Anton Epple

Lab Guide HOL3962

Java Everywhere Again
with DukeScript

Java Everywhere Again with DukeScript

JavaOne Hands on Lab 3962

Anton Epple

Contents

Preface

For many years, Java Swing enabled developers to write applications that could run on any operating system. That all came to an end with the arrival of smart phones, tablets, and embedded computers. In the enterprise, the desktop dominated for many years. In the meantime, however, almost every IT project includes plans for a future where the application will need to be ported to mobile platforms. Building native applications for all platforms requires special skills and is expensive in terms of both maintenance and development.

We wanted to solve this problem, to make cross-platform development available for everyone again. That's why we developed DukeScript. In this lab you will learn how to use DukeScript and develop Java applications that run on a lot of different devices.

How it works

The basic requirement to run DukeScript on a System is a JVM and a HTML5 renderer component.

On the Desktop we use Oracles HotSpot Virtual Machine and as a HTML-Renderer we use the JavaFX WebView. In the Browser we use the bck2brwsr VM or TeaVM, and the Browser itself is the HTML-Renderer. Here's a little table to show what we use on supported platforms:

OS Platform	Java VM	HTML5 Renderer
OS X, Windows, Linux	Oracle Hotspot	JavaFX WebView
iOS	RoboVM	NSObject.UIResponder.UiView.UIWebView
Android	Dalvik/ART	android.webkit.WebView
Browser	bck2brwsr/TeaVM	Chrome, Firefox, IE…
Embedded	OpenJDK/JamVM/Java SE Embedded	Chromium

This architecture made it really easy to port to different platforms, and we can easily adapt to new platforms if there is demand. But now without further ado, let's start coding DukeScript!

1. Getting started!

It's time to get your hands dirty. In this first exercise you'll learn everything you need to create your first DukeScript application using NetBeans.

Prerequisites

If you're using this Lab at JavaONE 2015, the Laptops in the room are equipped with a VirtualBox Image that has all the required software already installed. Just start VirtualBox, choose "HOL3962" from the list of virtual machines and start it.

In case you prefer to use your own hardware you need at least JDK 7[1] and Maven 3.2.5[2]. The best IDE for developing DukeScipt is NetBeans which provides some extra features like hot swapping of code and visual DOM Inspection. You can also use Eclipse or IntelliJ IDEA, but for this lab, we assume you're using NetBeans. For developing Android applications you need to install the Android SDK[3] and API Level 19, and for building for iOS you need the XCode 6[4] and a Mac running OS X 10.9 or later.

Developing DukeScript apps with the NetBeans Plugin

The easiest way to develop, debug and deploy DukeScript applications is to use NetBeans and it's DukeScript Plugin. There are a lot of extra features we've put into that, so we strongly suggest to use it. Throughout the lab we'll be using NetBeans for most tasks.

On the virtual machine in this lab NetBeans already has the Plugin installed. If you're using your own hardware, go to **Tools/Plugins**, refresh the catalog, select available plugins tab and install **DukeScript Project Wizard**. The Plugin will install a new Wizard and an Update Center.

Create a new Project

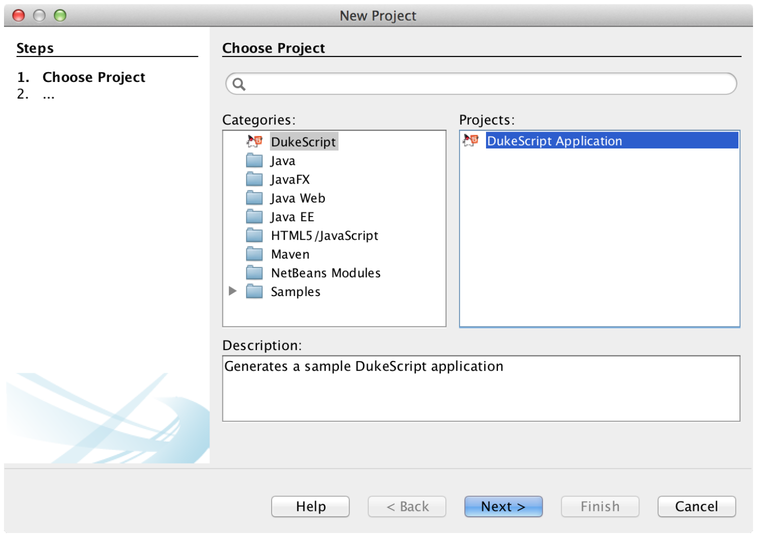
Now create new project (File | New Project...). In the **New Project** Wizard switch to category **DukeScript**. Choose the template **DukeScript Application**:

[1]http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

[2]https://maven.apache.org/

[3]https://developer.android.com/sdk/index.html

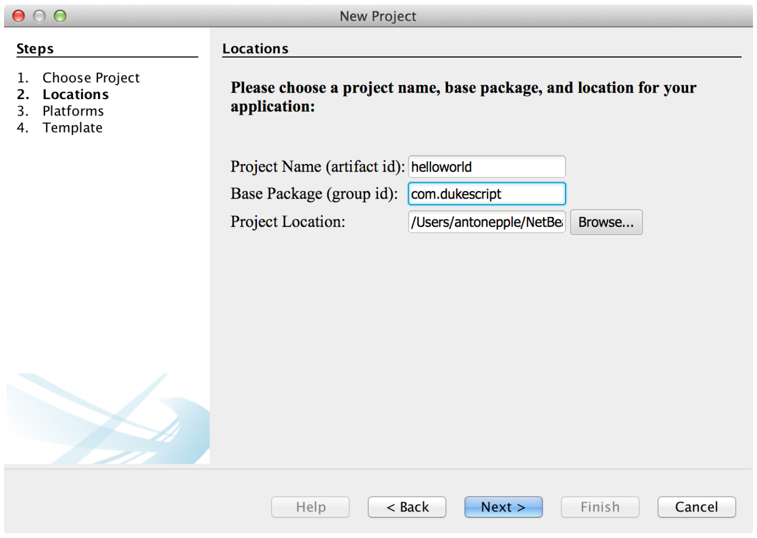
[4]https://itunes.apple.com/us/app/xcode/id497799835?mt=12



Generates a sample DukeScript application

New DukeScript Application

In Step 2 you need to specify the location where to create your project and the Maven coordinates. Use “helloworld” as the application name and whatever package name you like. The Maven group id will automatically be used as the base package of your application.

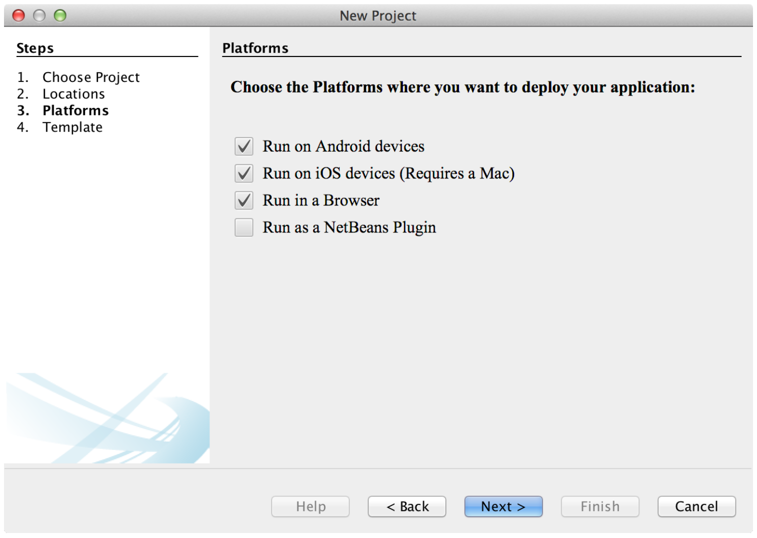


Please choose a project name, base package, and location for your application:

Project Name (artifact id): helloworld
Base Package (group id): com.dukescript
Project Location: /Users/antonepple/NetBe

Add Maven Coordinates

The next wizard steps ask to what platforms you would like to deploy. The options are Android, iOS, Browser and NetBeans Plugin. In addition to that a Desktop Client will automatically created for you. This is the one that is used for testing and debugging by default. Please select Android and Browser as target platforms. (iOS requires the build to be run on a Mac, while the VM used in this lab is running Windows 7)



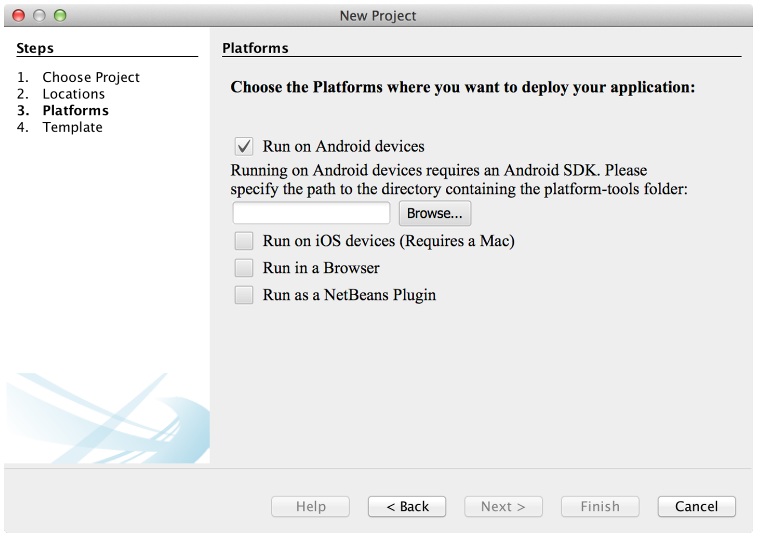
Platforms

Choose the Platforms where you want to deploy your application:

- [✓] Run on Android devices
- [✓] Run on iOS devices (Requires a Mac)
- [✓] Run in a Browser
- [ ] Run as a NetBeans Plugin

Choose Target Platforms

If you choose Android as a platform here for the first time, the wizard will ask you where your Android SDK is installed. Point it to "C:/Program Files(x86)/Android/Android SDK"



Platforms

Choose the Platforms where you want to deploy your application:

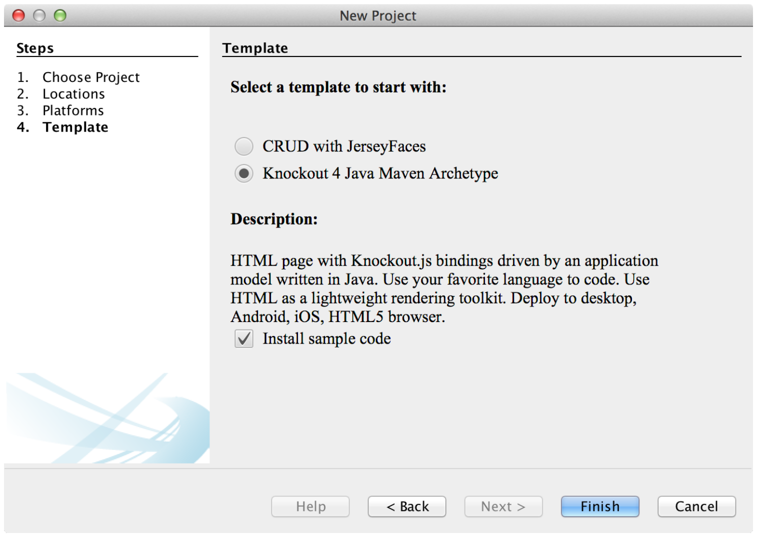
- [✓] Run on Android devices

Running on Android devices requires an Android SDK. Please specify the path to the directory containing the platform-tools folder:

- [ ] Run on iOS devices (Requires a Mac)
- [ ] Run in a Browser
- [ ] Run as a NetBeans Plugin

Choose Target Platforms

In the last wizard step you can choose between the available project templates. Let's go for the simplest one **Knockout 4 Java Maven Archetype**. Make sure you also check the checkbox to **install sample code**.



Template

Select a template to start with:

( ) CRUD with JerseyFaces
(●) Knockout 4 Java Maven Archetype

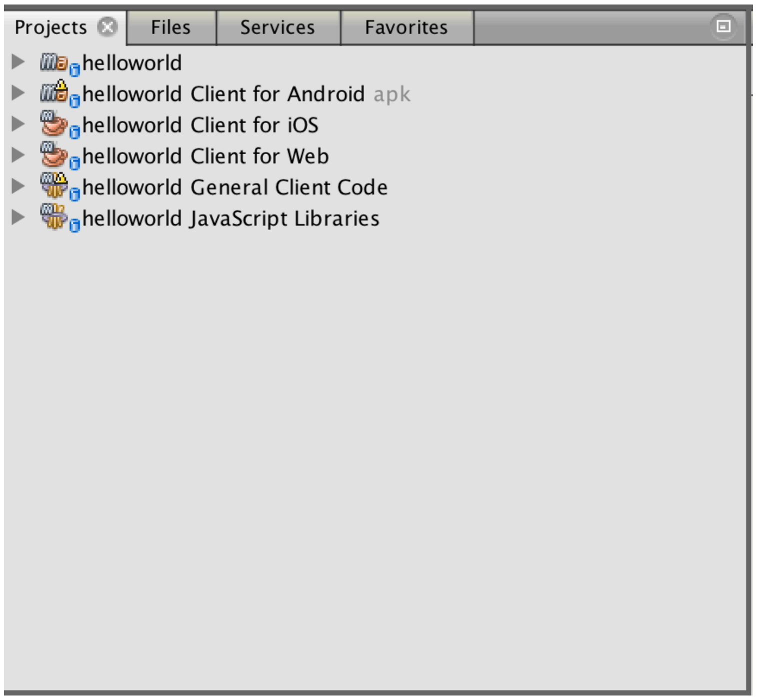
Description:

HTML page with Knockout.js bindings driven by an application model written in Java. Use your favorite language to code. Use HTML as a lightweight rendering toolkit. Deploy to desktop, Android, iOS, HTML5 browser.

[✓] Install sample code

Select Archetype

After the project has been created the Wizard will automatically run a priming build which downloads all the required artefacts. When this build is finished you're ready to run the sample application.



helloworld
helloworld Client for Android apk
helloworld Client for iOS
helloworld Client for Web
helloworld General Client Code
helloworld JavaScript Libraries

Generated Projects

NetBeans Plugins with DukeScript

By the way: The Project creation Wizard you just used in NetBeans is itself a DukeScript application. You can seamlessly integrate DukeScript into Swing and JavaFX.

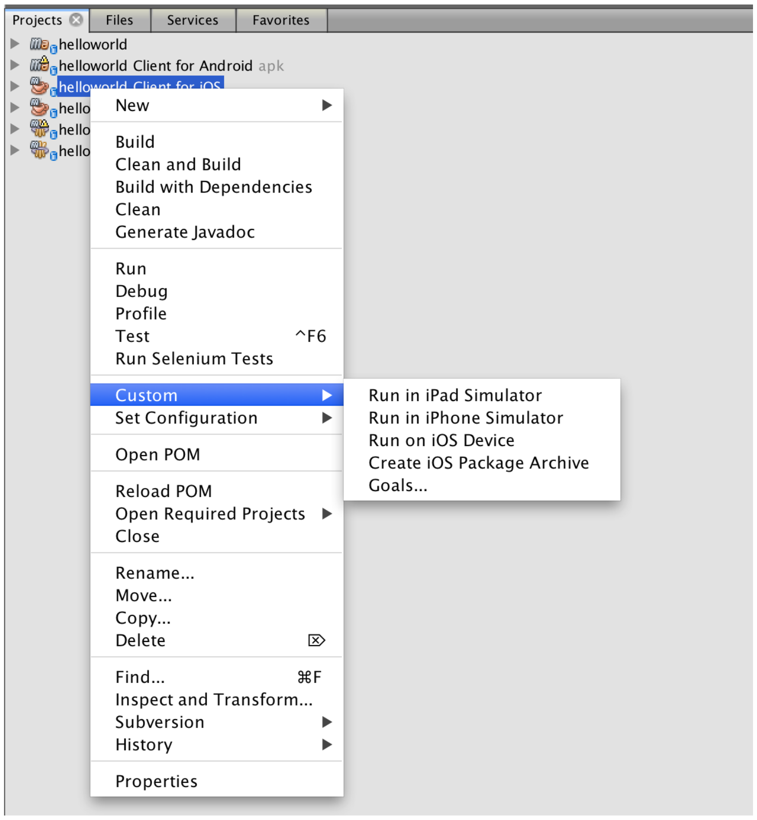
The Generated Projects

Maven has created several projects for you with the help of the archetype:

- helloworld
  - helloworld Client for Android
  - helloworld Client for iOS
  - helloworld Client for Web
  - helloworld General Client Code
  - helloworld JavaScript Libraries

The **helloworld** project is the parent project. It has common configuration for the subprojects and can be used to build all the subprojects. The "General Client Code" is the project that contains the actual Java code and the view definition. Use this to run, develop, test and debug your project.

For each target platform you selected in the wizard, there will also be a separate project. These projects can be used to deploy, debug and test the project to the individual platforms. In NetBeans each of the projects context menu has a submenu "custom" with entries that apply only to the deployment platform. For example the iOS project has an entry that allows you to run in an iPad simulator or deploy to a real iOS device, while the Android project has entries for running on an attached Android device.



Custom Actions

There's also a separate project for JavaScript Libraries. If you want to create your own API, or if you want to make calls from Java to JavaScript and back, this is where the code should go.

Summary

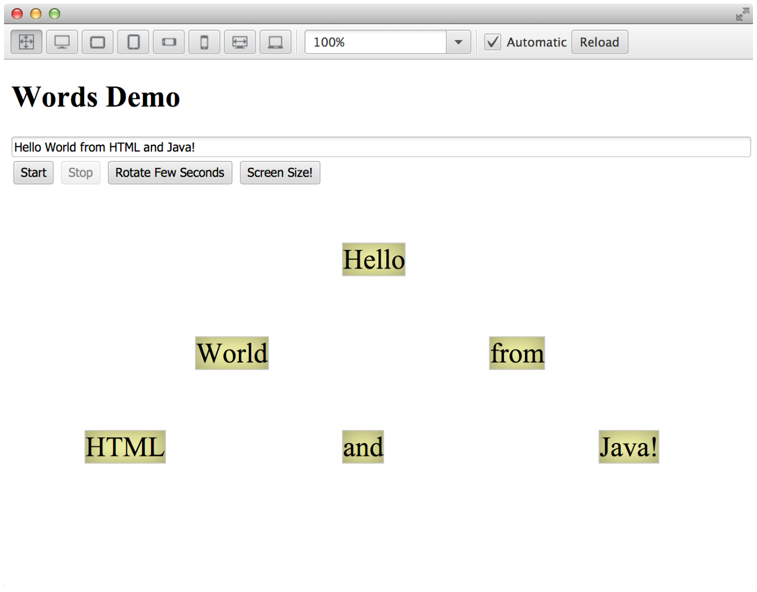
That's it. By now you've learned how to create a new DukeScript project using your preferred development environment. Now let's have a closer look at the sample application!

2. Understanding the sample application

Sometimes a demo is worth a thousand words, and often it's easier to understand the concepts of a technology when looking at an example and figuring out how the parts play together. In this part of the lab I'll try to guide you through that process. We'll have a look at how the sample application works. You'll learn how View and ViewModel can work together while still being cleanly separated.

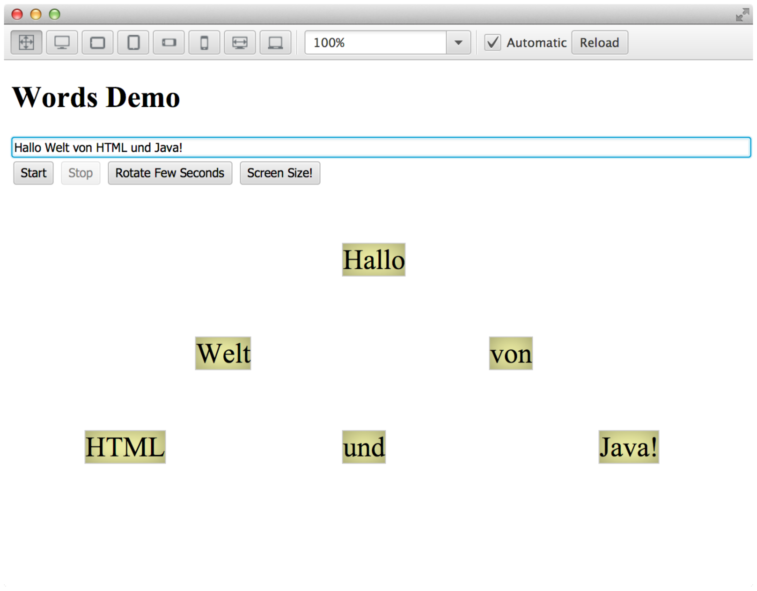
Running the application

I assume you have completed the project creation from the first exercise. Now open the project that ends with "helloworld General Client Code". If you're inside NetBeans you can launch the application from the projects context menu. So right-click and select the **run** action. The application starts and you should see something like this:



The demo application

You can now play with the application. Below a little form there are six boxes with words. These boxes show the text in the textfield. If you change that Text the words in the boxes will also update immediately. If you click on the **start** button, the button will be disabled and the words will start rotating. Also the **stop** button will be enabled now. you can use it to stop the rotation and reenable the **start** button. You need to confirm that action in a dialog. There's another button that will rotate the boxes for just a few seconds and a button that will get the screen size and display it in the textfield. So despite the simplicity of the UI there's actually a lot to investigate, because all of the logic is written in Java using the DukeScript APIs.



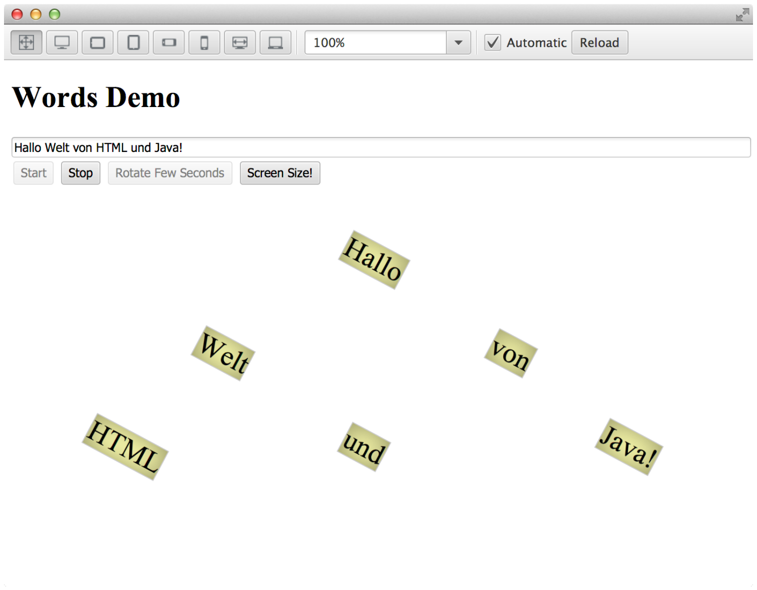
Enter archeype

The View

Let's first have a look at how the view is defined, how the animation works and how the view is bound to the model.

Animation via CSS

In DukeScript, the view is not created in Java code, but in a declarative format, namely in HTML. In our example project the view is defined in the file 'src/main/webapp/pages/index.html'.



Animation via CSS

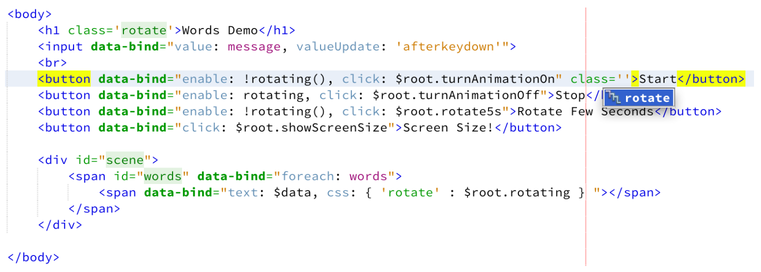
In the Project Tab, you can easily locate it under 'Web Pages/pages'. The page is just simple HTML. To preview it, right-click it and choose 'View' from the Menu. It will open in a browser. Now open the file and have a look at it's content. The head contains some CSS:

```css
@-webkit-keyframes spin {
    0% { -webkit-transform: rotate(0deg); }
    100% { -webkit-transform: rotate(360deg); }
}

input {
    width: 100%;
}

.rotate {
    -webkit-animation-name: spin;
    -webkit-animation-duration: 3s;
    -webkit-animation-iteration-count: infinite;
    -webkit-animation-direction: alternate;
}

#scene {
    position: relative;
    top: 60px;
    text-align: center;
}
```

```

#words span {
    border: 1px solid #ccc;
    background: rgba(255,255,155,0.8);
    text-align: center;
    font-size: 30px;
    -webkit-box-shadow: inset 0 0 40px rgba(0,0,0,0.4);
    position: absolute;
}

#words span:nth-child(1) { left: 45%; top: 0px; }
#words span:nth-child(2) { left: 25%; top: 100px; }
#words span:nth-child(3) { left: 65%; top: 100px; }
#words span:nth-child(4) { left: 10%; top: 200px; }
#words span:nth-child(5) { left: 45%; top: 200px; }
#words span:nth-child(6) { left: 80%; top: 200px; }
```

For simplicity the File contains the styles in-line, but you can also define your CSS in a separate File. Here we've defined the positions of the words, and the spinning animation. This is just standard CSS. Later in the example you'll see how the class 'rotate' is assigned to some elements to make them spin. To try out this style-class just assign it to one of the Buttons. Place the cursor inside one of the buttons and type "class='rotate'".

Note how NetBeans helps you by offering you the class in code-completion:



```
<body>
    <h1 class='rotate'>Words Demo</h1>
    <input data-bind="value: message, valueUpdate: 'afterkeydown'">
    <br>
    <button data-bind="enable: !rotating(), click: $root.turnAnimationOn" class=''>Start</button>
    <button data-bind="enable: rotating, click: $root.turnAnimationOff">Stop</
    <button data-bind="enable: !rotating(), click: $root.rotate5s">Rotate Few Seconds</button>
    <button data-bind="click: $root.showScreenSize">Screen Size!</button>

    <div id="scene">
        <span id="words" data-bind="foreach: words">
            <span data-bind="text: $data, css: { 'rotate' : $root.rotating } "></span>
        </span>
    </div>

</body>
```

Enter archeype

Now preview the page again using right-click -> 'View'. You will see the effect of adding this class: The Button rotates.

Binding the View to the Model

Let's remove the class again and have a look at the rest of the file. First there's a text input and some buttons. Each of these elements has a 'data-bind' attribute. These attributes are used to bind attributes of the Element to Properties of the data-model. The basic idea behind this is that whenever the bound Properties of the data model change, the view will update itself automatically. The binding also works in the opposite direction. We can define it so that e.g. entering a text in the input will automatically update a bound property in the data model. The bindings are only defined in the view. When we look at the datamodel later, you'll see that it doesn't have to reference the view at all. This technique is known as the Model-View-ViewModel (MVVM) Pattern.

Binding with Knockout

DukeScript uses Knockout, a very popular JavaScript library for data binding. Knockout is very well documented, and we suggest that you explore our documentation[5] if you want to learn more about it. For now it's sufficient to know what the 'data-bind'-attributes in our example are doing. The input-Element has this data-bind directive:

```
data-bind="textInput:  message"
```

This means that the value of the input field is bound to the 'message' property of the datamodel. This binding is bidirectional. When the datamodel's message-Property changes, the input will update. And when the user enters some text in the Textfield (and thus modifies the 'value'-attribute), this will update the datamodels 'message'-Property.

Enabling and disabling Buttons via binding

Next we have the 'Start' and 'Stop' Buttons. Their data-bind directives look very similar. Both bind their 'enable'-Attribute to the rotating-Property of the data model. The 'enable'-Attribute of a Button is a boolean. The Stop-Button is enabled, when the 'rotating'-property of the datamodel is 'true'. If the value in the data-model changes to 'false', the Button is automatically disabled as a result. The Start-Button is enabled, when the 'rotating'-Property is 'false'. The boolean negation is indicated by the leading exclamation mark. Both Buttons also have a 'click'-Binding. With this binding we declare what happens when the button is clicked. Here a method of the datamodel is called in return. Datamodels can be nested; with '$root' the root of the datamodel is referenced. In our case either the method 'turnAnimationOn' or 'turnAnimationOff' is called. The 'Rotate Few Seconds'-Button is similar, it only calls a different Method called 'rotate5s'. Another Button calls the Method 'showScreenSize'.

The Element with the id 'words' has a 'foreach'-Binding. The Property it binds to is words. The 'foreach'-Binding obviously requires an Array-Property, and it does something for each Element of the Array. What it does is defined in the inner span-Element: The 'text'-Attribute is bound to the Element ('$data'). With knockout we can also bind CSS-classes of an Element to Model-Properties. In our case the class 'rotate' is added when the 'rotating'-Property of the datamodel is 'true'. We've already tried out the effect of this class. So it's obvious what happens when we set the datamodels 'rotating'-Property to 'true'.

[5]https://dukescript.com/knockout4j.html

The Model

In a ‘normal’ Knockout-Application the model-class referenced from the view is defined as JavaScript. DukeScript changes that in order to make the application better maintainable, testable and to leverage the superior tooling of Java.

Simple Properties

Let’s have a look at the DataModel now. It’s defined in the File ‘DataModel.java’. The package it sits in depends on the Maven-Coordinates you entered during Project creation. You can find it in the ‘Source Packages’-Folder of the Project. The DataModel is annotated on class-level:

```java
@Model(className = "Data", properties = {
    @Property(name = "message", type = String.class),
    @Property(name = "rotating", type = boolean.class)
})
```

The ‘Model’-Annotation is evaluated at compile time. As a result a class is created. The name of this class is defined via the ‘classname’-Attribute. In our case the classname is ‘Data’. In the Project-View of NetBeans you can find the generated class ‘Data.java’ in ‘Generated Sources’. The benefit of this approach is that you don’t need to write Setters or Getters for your Properties, or care about the special requirements of a Model that works with Knockout. Instead you define the Properties via the Annotations ‘properties’-Attribute. In our case there’s a Property ‘message’ of type String, and a boolean-Property named ‘rotating’. We’ve already seen these two Properties in our View Definition.

Computed Properties

But the View also uses a Property called ‘words’.

```java
@ComputedProperty static java.util.List<String> words(String message) {
    String[] arr = new String[6];
    String[] words = message == null ? new String[0] : message.split(" ", 6);
    for (int i = 0; i < 6; i++) {
        arr[i] = words.length > i ? words[i] : "!";
    }
    return java.util.Arrays.asList(arr);
}
```

This Property is not defined in the Annotation. The reason for this is, that the Array of Strings returned from this method is computed by splitting the ‘message’-Property into single words. This cannot be defined in a simple Annotation. Instead we wrote a static Method of that name that does the computation. We annotated it with ‘@ComputedProperty’. So the value of the ‘words’-Property referenced in the View is the String-Array returned from this method.

The ‘Function’-Annotation

You already saw that the view also calls a couple of Methods of the Model. Like the ComputedProperties these Methods are also defined as static Methods and annotated with ‘@Function’. The Method ‘turnAnimationOn’ simply sets the rotating-Property to ‘true’. So it becomes obvious now, how the control flow works. The user clicks the Start-Button. This calls the Method ‘turnAnimationOn’. The Method in turn changes the value of the ‘rotating’-Property:

```
@Function static void turnAnimationOn(Data model) {
    model.setRotating(true);
}
```

As a result the CSS-class ‘rotate’ is added to the spans created from the words, and they start rotating. What you’ve seen up to here is the basic usage of the APIs. The idea behind the framework is, that a normal developer does not have to deal with JavaScript at all. But sometimes it’s required to integrate other JavaScript functions. In the workflow we’ve developed for DukeScript this is part of the role of API-Designers. There’s also an example of that in our demo application. Open the project with a name ending in “JavaScript Libraries”. It contains a class that shows how we define an API in DukeScript. The class Dialogs provides a way to invoke javascript functions from Java.

```
/** Shows direct interaction with JavaScript */
@net.java.html.js.JavaScriptBody(
    args = { "msg", "callback" },
    javacall = true,
    body = "if (confirm(msg)) {\n"
         + "  callback.@java.lang.Runnable::run()();\n"
         + "}\n"
)
static native void confirmByUser(String msg, Runnable callback);
```

The ‘@JavaScriptBody’-Annotation

The @JavaScriptBody-Annotation makes it easy to execute JavaScript-Code. To use it you need to define a static native Method. The Annotation has a parameter called ‘body’ which defines the JavaScript to execute. The value returned from the Method call can either be a Java String or primitive Type, or a JavaScript-Object. The Method ‘showScreenSize’ that we call from the view in turn calls such a ‘native’ Method called ‘screenSize’. This Method returns a String which is automatically converted to a Java String.

The @JavaScriptBody-Annotation also makes it easy to pass arguments to the JavaScript. This is done via the ‘args’ parameter. The Method ‘confirmByUser’ takes a String and a Runnable as arguments. You can see in the Annotation that these two Arguments are declared in the ‘args’-Parameter of the Annotation. Like this we can use them in the JavaScript. In case of a Java Object, we can even call it's methods following a standardized syntax. This is how the Runnable is executed from the confirmation Dialog. We'll have a closer look at this syntax later. The @JavaScriptBody Annotation makes calls to JavaScript typesafe for the caller and hides the JavaScript from 'normal' users. This makes applications maintainable and the Compiler can find Bugs at compiletime that would otherwise occur at Runtime and require extensive testing.

Putting it all together

Now that we have looked at Model and View and how they play together, the only thing that is missing is how to bind them together. This is the purpose of the Main class:

```java
public final class Main {
    private Main() {
    }

    public static void main(String... args) throws Exception {
        BrowserBuilder.newBrowser().
            loadPage("pages/index.html").
            loadclass(Main.class).
            invoke("onPageLoad", args).
            showAndWait();
        System.exit(0);
    }

    /**
     * Called when the page is ready.
     */
    public static void onPageLoad() throws Exception {
        Data d = new Data();
        d.setMessage("Hello World from HTML and Java!");
        d.applyBindings();
    }

}
```

The main method of this class creates the main Window using BrowserBuilder. What happens behind the scenes here depends on the platform. The platform specific HTML-Renderer-Window is created (newBrowser()) and loads the view using the loadPage method. With loadclass the Controller is created, and with invoke we can define a Method to be invoked when the Page is loaded The 'onPageLoad'-Method referenced here is responsible for creating the Model and initializing the binding (applyBindings).

Summary

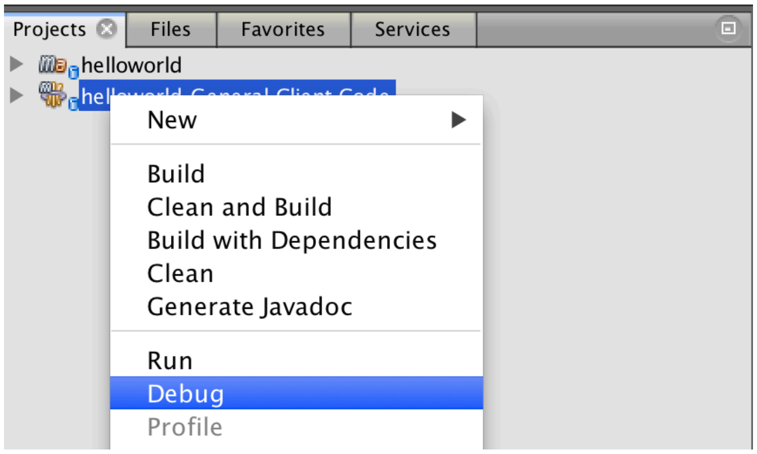
This is how the example application works and how the parts play together. Now that you have a basic understanding of the architecture of a typical DukeScript application and the Knockout API for Java.

4. Debugging a DukeScript application

You now know how to create a project and run an application. I've shown you how the application works. Now it's time to learn how to debug it. We'll first have a look at how to debug the Java Code of the application. Then we'll have a look at visual Debugging using the NetBeans Inspector.

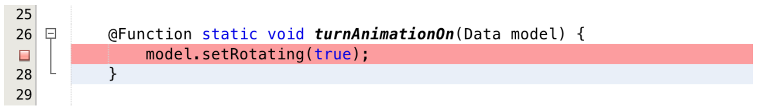
Debugging Java Code

The Maven projects created with our archetypes are configured for debugging with NetBeans, so you don't need to do anything but start the application in debug mode. You either do that by selecting *debug* from the projects context menu, or you can use the debug button in the toolbar while the project is selected.



Launching in debug mode

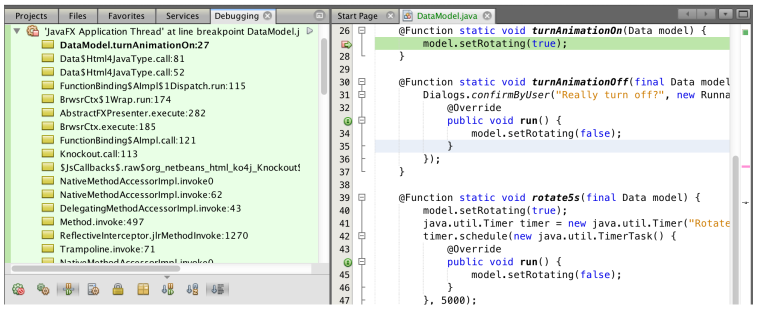
Let's set a breakpoint now. Open the project "helloworld General Client Code" and in the *DataModel* class look for the method *turnAnimationOn*. You can set a breakpoint by clicking in the grey bar left of the editor.



```
    @Function static void turnAnimationOn(Data model) {
        model.setRotating(true);
    }
```

Setting a breakpoint

If you click the *start* button of the application this function will be invoked and program execution will stop at the breakpoint. In the Debugging window you can inspect the stack.

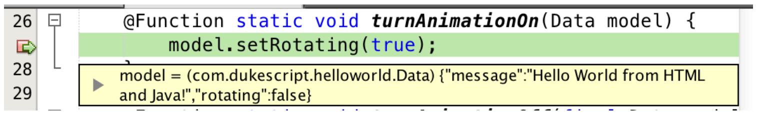


```
'JavaFX Application Thread' at line breakpoint DataModel.j
    DataModel.turnAnimationOn:27
    Data$Html4JavaType.call:81
    Data$Html4JavaType.call:52
    FunctionBinding$AImpl$1Dispatch.run:115
    BrwsrCtx$1Wrap.run:174
    AbstractFXPresenter.execute:282
    BrwsrCtx.execute:185
    FunctionBinding$AImpl.call:121
    Knockout.call:113
    $JsCallbacks$.raw$org_netbeans_html_ko4j_Knockout$
    NativeMethodAccessorImpl.invoke0
    NativeMethodAccessorImpl.invoke:62
    DelegatingMethodAccessorImpl.invoke:43
    Method.invoke:497
    ReflectiveInterceptor.jlrMethodInvoke:1270
    Trampoline.invoke:71
```

```
    @Function static void turnAnimationOn(Data model) {
        model.setRotating(true);
    }

    @Function static void turnAnimationOff(final Data model
        Dialogs.confirmByUser("Really turn off?", new Runna
            @Override
            public void run() {
                model.setRotating(false);
            }
        });
    }

    @Function static void rotate5s(final Data model) {
        model.setRotating(true);
        java.util.Timer timer = new java.util.Timer("Rotate
        timer.schedule(new java.util.TimerTask() {
            @Override
            public void run() {
                model.setRotating(false);
            }
        }, 5000);
```

Stopped on breakpoint

The method takes an instance of *Data* as an argument. When hovering over the variable you can see the content of the model as a JSON String. That's because the *toString* method of the model class automatically produces JSON.

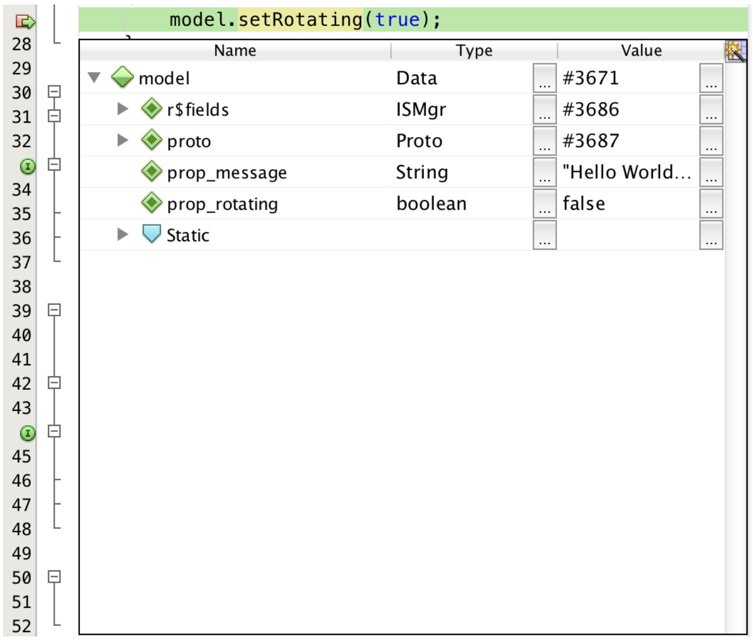


```
    @Function static void turnAnimationOn(Data model) {
        model.setRotating(true);
```

```
model = (com.dukescript.helloworld.Data) {"message":"Hello World from HTML
and Java!","rotating":false}
```

Inspecting the Model Object

If you click the little grey triangle to the left of the tooltip the window expands and you can further inspect the content of the model object.



```
        model.setRotating(true);
```

Name	Type	Value
model	Data	#3671
r$fields	ISMgr	#3686
proto	Proto	#3687
prop_message	String	"Hello World…
prop_rotating	boolean	false
Static		

Inspecting Variables

You can now use the toolbar buttons or the keyboard shortcuts to "step over", "step over expression", "step into", "step out" like in any regular Java application. You can find a detailed guide on all the actions in The 'NetBeans Field Guide'[6].

Hot Swapping Java Code

Debugging is nice, but nothing special. With DukeScript we can do much better. DukeScript supports hot swapping. While the application is running, you can change code and it will take immediate effect.

Start the project "helloworld General Client Code" and open DataModel.java in the editor. While the application is running locate any of the methods that can make a visible change in the ui.

For example you can change the text in the method turnAnimationOff from "Really turn off?" to "Really stop animation?" while the application is running. Now save your change, start the animation and hit the button. Your new text will be displayed. You could also change the words method and return all words in uppercase:

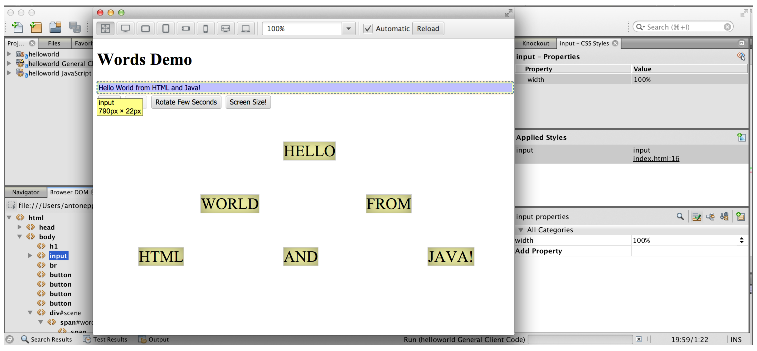
```
@ComputedProperty static java.util.List<String> words(String message) {
        String[] arr = new String[6];
        String[] words = message == null ? new String[0] : message.toUpperCase()\
.split(" ", 6);
        for (int i = 0; i < 6; i++) {
            arr[i] = words.length > i ? words[i] : "!";
        }
        return java.util.Arrays.asList(arr);
    }
```

Don't forget to save your change. The application will continue to run, but when you change the input String all words will be in upper case. The cool thing is, that the application will not redeploy, but keep it's state. So you can develop your application while it's running.

Visual Debugging

Now you know how to debug your ViewModel and hot swap code, but what about the View? When developing in NetBeans debugging the view is very simple and comfortable. When you run the application you'll see a couple of Tabs pop up. One of them is labeled "Browser DOM". This tab shows the live DOM tree of the running application. Select an Element there and it will be highlighted in the running application.

[6]https://netbeans.org/project_downloads/usersguide/nbfieldguide/Chapter5-Debugging.pdf



file:///Users/antonep

```
html
  head
  body
    h1
    input
    br
    button
    button
    button
    button
    div#scene
      span#wor
```

input – Properties

Property	Value
width	100%

Applied Styles

input	input index.html:16

input properties

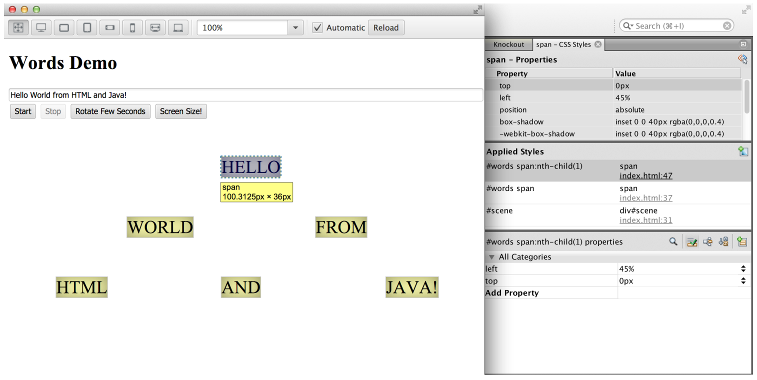
All Categories

width	100%
Add Property	

Live DOM Tree inspection

On the top left of the “Browser DOM”-tab there is a little button (dashed rectangle) that starts the “Inspect Mode” in the Browser. If you activate it, you can mouse over the elements in the browser and they will be highlighted and also selected in the DOM tree. There’s a useful little tooltip showing the size of the element.

But there’s more. Another new tab that was opened when launching the application is the “CSS-Styles” tab. It will show all relevant CSS style attributes of our project and where they come from. Try it out and select the word “Hello” (which is a span element).



span – Properties

Property	Value
top	0px
left	45%
position	absolute
box-shadow	inset 0 0 40px rgba(0,0,0,0.4)
-webkit-box-shadow	inset 0 0 40px rgba(0,0,0,0.4)

Applied Styles

#words span:nth-child(1)	span index.html:47
#words span	span index.html:37
#scene	div#scene index.html:31

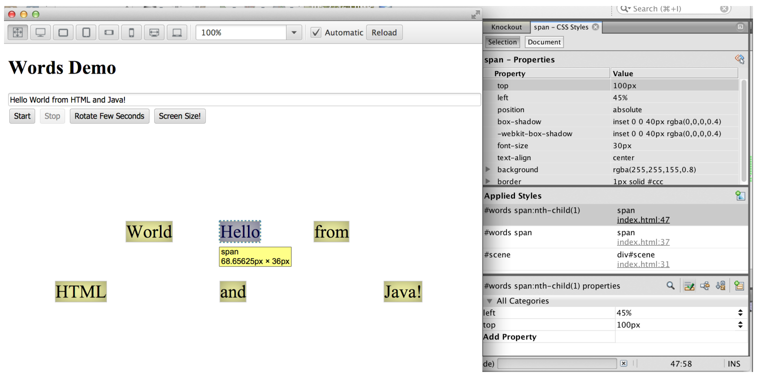
#words span:nth-child(1) properties

All Categories

left	45%
top	0px
Add Property	

Live DOM Tree inspection

On top of the “CSS-Styles” tab you’ll find the properties of the span. You can see the current value of the property, for example the property top has a value of 0px for the word “Hello”. The context menu of this property has the action “Go to Source” which will open index.html at the position where the inline styles are defined. You can now edit the HTML-file and change this value to 100px. When you save this change, the UI will update, and the Span will be relocated.



span – Properties

Property	Value
top	100px
left	45%
position	absolute
box-shadow	inset 0 0 40px rgba(0,0,0,0.4)
-webkit-box-shadow	inset 0 0 40px rgba(0,0,0,0.4)
font-size	30px
text-align	center
background	rgba(255,255,155,0.8)
border	1px solid #ccc

Applied Styles

#words span:nth-child(1)	span index.html:47
#words span	span index.html:37
#scene	div#scene index.html:31

#words span:nth-child(1) properties

All Categories

left	45%
top	100px
Add Property	

Live DOM Tree inspection

You can do a lot more things with this little subwindow in the tab, for example show the properties for different pseudo classes (states) of your element or inspect the stylesheet hierarchy.

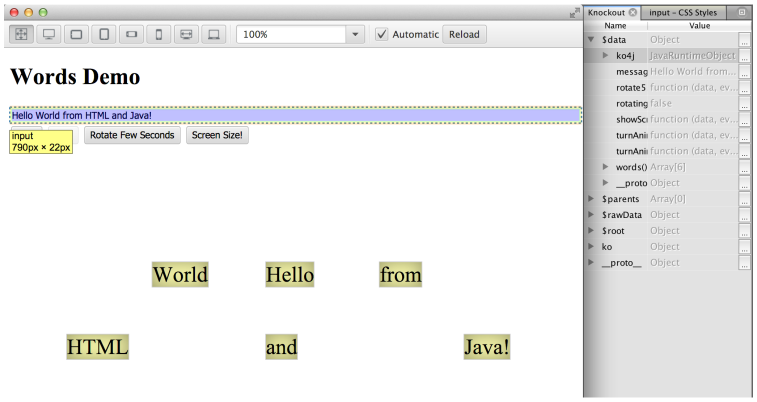
In the middle pane you can see the applied styles listed by name, and you can jump directly to the CSS-File where they are defined (or in our case to the html file).

The lower pane displays all the properties of the style class that is currently selected in the middle pane. You can also remove or change a property or add a new one. You will even get a list of all the properties relevant for this element.

Knockout Tab

Another tab is labeled "Knockout". It shows you the knockout context of the active element. This is really useful to analyze where you are in your viewmodel hierarchy, especially when using deeply nested viewmodels.

The value for "$data" shows you the current context. If you select the body, or the input element, you can see the current context is the full Data model. It is displayed as "JavaRuntimeObject", and you can see it's Properties and functions.

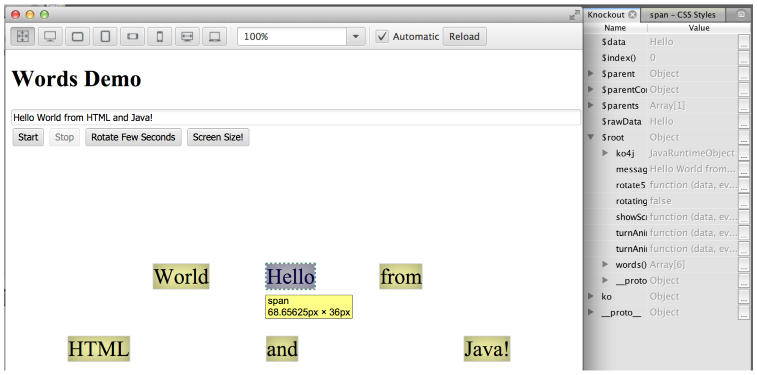


Name	Value
$data	Object
ko4j	JavaRuntimeObject
messag	Hello World from...
rotate5	function (data, ev...
rotating	false
showSc	function (data, ev...
turnAni	function (data, ev...
turnAni	function (data, ev...
words()	Array[6]
__proto	Object
$parents	Array[0]
$rawData	Object
$root	Object
ko	Object
__proto__	Object

Debugging Knockout Contexts

If you select the “Hello”-Span instead, the context is “Hello”. That is because you’re inside the “forEach” loop now, and the context of each element is one member of the “words” array.

You can also see that it’s $parent is an array, and that the $root context is our full Data instance.



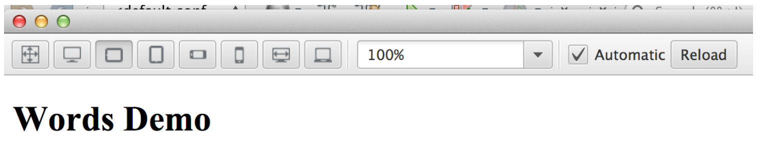
Name	Value
$data	Hello
$index()	0
$parent	Object
$parentCo	Object
$parents	Array[1]
$rawData	Hello
$root	Object
ko4j	JavaRuntimeObject
messag	Hello World from...
rotate5	function (data, ev...
rotating	false
showSc	function (data, ev...
turnAni	function (data, ev...
turnAni	function (data, ev...
words()	Array[6]
__proto	Object
ko	Object
__proto__	Object

Debugging Knockout Contexts

The property names $root and $parent are also the names you can use in the data binding to access the parent or the root model. This is important if you want to call a method on them from a nested level. You’ll learn more about the scope of knockout viewmodels and their scope in the next chapter.

Device sizes

If you launch the application from NetBeans, there’s a toolbar on top. You can use this toolbar to test your UI at various device sizes.

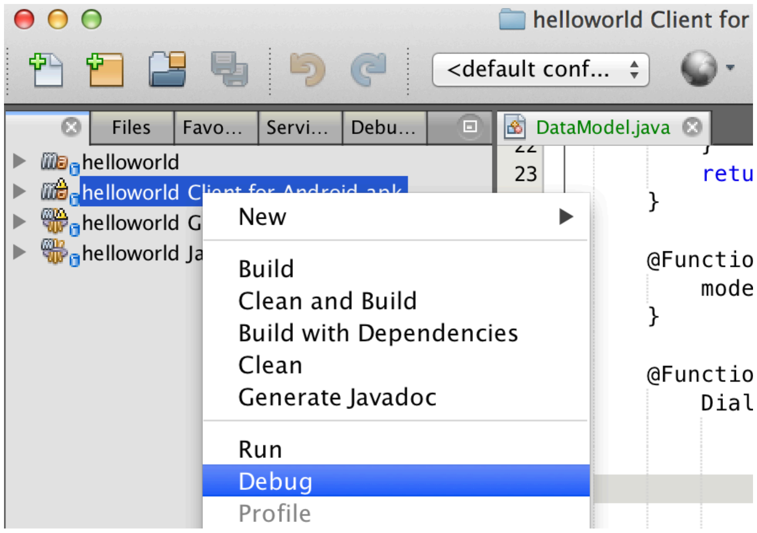


Testing Device sizes

Debugging on Android Devices

DukeScript also supports debugging of the deployed application on Android. If you have a device connected via USB to your development computer, you need to enable USB debugging first. On most devices running Android 3.2 or older, you can find the option in Settings > Applications > Development. On Android 4.0 and newer, it’s in Settings > Developer options. On Android 4.2 and newer, Developer options is hidden by default. To make it available, go to Settings > About phone and tap Build number seven times. Then return to the previous screen to find Developer options.

Open the project “helloworld Client for Android apk”. It has the action “debug” in it’s context menu.



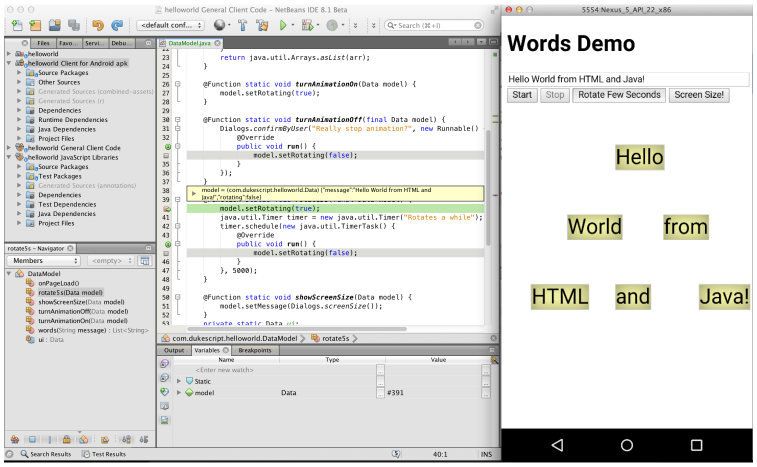
Debug Android app

If you invoke it, the build will look for attached devices. It will print a message when your device was found and launch the application in debug mode. Now you can set breakpoints like in a desktop app.

You can also use an emulated device instead. Just use the android AVD manager to create and/or start the virtual device. Then invoke the debug action. It will work exactly the same, but it takes a bit longer. On the lab computers, there's already a predefined device.

Use the File Explorer to navigate to "C:/Program Files(x86)/Android/Android SDK". Double-click the SDK Manager.exe file.

In the Menu, go to "Tools" and select "Manage AVDs". Now select the device "nex4" and afterwards click the start button. Please wait until the device is fully booted before you continue with the task.


```
        return java.util.Arrays.asList(arr);
    }

    @Function static void turnAnimationOn(Data model) {
        model.setRotating(true);
    }

    @Function static void turnAnimationOff(final Data model) {
        Dialogs.confirmByUser("Really stop animation?", new Runnable()
            @Override
            public void run() {
                model.setRotating(false);
            }
        });
    }
model = (com.dukescript.helloworld.Data) {"message":"Hello World from HTML and Java!","rotating":false}
        model.setRotating(true);
        java.util.Timer timer = new java.util.Timer("Rotates a while");
        timer.schedule(new java.util.TimerTask() {
            @Override
            public void run() {
                model.setRotating(false);
            }
        }, 5000);
    }

    @Function static void showScreenSize(Data model) {
        model.setMessage(Dialogs.screenSize());
    }
```

Name	Type	Value
<Enter new watch>		
Static		
model	Data	#391

Stopped at breakpoint on Android Emulator

Debugging JavaScript Code

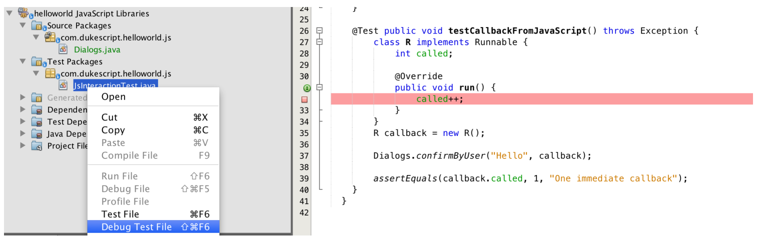
Sometimes in your application you need to write JavaScript code as well. A prerequisite for this to work is that you're using JDK 8 for this example, because we need to make sure that the Nashorn Scripting engine is running our JavaScript code. In our demo application there is JavaScript code to open a dialog. If you close the dialog by pressing the 'ok'-button a callback will be executed. You can debug this inside a unit test. If you open the project "helloworld JavaScript Libraries" you will find a Unit Test called "JsInteractionTest". One of the methods tests this behaviour.

In order to run the unit test you need to setup the JavaScript Environment by activating the Script Presenter. This is a very simple implementation of the Presenter service provider interface that uses the Java Scripting API[7] to execute the JavaScript Code that is normally executed in the browser.

These lines in the unit test are responsible for setting this up and shutting it down:

```
@BeforeMethod public void initializeJSEngine() throws Exception {
    jsEngine = Fn.activate(Scripts.createPresenter());
}

@AfterMethod public void shutdownJSEngine() throws Exception {
    jsEngine.close();
}
```

Locate the method "testCallbackFromJavaScript" and set a breakpoint on the line where the "called" counter is increased inside the Runnable. Then rightclick the test and from the context menu execute "Debug Test File".

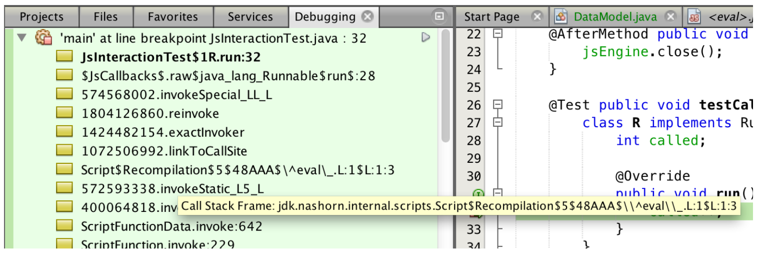


```
    }

    @Test public void testCallbackFromJavaScript() throws Exception {
        class R implements Runnable {
            int called;

            @Override
            public void run() {
                called++;
            }
        }
        R callback = new R();

        Dialogs.confirmByUser("Hello", callback);

        assertEquals(callback.called, 1, "One immediate callback");
    }
}
```

Mixed Debugging of JavaScript and Java

The test will stop at our breakpoint as expected. Now have a look at the call stack. You should find a stack element that has "eval" in it. To locate it you can hover over the stack elements and find the frame that has "jdk.nashorn.internal.scripts.Script" in it.

[7]http://docs.oracle.com/javase/8/docs/api/javax/script/package-summary.html



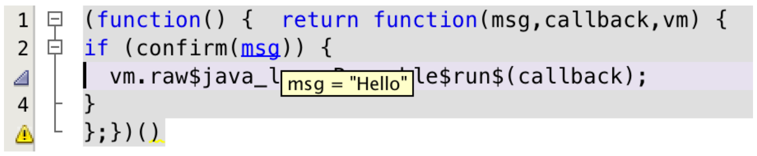
```
'main' at line breakpoint JsInteractionTest.java : 32
    JsInteractionTest$1R.run:32
    $JsCallbacks$.raw$java_lang_Runnable$run$:28
    574568002.invokeSpecial_LL_L
    1804126860.reinvoke
    1424482154.exactInvoker
    1072506992.linkToCallSite
    Script$Recompilation$5$48AAA$\^eval\_.L:1$L:1:3
    572593338.invokeStatic_L5_L
    400064818.inv
    ScriptFunctionData.invoke:642
    ScriptFunction.invoke:229
```

```
    @AfterMethod public void
        jsEngine.close();
    }

    @Test public void testCal
        class R implements Ru
            int called;

            @Override
            public void run()
```

```
Call Stack Frame: jdk.nashorn.internal.scripts.Script$Recompilation$5$48AAA$\\^eval\\_.L:1$L:1:3
```

Locating the JavaScript call

Doubleclick it and it will open a JavaScript file in the editor.



```
(function() {  return function(msg,callback,vm) {
if (confirm(msg)) {
  vm.raw$java_l        le$run$(callback);
}
};})()
```

```
msg = "Hello"
```

JavaScript Debugging

You can now mouse over the elements, and you will see the values of the variables. In the screenshot above you can see the value of the "msg" variable coming from the Java call. The method we have here is rather short, but if you have a longer method, you can also set breakpoints and step through the execution just like in a Java file.

So not only can you debug the nice and clean Java code you write, but even the otherwise hidden "native" calls under the hood.

Summary

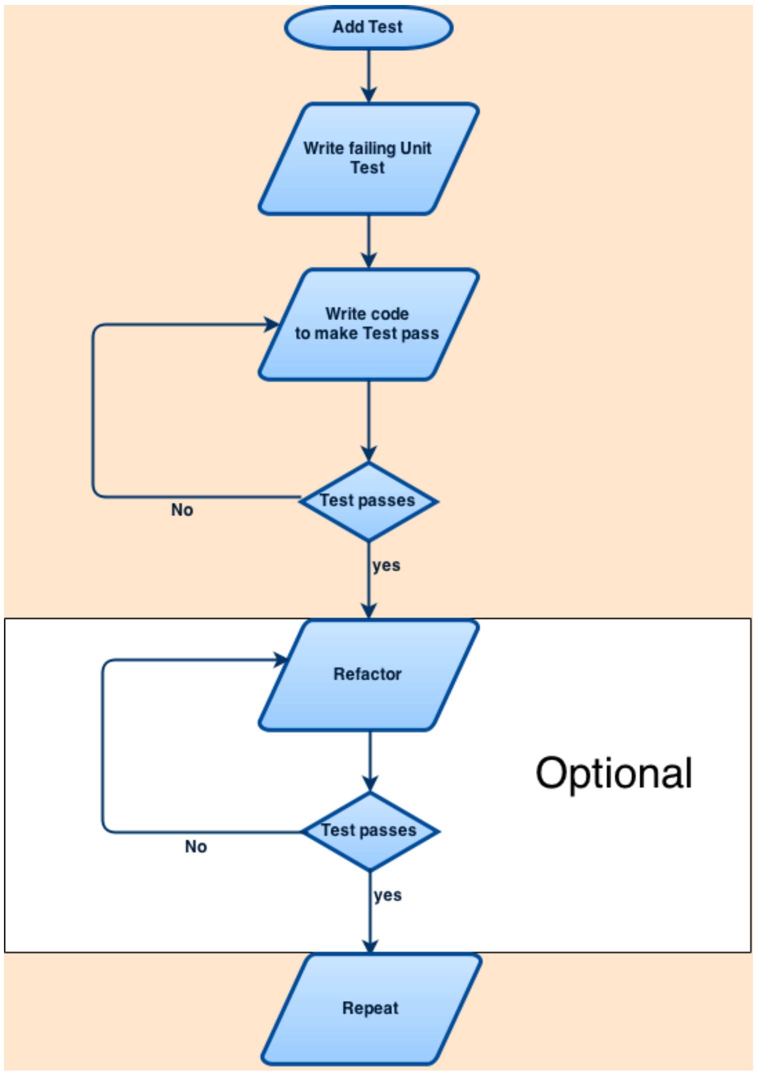
In this Exerceise you've learned how to debug all the aspects of a DukeScript application. We've inspected the Java code and did hot swapping while keeping the state of the application intact. We changed the appearance of the application while it was running and inspected the Dom and CSS. You've learned how to debug an Application on Android or iOS devices or in the emulator. And we had a look what is required to debug JavaScript code.

Testing

In this part of the lab you'll learn how to write unit tests for the ViewModel of your application. This allows you to test the view logic, even before you write the view.

Test Driven Development in a nutshell

DukeScript makes a clean separation of view logic and view in order to simplify testing and support Test Driven Development (TDD). In TDD, whenever you're implementing a feature you start by writing a test. Then quickly implement the tested functionality until the test - and all other tests in your project - succeed. As a last step clean up and refactor what you created. Then repeat this process with the next test. As soon as all the tests succeed you're done. That's TDD in a nutshell!



Alt Test Driven Development

In DukeScript we're using a Model View ViewModel[8] (MVVM) architecture. This has a lot of is perfect for Test Driven Development.

You simply start by writing a unit test for the ViewModel. Then you create the ViewModel class (if it doesn't exist yet) and implement the functionality to pass the test. Since the ViewModel knows nothing about the View, there's no need to bother implementing the View yet. You can perfectly test your UI without it. You can test the pure functionality. As a result your test code and the ViewModel itself will be very concise.

[8]http://en.wikipedia.org/wiki/Model_View_ViewModel

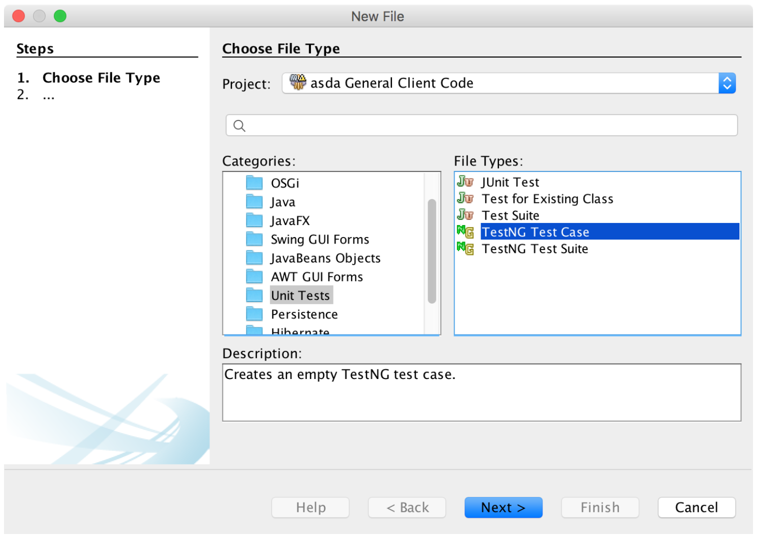
Unit Testing a ViewModel

We want to create a very simple Todo-List. The UI should have a TextField on top to enter new Items and a Button next to it to add them. The Button should only be enabled if the item entered is at least three letters long. Below these inputs we want to show the actual todo list. The user can select an item and delete it. This is done via a button below the list. It should only be enabled when an item is selected. When the button is pressed, the selected item should be removed from the list.

Start by creating a new DukeScript project as in Exercise 1. But in th elast Wizard step **don't** check the checkbox to **install sample code**.

We start with iteration 1. Let's test enabling and disabling the "add"-button first:

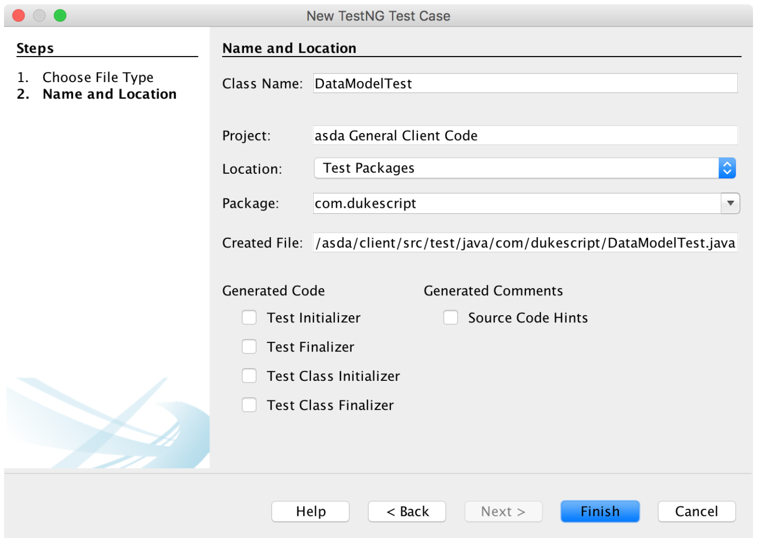
Open the Menu File and select "New File". This opens a dialog. Select category "Unit Tests", and "TestNG Test Case".



Creates an empty TestNG test case.

Wizard Step 2

In the next Wizard page choose a name ("DataModelTest") and a base package, and click finish.



Name and Location

Class Name: DataModelTest

Project: asda General Client Code

Location: Test Packages

Package: com.dukescript

Created File: /asda/client/src/test/java/com/dukescript/DataModelTest.java

Generated Code
- [ ] Test Initializer
- [ ] Test Finalizer
- [ ] Test Class Initializer
- [ ] Test Class Finalizer

Generated Comments
- [ ] Source Code Hints

Wizard Step 2

The generated class will open in the Editor, add the following Test method:

```java
import static org.testng.Assert.*;
import org.testng.annotations.Test;

public class DataModelTest {

    @Test
    public void addButtonEnabled() {
        TodoListViewModel model = new TodoListViewModel();
        model.setInputText("Bu");
        assertFalse(model.getAddEnabled());
        model.setInputText("Buy");
        assertTrue(model.getAddEnabled());
        model.setInputText("Bu");
        assertFalse(model.getAddEnabled());
    }
}
```

Now we need to write the ViewModel that passes this test. Create a new class named "TodoListViewModelDefinition" and add a Model Annotation like this. The addEnabled method is a ComputedProperty[9], just like the "words" property in our first exercise. It depends on the inputText Property and automatically will be reevaluated if the inputText changes:

[9]http://bits.netbeans.org/html4j/1.1/net/java/html/json/ComputedProperty.html

```java
@Model(className = "TodoListViewModel", targetId="", properties = {
    @Property(name = "inputText", type = String.class)
})
final class TodoListViewModelDefinition {

    @ComputedProperty static boolean addEnabled(String inputText){
        return (inputText != null && inputText.length() > 2);
    }

}
```

Execute the test, it should pass. That was fairly simple. We can skip the refactoring step, because the code looks OK. Ready for iteration two. Let's write another test. This time we test if the "add"-button does it's job:

```java
    @Test
    public void addButtonAdd() {
        TodoListViewModel model = new TodoListViewModel();
        assertEquals(model.getTodos().size(), 0);
        model.setInputText("bu");
        model.addTodo();
        assertEquals(model.getTodos().size(), 0);
        model.setInputText("buy milk");
        model.addTodo();
        assertEquals(model.getTodos().size(), 1);
        assertEquals("", model.getInputText());
    }
```

Now lets alter the ViewModel to also pass this test. The @ModelOperation[10] annotation used here ensures that we can call our method from any Thread, which simplifies the Testing:

```java
@Model(className = "TodoListViewModel", targetId = "", properties = {
    @Property(name = "inputText", type = String.class),
    @Property(name = "todos", type = String.class, array = true)
})
final class TodoListViewModelDefinition {

    @ComputedProperty
    static boolean addEnabled(String inputText) {
         return (inputText != null && inputText.length() > 2);
```

[10]http://bits.netbeans.org/html4j/1.1/net/java/html/json/ModelOperation.html

```
    }

    @Function
    @ModelOperation
    public static void addTodo(TodoListViewModel model) {
        if  (model.getInputText() != null && model.getInputText().length() > 2) {
            model.getTodos().add(model.getInputText());
            model.setInputText("");
        }
    }

}
```

Perfect all our tests pass. The “add”-function works. But the code is not nice. There’s a duplication for validating the input. We could check for isAddEnabled, but that would be ugly. First we might decide to add new entries via a different channel, and second the method name should reflect the purpose of the method. So let’s refactor it:

```
@Model(className = "TodoListViewModel", targetId = "", properties = {
    @Property(name = "inputText", type = String.class),
    @Property(name = "todos", type = String.class, array = true),
    @Property(name = "selectedItem", type = String.class)
})
final class TodoListViewModelDefinition {

        @ComputedProperty
    static boolean addEnabled(String inputText) {
        return isValidInput(inputText);
    }

    @Function
    @ModelOperation
    public static void addTodo(TodoListViewModel model) {
        if (isValidInput(model.getInputText())) {
            model.getTodos().add(model.getInputText());
            model.setInputText("");
        }
    }

    private static boolean isValidInput(String inputText) {
        return (inputText != null && inputText.length() > 2);
    }

}
```

Now it's time for the "delete"-button. We'll first test if it's correctly enabled and disabled:

```
    @Test
    public void deleteButtonEnabled() {
        TodoListViewModel model = new TodoListViewModel();
        assertFalse(model.isDeleteEnabled());
        model.add("buy milk");
        model.getSelected().add("buy milk");
        assertTrue(model.isDeleteEnabled());
    }
```

Again lets alter the ViewModel to pass this test;

```
@Model(className = "TodoListViewModel", targetId = "", properties = {
    @Property(name = "inputText", type = String.class),
    @Property(name = "todos", type = String.class, array = true),
    @Property(name = "selected", type = String.class, array = true)
})
final class TodoListViewModelDefinition {

    @ComputedProperty
    static boolean addEnabled(String inputText) {
        return isValidInput(inputText);
    }

    @Function
    @ModelOperation
    public static void addTodo(TodoListViewModel model) {
        if (isValidInput(model.getInputText())) {
            model.getTodos().add(model.getInputText());
            model.setInputText("");
        }
    }

    private static boolean isValidInput(String inputText) {
        return (inputText != null && inputText.length() > 2);
    }

    @ComputedProperty
    static boolean deleteEnabled(List<String> selected) {
        return (selected != null && selected.size() > 0);
    }
}
```

The code looks OK, no refactoring required. And finally we'll test if pressing the delete-button really removes the item from the list:

```
   @Test
    public void deleteButtonDelete() {
        TodoListViewModel model = new TodoListViewModel();
        model.getTodos().add("buy Milk");
        assertEquals(model.getTodos().size(), 1);
        model.getSelected().add("buy Milk");
        model.deleteTodo();
        assertEquals(model.getTodos().size(), 0);
        assertEquals(model.getSelected().size(), 0);
    }
```

Here are the required changes to the ViewModel:

```
    @Function
    @ModelOperation
    public static void deleteTodo(TodoListViewModel model) {
        List<String> selected = model.getSelected();
        for (String selected1 : selected) {
            model.getTodos().remove(selected1);
        }
        model.getSelected().clear();
    }
```

Done, all the tests pass, our ViewModel works. Now we can start developing the View part:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Todo List</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <input data-bind="textInput: inputText" />
        <button data-bind="enable: addEnabled, click: addTodo">add</button><br>
        <select style="width: 100%" size=10 data-bind="foreach: todos, selectedO\
ptions: selected">
            <option data-bind="value: $data, text:$data"></option>
        </select>
        <button data-bind="enable: deleteEnabled, click: deleteTodo">delete</but\
ton>

    </body>
</html>
```

All the bindings are declarative and simple. They don't have complex binding logic. That is an important part. If we start writing complex bindings, we will often unintentionally put UI logic into the View where it is hidden from our test, and we'll loose the benefits of TDD. But with simple bindings like in this case there's not much that can go wrong.

Obviously there is some cheating involved in this process ;-). I sometimes add the required Properties just before I write the test. That prevents unnecessary little bugs and typos like writing "getEnabled()" instead of "isEnabled()". But generally I like to write the test as a way to design the API. It forces me to take the API user perspective. That helps to design an API that is clean and simple.

You can also checkout this project on Github[11].

Summary

In this part of the lab you've learned everything you need to create an application with excellent code coverage which will make your boss and your customers happy. That's a nice side effect. But most importantly, you've seen how DukeScript makes it really simple to unit test most aspects of the application by using the MVVM pattern. Using these test patterns you'll be able to create applications that are easy to refactor, because everything is tested. And they are easy to understand as well, because unit tests are a great way of documenting how your APIs are used.

[11]https://github.com/dukescript/tdd-example

This is the last part of this lab. To learn more about DukeScript, please check out our Blog[12] and website[13]. Thanks for participating in this lab, and enjoy JavaONE!

[12] https://dukescript.com/blog.html
[13] https://dukescript.com