GETTING STARTED WITH ES6

ES6 or (ECMAScript 6, prevously called 'Harmony') is the sixth major release of the ECMAScript language specification and it is the specification that JavaScript also implements (along with languages such as JScript and ActionScript)

The proposal for ES6 has been created in order to give JavaScript the much needed rewamp - as JS is deemed to be the most used (and the most misunderstood) language of our time there needed to be sufficient changes in order to make the language work in a way that developers no longer need to hack their way around to get some basic functionality done.

WHY USE ES6?

This question comes up a lot of times - why would someone want to use ES6 at the first place?

ES6 is packed with features that will help you be more productive at writing JavaScript code. It will also help you write JavaScript using a better syntax - by using the `let` keyword instead of `var` you can now create block-scoped variables.

Throughout this course we will investigate some of the key features of ES6 and hopefully by the end of the course you'll have the urge to use it.

HOW TO RUN ES6?

Some ES6 features already exist in current browswers but some of course do not, however there are tools out there that can help you with running your ES6 code in an ES5 environment. Such tools are called JavaScript compilers and there are quite a few of them. The process of compiling in this case means that we take the source code of an ES6 application and we generate ES5 code from it.

Check out these compiler:

- Babel (previously known as '6to5')
- Traceur
- ES6-Shim

Throughout the course we will be using Babel - mostly the Babel CLI running in Node.js.

> To get a similar setup to ours, please install Babel globally using npm: `npm install -g babel`
>
> To verify a successful setup execute `babel --version`

You can of course use Babel through browser, make it part of build systems and test frameworks.

Once you have installed Babel you'll get two command line commands: `babel` and `babel-node`. We will

be using both (for example `babel-node` can run generators (it automatically includes the babel/polyfil library, wherease `babel` from the cli does not)

Babel has lots of options for the cli - whether you want to compile entire folders, pipe files or even compile with source maps.

For the purposes of this training we'll be using Babel to compile single files with the following syntax: `babel script.es6 --out-file script.es6.js`

> The naming convention throughout the training is consistent. ES6 files all have an .es6 extension and the compiled file will have either an .es5.js or a .es6.js extension)
>
> Another important note for compilation is that you need to pass in valid JavaScript to Babel. Sometimes missing semicolons can cause unexpected behaviour or the total lack of compilation.

Compatibility table

If you'd like to see which Browser / Compiler supports what ES6 features this page can help you out: https://kangax.github.io/compat-table/es6/

Text Editors

Most text editors would already support JavaScript/ES6 syntax highlighting. If yours does not, try to install a package or apply JavaScript syntax highglighting to your ES6 code.

LET

This unit will walk you through on how to use the `let` and `const` keywords.

SIMPLE EXAMPLE

You can think of the `let` keyword as you think of `var` in previous versions of JavaScript In ES6 you can use both `let` and `var` but you need to be aware of some peculiarities.

First of all let's have a look at a very straight forward example:

```
let msg = 'hello world';
console.log(msg); //hello world
```

The above is really not different from any other standard variable declaration using the `var` keyword. You can assign the same types: strigs, integers, objects so on and so forth.

BLOCKS

When you create a variable using `let` , the scope of that variable will be for the block where you've created

let.

See the following for an example:

```
if (true) {
    let msg = 'hello';
    console.log(msg); //hello
}
console.log(msg); //ReferenceError - msg variable doesn't exist
```

Another classic example is of course using the for loop. Here's an **ES5** example where we create a variable `i` in the for loop however accessing that variable is possible also outside the scope of the loop:

```
for (var i = 0; i < 5; i++) {
    console.log(i);
}
console.log('value of i is --> ' + i); //this will print 5
```

Using `let` this looks significantly different:

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
console.log('value of i is --> ' + i); //ReferenceError - you can no longer access i
```

`let` statements do not utilise variable hoisting:

```
if (true) {
  console.log(msg); //undefined
  let msg = 'hello';
}
```

`let` does not allow the redefinition of variables:

```
var msg = 'hello';
let msg = 'world'; //Duplicate declaration "message" (error is thrown during the Babel compilation)
```

However you can define variables with the same name providing the fact that they are in a block, take the following example:

```
var message = 'hello';
if (true) {
  let message = 'world';
  console.log(message); //world
}
console.log(message); //hello
```

CONSTANT

Constants work very similarly to `let` statements but there is an important difference. With constants you cannot redefine a variable, not matter if it is inside or outside a block.

To define a constant you can use the `const` keyword:

```
const MSG = 'hello';
```

This example highlights the fact that you cannot redefine a constant:

```
const MSG = 'hello';
MSG = 'world'; //MSG" is read-only (error is thrown during the Babel compilation)
```

```
const MSG = 'hello';
if (true) {
    MSG = 'world'; //MSG" is read-only (error is thrown during the Babel compilation)
}
```

ARROW FUNCTIONS

Probably the most exciting addition to ES6 are the arrow functions - they allow developers to use a new syntax to create functions (the synax is `=>` henceforth the name ‘arrow function’ or sometimes referred to as ‘fat arrow’)

Of course it doesn’t only let you create functions in a different way than before in JavaScript ES5 but they come with a set of new features that the JS community welcomed:

- The value of `this` remains the same throughout the function and it’s also lexically bound
- there’s no `arguments` object

CREATE A FUNCTION USING =>

To use the arrow notation it’s extremely simple to create functions.

If the function has no argument you can define your function in literally one single line:

```
var hello = () => 'hello world';
```

If you run this script through Babel you’ll notice that the generated ES5 script looks like this:

```
var hello = function hello() {
  return 'hello world';
};
```

Note that you don't need to define the `return` keyword if your function returns a single value or statement like above.

If you have an argument to pass in to the function you can ommit the parentheses and have this piece of code where `name` is the argument

```
var greet = name => 'Hello ' + name;
//console.log(greet('Tamas')); //outputs Hello Tamas
```

However if you have multiple arguments you need to specify those using the parentheses again:

```
var multiply = (n1, n2) => n1 * n2;
//console.log(multiply(2, 5)); //prints 10
```

We could rewrite the above to use curly braces as well which would feel a bit like ES5:

```
var multiply = (n1, n2) => {
    return n1 * n2;
}
```

Finally if your function returns an object literal for example, you need to wrap your return statement with parentheses as well:

```
var person = (name, age) => ({name: name, age: age});
//in ES this would be:
var person = function person(name, age) {
  return { name: name, age: age };
};
```

THIS BINDING

Let's discuss the point about `this` first a bit more in detail. We are going to look at an ES5 code snippet so that we understand the nature of the problem that JS developers have to face today. For the purposes of this example we are going to make the assumption that there's a very simple HTML page with a button on it:

```
<body>
   <button id="greet" style="font-size: 30px">Greet</button>
   <script src="/es5/browser.es5.js"></script>
</body>
```

And that we have the following JavaScript code in place:

```
'use strict';

var btn = document.getElementById('greet');
```

```
var person = {
  name: 'Joe',
  age: 21,

  greet: function greet() {
    btn.addEventListener('click', function () {
      return this.display('Well hello there ' + this.name);
    });
  },

  display: function display(message) {
    console.log(message + '. You are ' + this.age + ' years old.');
  }
};

person.greet();
```

We create a person object that has some properties as well as some functions defined. The `greet` function adds an event listener to the greet button on our HTML page and it attempts to call another method defined inside the same object (`display`).

If you run this example and you click on the button you'll notice an error message in your browser's console `Uncaught TypeError: this.display is not a function`.

Doing some additional investigation and logging the value of `this` inside the event listener will reveal some interesting facts. `this` will refer to the button itself not to the person object as it is bound to the `btn` variable now.

There's a way around this which is to add a binding by calling the `bind()` method and making the anyonymous function an Immediately Invoked Functional Expression (IIFE):

```
greet: function greet() {
  btn.addEventListener('click', (function () {
    return this.display('Well hello there ' + this.name);
  }).bind(this));
},
```

Now our code works just as expected. Luckily we do not need to worry about such 'hacks' in ES6 no more!

If we use the arrow syntax to define our functions we can safely assume that the value of this is not going to change:

```
var btn = document.getElementById('greet');

var person = {
  name: 'Joe',
  age: 21,

  greet: function() {
    btn.addEventListener('click', () => this.display('Well hello there ' + this.name));
  },

  display: function(message) {
    console.log(message + '. You are ' + this.age + ' years old.');
```

```
  }
};

person.greet();
```

FUNCTIONS IN ES6

On top of the previously discussed arrow function there are other features in ES6 that are related to functions. We will now have a look at these.

REST PARAMETERS

The rest parameter is a special named function parameter that aims to overcome the challenge that JavaScript developers faced with the `arguments` object. Imagine a scenario where you'd like to create a function that adds numbers together that were passed in as parameters. In ES5 this would look like this:

```
function sum(numbers) {
  var result = numbers;

  for (var i = 1, length = arguments.length; i < length; i++) {
    result += arguments[i];
  }

  return result;
}

console.log(sum(1,2));
console.log(sum(1,5,9));
```

> Note that the arguments object is an Array-like object that contains the arguments that were passed to a function. However the arguments object does not have all the Array capabilities - it for example only has the `.length` property. If you want to use array manipulation methods such as `.pop()` you need to convert arguments to be an array: `var args = Array.prototype.slice.call(arguments);` (or: `var args = Array.slice(arguments);`)

The very same function can be rewritten in a much better way using ES6 and the rest parameter - notice that we are no longer useing the `arguments` object but we can access the `numbers` rest parameter directly and it acts like an array so we can directly call the `.length` property:

```
function sum(...numbers) {
  var result = null;

  for (var i = 0, length = numbers.length; i < length; i++) {
    result += numbers[i];
  }

  return result;
}

console.log(sum(1));
```

rest parameters are defined using the `...` followed by a name of the argument, in this case 'numbers'.

You can of course mix normal parameters for functions with rest parameters for example: `function sum(op, ...numbers)`.

SPREAD OPERATOR

The spread operator compliments the rest operator. It is commont to use the `.apply()` method in JavaScript in cases where we want to use an array as arguments to a function. Think about the following example (written in ES5) where we'd like to determine the largest number of the array using `Math.max()`

```
var numbers = [5, 10, 3, 15];
console.log(Math.max.apply(Math, numbers));
```

With the ES6 spread operator we can very simply pass in the variable name prefixed with the new familiar `...`:

```
var numbers = [5, 10, 3, 15];
console.log(Math.max(...numbers));
```

What is really cool that we can also add an extra parameter and that would be taken into consideration as well as part of the operation:

```
var numbers = [5, 10, 3, 15];
console.log(Math.max(...numbers, 22)); //max will be 22
```

DEFAULT PARAMETERS

A default parameter is a function parameter that has a default value provided to it. Unfortunately as of ES5 JavaScript has no such features but this is about to change with the introduction of default parameters in ES6.

There are of course workarounds in ES5. Consider the following example:

```
function greet(name, lang) {
  lang = lang || 'en'
  name = name || 'Unknown';

  var msg = null;
  if (lang === 'en') {
    var msg = 'Hello ';
  } else if (lang === 'it') {
    msg = 'Ciao ';
  } else {
    msg = 'Hello ';
  }
```

```
  return msg + name;
}

console.log(greet());
```

We can programatically setup default parameters using the || operator.

With ES6 we can now specify the default parameters:

```
function greet(name = 'Unknown', lang = 'en') {
  var msg = null;
  if (lang === 'en') {
    var msg = 'Hello ';
  } else if (lang === 'it') {
    msg = 'Ciao ';
  } else {
    msg = 'Hello ';
  }

  return msg + name;
}

console.log(greet());
```

Remember what we have learnt before? Arrow functions! So let's use them:

```
var greet = (name='Unknown', lang='en') => {
  var msg = null;
  if (lang === 'en') {
    var msg = 'Hello ';
  } else if (lang === 'it') {
    msg = 'Ciao ';
  } else {
    msg = 'Hello ';
  }

  return msg + name;
};

console.log(greet('Tamas', 'it'));
```

BLOCK LEVEL FUNCTIONS

In ES5, because of hoisting, functions can be defined inside an if block but as you can see from the code sample below they can be called anywhere inside the application code:

```
"use strict";

if (true) {
  function greet() {
    return 'hello there';
```

```
  }

  console.log(greet());
}

console.log(greet());
```

On the contrary to ES6 the function gets hoisted to the top of the block where it's defined, therefore as displayed by the following code sample the greet function can only be called inside the same block where it's defined:

```
"use strict";

if (true) {
  // function greet() {
  //   return 'hello there';
  // }
  // or
  var greet = () => 'hello there';

  console.log(greet());
}

//console.log(greet()); //undefined
```

OBJECTS

ES6 comes with two interesting updates for JavaScript objects.

COMPUTED PROPERTY NAMES

It is very easy to add computed property names to Objects using ES6:

```
var prefix = 'my';
var person = {
  [prefix + 'Name']: 'Joe',
  [prefix + 'Age']: 21
};

console.log(person.myName);
```

You can of course have 'standard' property names along with computed ones:

```
var prop = 'age';
var person = {
    'name': 'Joe',
    [prop]: 18,
    'birth place': 'Oxford'
};
```

```
console.log(person);
```

DESTRUCTURING OBJECTS

Let us assume that you have a person object which looks similar to this one:

```
var person = {
  name: 'Joe',
  age: 21,
  married: true
};
```

If you've worked with JavaScript probably you know the laborous exercise to get the properties for the object:

```
var myName = person.name;
var myAge = person.age;
console.log(myName + ' is ' + myAge);
```

This works of course but it's a tad cumbersome. But we have ES6 and the object destructuring. It is as simple as defining the properties of the object that you'd like to extract using the following notation:

```
var {name, age} = person;
console.log(name + ' is ' + age);
```

If you'd like your variable names to have a different name than the actual property you can update your code to read something like this:

```
var { name: myName, age: myAge } = person;
console.log(myName + ' is ' + myAge);
```

Object destructuring not only works with Objects but with Arrays as well:

```
var numbers = [1, 2, 3];
var [first, second] = numbers;
console.log(first + ', ' + second);
```

And of course you can mix Object and Array destructuring as well:

```
var person = {
  name: 'Joe',
  age: 21,
  favourite: [1, 2, 3]
};
var { name, age, favourite: [first] } = person;
console.log(name + ' is ' + age + ' and likes ' + first);
```

> Notice how we have only extracted the first value of the favourite array in our destructuring statement

---

TEMPLATE STRINGS

Templates and templating in JavaScript has always been a bit difficult, there was no easy way to add template strings that spanned across multiple lines and the `\n` ('newline') sequence symbol had to be used.

With ES6 things change quote a bit. Template strings can now be added using the ` (backtick) symbols. Anything that you put in between these backticks will remain in its original formatting.

Let's have a look at few examples:

```
var multiline = `this
is
a
multiline string`;

console.log(multiline);

var multiline = `this
is
a
multiline string
  with    tabbing`;

console.log(multiline);
```

There's one thing you need to be aware of. The following sentence (without extra tabs or linebreaks) is 39 characters long. You can easily verify this: `console.log("this is a multiline string with tabbing".length)`

However, if you do the same for the multiline variable you will get a completly different number.

> Try this out now `console.log(multiline.length)` should return 44.

You can do all sorts of other things with template strings, such as do variable substitution:

```
var name = 'Joe';
var msg = `Hello ${name}`;

console.log(msg);
```

And also you can do basic expressions inside the template strings such as this example below:

```
var value = 12;

var new msg = `Hello ${name}. You are worth ${value * 2}`;
console.log(new msg);
```

CLASSES

Since the existence of JavaScript classes have not been part of the language specification. All other object-oriented languages have support for classes, inheritance and all other OO features.

In ES5 you had to create a constructor and assign functions to its prototype. In ES6 this no longer is the case. You can now very simply define a class using the following syntax:

```
class Car {
  engine() {
    console.log('This car has an engine');
  }

  wheel() {
    console.log('Ideally four of them');
  }
}

var myCar = new Car();
myCar.engine();
myCar.wheel();
```

You can also create a constructor - just as if you'd do it using a standard OO language:

```
class Car {
  constructor(colour) {
    this.colour = colour;
  }

  getColour() {
    console.log('The car is ' + this.colour);
  }

  engine() {
    console.log('This car has an engine.');
  }

  wheel() {
    console.log('Ideally four of them.');
  }
}

var myCar = new Car('red');
myCar.engine();
myCar.getColour();
```

We could have more than one item in our constructor, for example a colour and a make:

```
constructor(colour, make) {
    this.colour = colour;
    this.make = make;
}
```

Classes can of course extend other classes - let's have a look at an example here:

```
class Truck extends Car {
  constructor() {
    super('black');
  }

  getColour() {
    return super.getColour();
  }
}

var myCar = new Car('red');
myCar.engine();
myCar.getColour();

var truck = new Truck();
truck.getColour();
```

super

The `super` keyword is used to call functions on an object's parent. This means that calling `super.getColour()` in the previous example actually calls the `.getColour()` method from the Car class.

Because we are using `super()` in our constructor we can overwrite the original instansiation property of car - `truck.getColour()` is therefore going to return us 'black'.

MODULES

Modules in JavaScript are not a new concept since there are lots of module loading systems out there, and also loading modules in Node.js is something that most JavaScript developers are already familiar with.

ES6 also brings a module loading system into JavaScript now.

In the first example we are going to have a look at a main file `app.js` that imports all modules from another file called `modules.es6.js` (note that this file was created by the Babel compiler)

```
import * as myModule from './modules.es6.js';
console.log(myModule.greet('Joe'));
console.log(myModule.age);
```

```
export function greet(name) {
  return 'Hello ' + name;
}
export var age = 21;
```

When using `import *` you name your module and access the methods and variables that you've exported from the module file.

But you can specify only a list of functions/variables that you'd like to import. To do this the following syntax needs to be utilised:

```
import {greet, age} from './modules.es6.js';
console.log(age);
console.log(greet('Joe'));
```

Modules can also export data from another modules. Consider the following example:

'anotherModule.es6':

```
export var colour = 'red';
```

'module.es6':

```
export function greet(name) {
  return 'Hello ' + name;
}

export var age = 21;

//this overwrites anotherModule
export var colour = 'black';

export * from './anotherModule.es6.js';
```

'app.es6':

```
import * as myModule from './modules.es6.js';
console.log(myModule.colour);
```

This is actually an interesting setup. We have two modules and they both define and export a variable called `colour`. We then print out that value - is it going to return the value 'red' or 'black'? If you run this code, you'll see that it in fact returns 'black'. In our main application (`app.es6`) we are importing a module (`modules.es6`) which overwrites the colour variable declaration coming from `anotherModule.es6`.

GENERATORS AND ITERATORS

As part of the ES6 specification JavaScript receives iterators as well as generators. As per MDN's definition an Iterator is an object "when it knows how to access items from a collection one at a time, while keeping track of its current position within that sequence".

These iterator objects have a .next() method will returns the next item in the aforementioned sequence. The method returns an object that has two properties - done and value. The done property is set to 'true' once the iterator finished and it'll have the value 'false' until the iterator has valid values.

Generators are special functions in JavaScript that works as a factory for iterators and they make use a new

keyword in JavaScript called `yield`. Generator functions are denoted by an asterisk in the name of the function (`function *myGenerator()...`)

> The `yield` keyword in JavaScript is used to pause/resume a generator function.

FOR OF AND ITERABLE

An iterable is an object that has an iterator specified for it. Arrays, objects (as well as maps, sets and strings) are also iterables which means that we can use a new iterator loop - `for...of` to iterate through the elements of an iterable object:

```
for (var item of [1, 2, 3]) {
  console.log(item);
};
```

This is the most basic example where we can use `for...of` as well as an iterable.

YIELD AND GENERATORS

> It's very important to note that at the time of writing this document you need to run the `babel-node` compiler instead of the `babel` compiler that we have been using up until now. If you have a generator function specified compile your code using `babel-node app.es6` for example.

Let's have a look at a Generator function that makes use of the `yield` keyword:

```
function *items() {
  yield 1;
  yield 2;
  yield 3;
}

var iterator = items();

for (var item of iterator) {
 console.log(item);
}
```

> `yield` is used to pause and resume a generator function. Once paused on a `yield` expression the generator's code execution remains paused until the generator's `next()` method is called. Each time the generator's next() method is called, the generator resumes execution and runs until it reaches either a `yield` statement, an exception, the end of a generator function or a return statment

The example above creates a generator called `items()` which we then assign to a variable called `iterator` and we finally utilise the `for...of` loop to iterate through the iterator.

We could also retrieve the values from the `items()` generator by calling the `.next()` method on the iterator variable:

```
console.log(iterator.next()); //value: 1; done: false
console.log(iterator.next()); //value: 2; done: false
console.log(iterator.next()); //value: 3; done: true
console.log(iterator.next()); //value: undefined; done: true
```

Note that you can also pass in values to the next method.

```
function *items() {
  var first = yield 1;
  var second = yield first + 2;
  yield second + 3;
}

var iterator = items();
console.log(iterator.next()); // value: 1; done: false
console.log(iterator.next(3)); // value: 5; done: false
console.log(iterator.next(5)); // value: 8; done: false
console.log(iterator.next()); // value: undefined; done: true
```

Let's have a look at why we see these results. When the `.next()` is called for the first time there's no point passing a value to it as any arguments passed to it are lost. (Arguments passed to `.next()` become the value returned by yield).

On the second call we are passing in a value of 3. This number is assigned to the variable `first` inside our generator function and we'll pause the execution on the `yield first + 2` statement which will give us the value '5'.

On the third call we pass in the value 5 and our generator function is going to continue with the execution - the value of `second` is going to be '5' and the generator function is going to pause on `yield second + 3` which will give us the result of '8'.

The last `.next()` call is going to return an undefined value as our generator function has reached it's end.

PROMISE

Simple put promises help you to work with asynchronous code in a synchronous fashion. The Promise can have 4 states, out of which you'll probably hear 2 being mentioned the most:

- pending
- fulfilled
- rejected
- settled

The idea behind promises is to allow you to create a function which - instead of a final value - returns a promise of having a value at some point in the future.

> Based on the above explanation you'll see how Promise was called Futures in older ECMAScript specifications.

Based on the previous explanation a promise can either be fulfilled or rejected. There are methods that can

be called when either of these happen and the methods are `.then()` and `.catch()` accordingly.

Interestingly enough both `.then()` and `.catch()` return a promise therefore calling these can be chained.

Let's have a look at a basic example which uses native promises (there are a lot of libraries out there that implement promises such as 'q' and 'bluebird')

We start off by having a very basic HTML structure that defines an input box and a button.

```
<body>
<div>
  <p>IATA code: <input type="text" maxlength="3" size="4" id="iata"></p>
  <p><input type="submit" class="btn" value="Get Information" onclick="go()"></p>
</div>
</body>
```

Non-promise implementation

Let's add some JavaScript. The code here is defines a `go()` function which will be called when the button is pressed. First we take the IATA code from that was written in the input box and we lookup some information about that airport (such as the city and the country of the airport) and we make another call to a different webservice method, this time, checking the weather at the 'destination' city/country.

You can see from the code below that we are building up what is called a 'pyramid of doom' where we are using values from our first HTTP request to generate our second HTTP request:

```
'use strict';
var city    = '';
var country = '';
var weather = '';

function go() {

  var code = document.getElementById('iata').value;
  var request = new XMLHttpRequest();
  request.open('GET', 'http://airportapi-tpiros1.rhcloud.com/airport/IATA/' + code);
  request.onload = function() {
    if (request.status === 200) {
      // parse & build up return values
      var result = JSON.parse(request.response)[0];
      city = result.municipality;
      country = result.iso_country;
      // we are resuing the values to make our second web service call
      request.open('GET', 'http://api.openweathermap.org/data/2.5/weather?q=' + city + ',' + country);
      request.onload = function() {
        if (request.status === 200) {
          // again, parse the values
          var result = JSON.parse(request.response);
          var celsius = (result.main.temp - 273.15).toPrecision(2); //Kelvin to Celsius
          var description = result.weather[0].description;
          weather = celsius + '°C and ' + description;
          console.log('The weather in ' + city + ', ' + country + ' is ' + weather + '.');
        } else {
          console.log(request.statusText);
```

```
        }
      }
      // this sends the second API call
      request.send();
    } else {
      console.log(request.statusText);
    }
  }
  // this sends the first API call
  request.send();
  // when this line is reached, we don't have the values fulfilled ... so we need to move it up
  // console.log('The weather in ' + city + ', ' + country + ' is ' + weather + '.');
}
```

This works however it's a little bit ugly, cumbersome to maintain and adding a third or even a fourth HTTP call would make the code look uglier.

Promise implementation

We could rework the example code to make use of native promises (meaning that we are not using any of the promise libraries).

The code here is defines a `getPromise()` method which will be called two times. First we take the IATA code from that was written in the input box and we lookup some information about that airport (such as the city and the country of the airport) and we make another call to a different webservice using the same `getPromise()` method, this time, checking the weather at the 'destination' city/country.

Because `getPromise()` returns a promise we can compose them (chain them) using the `.then()` method:

```
'use strict';
var city    = '';
var country = '';

// this function returns a promise (resolves or rejects)
function getPromise(url) {
  var promise = new Promise(function(resolve, reject) {
    var request = new XMLHttpRequest();
    request.open('GET', url);
    request.onload = function() {
      if (request.status === 200) {
        resolve(request.response)
      } else {
        reject(new Error(request.statusText));
      }
    };

    request.onerror = function() {
      reject(new Error('Cannot get data'));
    };

    request.send();
  });

  return promise;
}
```

```
function go() {
  var code = document.getElementById('iata').value;
  var promise = new getPromise('http://airportapi-tpiros1.rhcloud.com/airport/IATA/' + code);

  promise
  .then(function(result) {
    result = JSON.parse(result)[0];
    city = result.municipality;
    country = result.iso_country;
    return getPromise('http://api.openweathermap.org/data/2.5/weather?q=' + city + ',' + country);
  })
  .then(function(result) {
    result = JSON.parse(result);
    var celsius = (result.main.temp - 273.15).toPrecision(2); //Kelvin to Celsius
    var description = result.weather[0].description;
    var weather = celsius + '°C and ' + description;
    console.log('The weather in ' + city + ', ' + country + ' is ' + weather + '.');
  });
}
```

Now we have a much better code that can be easily maintained and we can keep on composing the `.then()` calls.