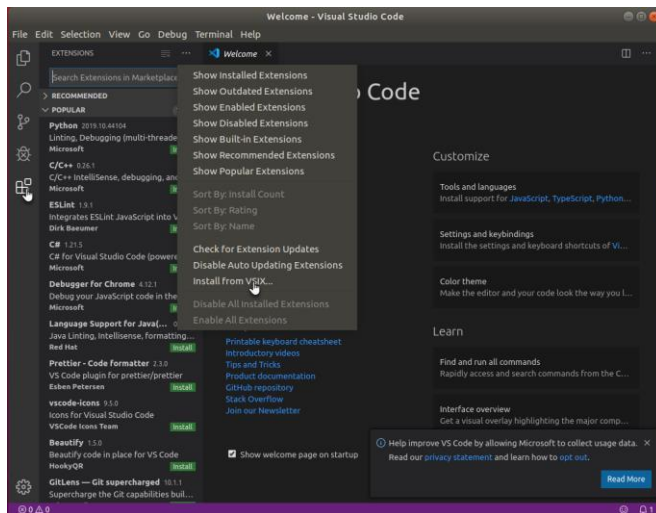
GraalVM provides a number of tools. These tools are useful enough when you are running just a monoglot application on GraalVM. However, they have tremendous benefit in the fact that they too understand polyglot. They help with debugging, profiling and tracing applications across the various polyglot language contexts.

The current GraalVM Platform toolset comprises

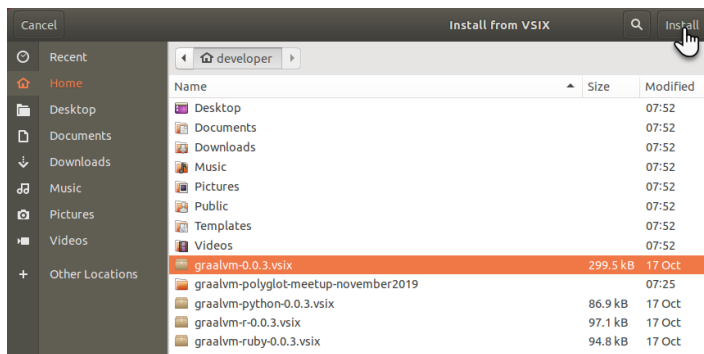
- Debugger linked to Chrome DevTools
- Profiler – command line CPU Sampler, CPU tracer and Memory tracer
- Graal VisualVM – Visual CPU Sampler and Heap Analyzer
- Ideal Graph Visualizer – to view and inspect interim graph representations from GraalVM and Truffle compilations
- Visual Studio Code Extensions – to help with Editing and debugging applications running on GraalVM

INSTALLING THE VISUAL STUDIO CODE EXTENSIONS

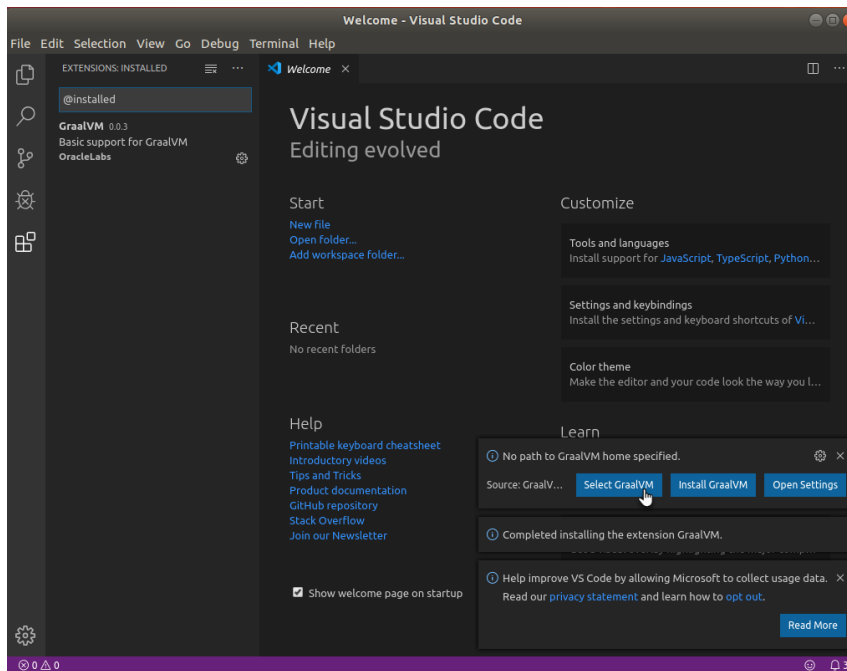
Open the VS Code extension management page. Click on context menu and select Install from VSIX. This will



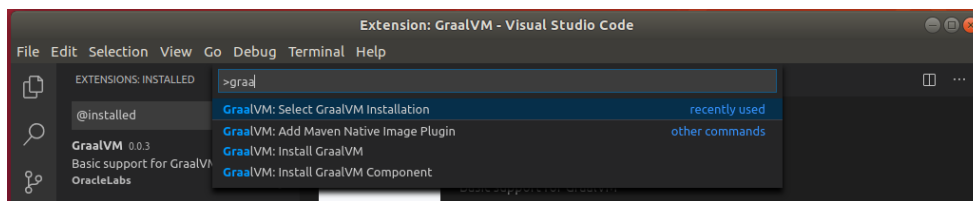
bring up a dialog window where you can select the VSIX file you want to import an extension from. In `/home/developer` there are four such VSIX files that you can each import. Make sure to at least import the main `graalvm-0.0.3.vsix` file.



After you press *install* the extension is loaded and initialized and you will be invited to select the GraalVM installation you want to make use of. You can immediately set up this reference:

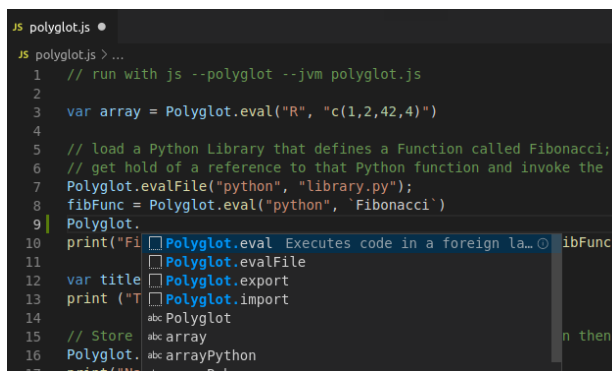


or Ctrl+Shift+P and type Graal; auto completion will list the actions available:



Select *GraalVM: Select GraalVM Installation*, and point at the GraalVM 19.2.1 installation in the VM.

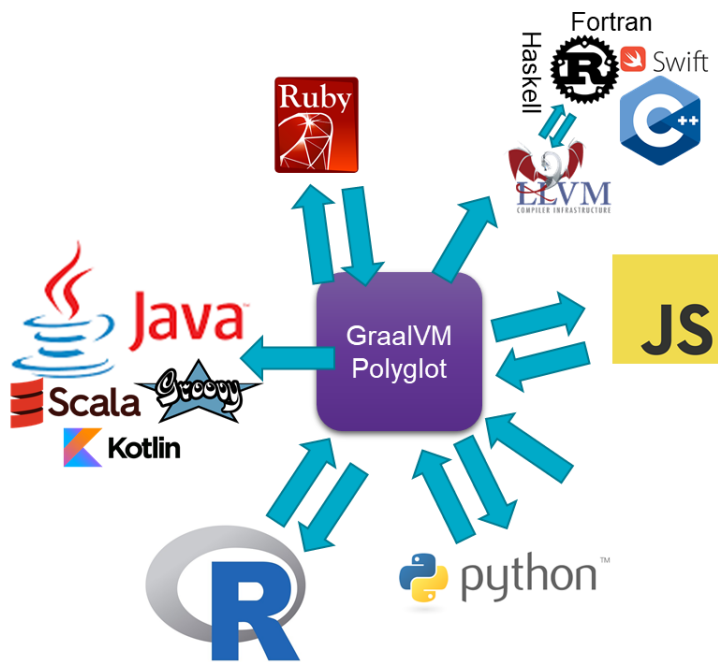
Code completion in Node and JavaScript sources:



Similar in Python, Ruby and R if you install the corresponding VS Code extensions.

POLYGLOT MULTI-DIRECTIONAL INTEROPERABILITY

Polyglot on GraalVM can start from Java (calling out to other languages) and from Node or JavaScript (calling out to Java). But these are just two examples. Polyglot can start from any of the languages that GraalVM can run – and can have interaction with any of the other languages that GraalVM can run. The next figure visualizes this.



In this section, we will look at some examples of polyglot language interaction, not necessarily focused on Java, and involving more than just two languages.

FROM PYTHON TO JAVASCRIPT, JAVA AND R

This [article](#) describes in detail a Python application running on GraalVM and interacting with both JavaScript, R and Java. The code discussed in the article is also available in the VM.

To try it out, run the application with this command – on the command line from directory `/home/developer/graalvm-polyglot-meetup-november2019/polyglot`:

```
graalpython --polyglot --jvm ./python_polyglot.py
```

Here is the output you may expect to get:

```
developer@ubuntu:~/graalvm-polyglot-meetup-november2019/polyglot$ graalpython --polyglot --jvm ./python_polyglot.py
Please note: This Python implementation is in the very early stages, and can run little more than basic benchmarks at this point.
JavaScript 42
Setting LC_COLLATE failed, using default
Setting LC_MONETARY failed, using default
Setting LC_TIME failed, using default
Setting LC_MESSAGES failed, using default
R 2
Java 42
Ruby 42
[python] WARNING: [deprecation] polyglot.export_value(str, str) is ambiguous. In the future, this will default to using the first argum
ent as the name and the second as value, but now it uses the first argument as value and the second as the name.
[python] WARNING: [deprecation] polyglot.export_value(value, name) is deprecated and will be removed. Please swap the arguments.
<foreign object at 0x476c7850>
Title from polyglot Polyglot Programming
The result of invoking the function produced by the JavaScript function: Hello Hank, welcome to the world of JavaScript
Imported key from polyglot value
22
The result of invoking function squared imported from polyglot 484
5
[1] "Imported title from Polyglot: Polyglot Programming"
squared from R using function from JS 25
```

Feel free to open, inspect and edit the file `python_polyglot.py`.

FROM JAVASCRIPT TO PYTHON AND R

Similar - but taking JavaScript as the starting language – is `polyglot.js`. This JavaScript application loads a Python library and invokes the Python function object that is created by that library. It also uses an R snippet that

creates a function object and hands it back to the Polyglot context for all participating language contexts to invoke.

To try it out, run the application with this command – on the command line from directory /home/developer/graalvm-polyglot-meetup-november2019/polyglot:

```
js --polyglot --jvm polyglot.js
```

Here is the output you may expect to get:

```
developer@ubuntu:~/graalvm-polyglot-meetup-november2019/polyglot$ js --polyglot --jvm polyglot.js
Setting LC_COLLATE failed, using default
Setting LC_MONETARY failed, using default
Setting LC_TIME failed, using default
Setting LC_MESSAGES failed, using default
99
Fibonacci(9) 21
Fibonacci(9) - called from JS, executed in Python 21
Title retrieved from Python: Polyglot Programming - defined in Python
Name from Polyglot Map: Wim
Name from Polyglot is Wim
[python] WARNING: [deprecation] polyglot.export_value(str, str) is ambiguous. In the future, this will default to using the first argument as the name and the second as value, but now it uses the first argument as value and the second as the name.
[python] WARNING: [deprecation] polyglot.export_value(value, name) is deprecated and will be removed. Please swap the arguments.
Name from Polyglot Map Hans
42
41
2
Square of 5 = 25 - according to Python
[1] "Calculated Square in R using Python Function retrieved from Polyglot through JS : 81"
Square of 9 = 81
```

Please, take this opportunity and modify the source *polyglot.js*. Make it jump through more hoops. Make the polyglot contexts share even more.