Course lesson plan was successfully updated. ×

ES6 and NodeJS

Previous Lab Review — 00:15

Author: Shaun McKinnon

Quick Look at Javascript Syntax — 00:30

A re-introduction to JavaScript (JS tutorial)

Variables

Variables declared with **var** are scoped to the nearest function block.

```
// var
function varring () {
  if (true) {
    var youCanSeeMe = 'Bob';
  }

  console.log(youCanSeeMe);
}
varring();
```

Variables declared with **let** are scoped to the nearest block. This is very powerful, but can be a gotchya when you're attempting to use a variable defined in a block elsewhere.

```
// let
function letting () {
  if (true) {
    let youCanSeeMe = 'Bob';
  }

  console.log(youCanSeeMe);
}
letting();
```

Both variables can be defined globally outside a function block. However, it is important to note, that the variables defined with **let** will not be available as properties on the **window** object.

const variables are constant. This means they cannot be redeclared or reinitialized.
They are also block-scoped, like **let**.

```
function consting () {
  if (true) {
    const youCanSeeMe = 'Bob';
  }

  console.log(youCanSeeMe);
}
consting();
```

Arrays

Arrays are available in every language, but in Javascript they're not type restricted, and all arrays are dynamically sized.

```
// arrays
let dc = ['Batman', 'Wonder Woman', 'Superman', 'Flash', 'Cyborg', 'Green Lantern'];
let marvel = ['Captain America', 'Black Widow', 'Incredible Hulk', 'Hawkeye', 'Thor', 'Iron Man',
'Vision'];

// current length
console.log( dc.length );
console.log( marvel.length );

// adding new elements
dc.push( 'Darkseid' );
marvel.push( 'Thanos' );

// new length
console.log( dc.length );
console.log( marvel.length );
```

Objects

You may hear people say that "Everything is an object in Javascript". While not **everything** in Javascript is an object, almost everything is. Variables, functions, and arrays are all objects. This provides a lot of flexibility to how we work with our program.

```
let student = {
  id: 200230778,
  name: 'Shaun',
  age: 39
};
student.output();
```

In the above example we've created an object with some information in it. We aren't limited to just data though. We can add functions, arrays, prototypes, whatever we really want within the object. We'll look at this again further on.

Functions

Functions are essential to DRYing up code. In Javascript we can define functions directly or anonymously.

```
// direct function declaration
function createAStudent ( id, name, age ) {
  return {
    id: id,
    name: name,
    age: age
  }
}
console.log( createAStudent( 12345, 'Shaun', 39 ) );
```

```
let student = function ( id, name, age ) {
  return {
    id: id,
    name: name,
    age: age
  }
}
console.log( student( 12345, 'Shaun', 39 ) );
```

Control Statements

We have all experienced control statements by this point. Basically they control the flow of our application. These can be conditional statements, functional statements, and iterative statements.

Decision/Conditional statements are how we control the direction we want a program to go. They allow us to evaluate an expression and determine based on a boolean result what to do next:

```
// standard if/else condition
let flag = true;
if ( flag === true ) {
  console.log( 'do this...' );
} else {
  console.log( 'do this instead...' );
}

// single line
if ( flag === true ) console.log( 'do this...' );

// implied unless false
if ( flag ) console.log( 'do this...' );

// ternary
console.log( ( flag ) ? 'do this...' : 'do this instead...' );

// logical operators
let doThis = null;
let doThisInstead = 'ohhh yeah';
console.log( doThis || doThisInstead );
doThis = true;
console.log( doThis && doThisInstead );

// switch statements
let name = 'Bob';
switch ( name ) {
  case 'Bob':
    console.log( 'Hi Bob' );
    break;
  default:
    console.log( 'Hi' );
    break;
}

// switch statements made cool
switch( true ) {
  case (5 > 3):
    console.log( 'Yes it is' );
  case (10 > 7):
  case (6 < 10):
    console.log( 'Hells yeah' );
  default:
    console.log( 'Guess we need some break statements' );
}
```

Repetition structures in a programming are essential when we're trying to read files, process arrays, listen for events, basically anything that involves the need to repeat ourselves. Javascript gives us several ways to iterate:

```
// Repetition Control Statements
// the for loop
for ( let i = 0; i < 10; i++ ) {
  console.log( 'Yay for ' + i );
}

// short hand, but you are responsible for
// handling the terminating condition
let i = 0;
let tCon = true;
for ( ; tCon; ) {
  console.log( 'Yay for ' + i );
  if ( i == 10 ) {
    tCon = false;
  }

  i++
}

// the while loop
let count = 0;
while ( count < 10 ) {
  console.log( 'Yay for ' + count );
  count++;
}

// the do while loop
let doCount = 0;
do {
  console.log( 'Yay for ' + count );
  count++;
} while ( count < 10 );

// iterating through arrays
let days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'];
// the basic way
for ( i = 0; i < days.length; i++ ) {
  console.log( days[i] );
}

// the cleaner way
for ( let day of days ) {
  console.log( day );
}

// iterating through objects
let months = {
  Jan: 'January',
  Feb: 'February',
  Mar: 'March',
  Apr: 'April',
  May: 'May'
}

for ( let key in months ) {
  // we should be sure the property is of this object
  if ( !months.hasOwnProperty( key ) ) continue;

  console.log( key, months[key] );
}
```

Author: Shaun McKinnon

Blocking VS Non-Blocking Code — 00:15

Callbacks

Synchronous Callbacks

We have seen in some examples of setting up the Node server, that we have the ability to pass functions as arguments to other functions. These are generally referred to as callbacks (but you may also see them called higher order functions).

The idea of a callback is for it to be executed at a specific time. The convenience of a callback is it has current functional scope and state. We generally execute callbacks once our application has identified that an event has occurred and it now wants to do something in response.

It is important to understand that while callbacks are used a lot in asynchronous programming, they are not exclusive to it. There are many reasons that you may want to maintain context and state as you execute your application, and callbacks allow for that.

Let's look at an extremely basic example of a callback function:

```
function myTask ( my_callback_func ) {
  console.log( "I have completed whatever it was that I wanted to do." );

  // I am now executing my callback function
  my_callback_func();
}

myTask( function () {
  console.log( "My turn to work!" );
});
```

As you can see, our function logic executed the callback by simply calling the symbol as a function. If, for any reason, we hadn't passed a function, this would result in an error:

```
▶Uncaught TypeError: my_callback_func  VM9327:5
is not a function
    at myTask (<anonymous>:5:3)
    at <anonymous>:1:1
```

This callback was synchronous. This means that our code executed in a blocking manner, not allowing other things to occur while it was processing. This is not the general norm for using a callback. Usually we use callbacks when an asynchronous action has occurred and we are ready to work with its response.

Asynchronocity

Before we get into **Asynchronous Callbacks** it is important that we understand the difference between synchronous and asynchronous code. In order to do this, we can use a handy Javascript function known as **setTimeout()** that allows us to delay when a piece of code will fire:

```
function output( message ) {
  console.log( message );
}

// setTimeout( function name, delay in ms, parameters );
setTimeout( output, 5000, 'I want to go first!' );
output( 'No, I want to go first!' );
```

"No, I want to go first!" executed before the "I want to go first!" statement due to that Javascript is **asynchronous**. This means it won't wait. It won't **STOP THE WORLD** as our friend from the video stated.

This is hands down the concept that confuses most new Javascript programmers. Most languages (unless you learn threading or forking) are synchronous. This means calls that take time, do exactly that: **they take time**. They stop our application from continuing and we have to wait before it will continue on. This is the standard way most programmers learn to program. However, once you begin working with Javascript, you start to realize the frustration of working with synchronous code, and you begin to appreciate the asynchronous nature of Javascript.

Callback Hell

Callback Hell is the result of needing several things to occur synchronously that want to occur asynchronously. The result is a slew of nested callbacks:

```
// callback hell
var https = require ( 'https' );

let getInsult = function ( callback ) {
  let body = '';

  // 1
  let req = https.get( 'https://insult.mattbas.org/api/insult',  function ( res ) {
    // 2
    res.on( 'data', function ( chunk ) {
      body += chunk;
      // 3
      callback( body );
    });
  });
}

getInsult( function ( body ) {
  console.log( body );
});
```

3 callbacks to achieve what we want. And it's very difficult to follow what's happening for people reading your code. Let's walk through this:

1. The first callback is the one that is fired when https.get has completed its request to the url we passed. It calls our passed function passing in the response object as an argument
2. The second one is called once we receive the 'data' packet. We call function and pass the data chunk as an argument to the callback function
3. The third callback is made so we can pass our parsed data to the callback. This allows the developer who's calling the function to trigger a response once they have the body

Callback hell is the reason **promises** now exist. It is cleaner and much easier to read when writing code.

Author: Shaun McKinnon

BREAK — 00:15

Introducing ES6 syntax — 00:45

TUTORIAL: Introducing the ES6 syntax

ES6 Cheat Sheet

What is ES6?

ES6 stands for ECMA Script 6. ECMA is an organization that standardizes information. Because Javascript is implemented differently in various browsers, there has to be a standard and specification that they must follow to ensure the developers have a common environment to work with. However, updating browsers to support every new standard can be costly, and therefore browsers will adopt when they choose. They also aren't required to update old browsers with the new standard, so it is very possible that new code will not work properly in older browsers.

Transpilers to the Rescue!

Transpilers allows us to use newer code functionality, regardless of support for that functionality, by transpiling the new code into legacy compatible code. This means that older browsers, and browsers that are taking their time, will still work.

However, we now have an extra step in our process. We must transpile in order to ensure we are cross browser compliant. That isn't a big deal considering we can add transpilers to our build process, which will result in us always having cross-browser compliant code, regardless of what we code.

Handy ES6 Stuff

We already showed how we can use **let** and **const** instead of **var**. However, ES6 introduces many other really cool features that make writing Javascript simple:

```
// no longer need IIFEs (Immediately Invoked Function Expressions) thanks to block scope :)
(function () {
  var myScopedVar = "Bob";
}());

// now becomes
{
  let myScopedVar = "Bob";
}
```

IIFEs are commonly used to maintain block scope. This aids in avoiding namespace collisions.

Arrow Functions are very handy, and not just because they're shorter to write than full functions. They maintain also maintain the scope of the keyword **this**. Many languages use the keyword **this** to represent the current scoped object. It gives you access to defined properties and methods and is fundamental to object oriented programming. However, in Javascript **this** changes scope to reference its current function container. That means if you call **this** from within a callback it is referring to its callback and not the surrounding function.

Many programmers have overcome this issue by assigning **this** to **that** or **self** or some other arbitrary variable that will maintain its scope for them. This also isn't ideal as you want **this** to be contextual. **Arrow Functions** do not have a sense of **this** in their own context. They pass **this** through with containing context. We definitely need an example:

```
function Person() {
  var that = this;
  that.age = 0;

  setInterval( function growUp() {
    // The callback refers to the `that` variable of which
    // the value is the expected object.
    that.age++;
    console.log( that.age );
  }, 1000 );
}

let peep = new Person();
```

The above may seem odd but is essentially a class with a constructor. The constructor is setting the property of **that** equal to **this** so it will maintain context when given to the callback function **growUp**, otherwise **growUp** will throw a reference error when we attempt to access **this.age** as it isn't present within its context.

Below we attempt the same example, but instead use the **arrow function**:

```
function Person(){
  this.age = 0;

  setInterval(() => {
    this.age++; // |this| properly refers to the person object
    console.log( this.age );
  }, 1000);
}

let peep = new Person();
```

In the above example, we are passing an **arrow function** as the callback. The **arrow function** maintains the context of **this** throughout the body of the function. Much cleaner. Much easier to read.

Semicolons are a fundamental pa...no, no they are not. **Semicolons** are no longer a requirement in ES6.

```
console.log( 'Semicolons are for ' )
console.log( 'things I cannot say unless I want to get fired' )
```

Map is very cool method of array. It allows us to walk through the array and perform operations on each element:

```
['me', 'you', 'them'].map(n => console.log(n.length))
```

The above is an inline expression. Below is a full operation:

```
['me', 'you', 'them'].map(n => {
  console.log( n )
  console.log( n.length )
})
```

Template Literals are in so many languages. Some even do it right (I'm lookin' at you Ruby). Template Literals allow you to embed expressions in strings. These expressions will be parsed within the string, making it very easy to embed variables, calculations, function calls, method calls, properties, etc...

```
let name = "Shaun McKinnon"
let age = 39

let text = `${name} is ${age - 20} years old.`
console.log( text );
```

Destructuring is one my most favourite constructs available in ES6. It makes working with arguments more readable. The goal is to get to a point where commenting isn't necessary because the syntax is understandable. **Destructuring** allows us to extract values and then immediately store them in variables that have a more meaningful syntax:

```
let [day, month, year] = [21, 5, 2018]
console.log(day, month, year)
```

The above **destructures** the values in the array. We can now access those values with the more contextual **day**, **month**, and **year** variables. We can literally call the variables whatever we want (following variable naming rules).

```
let Student = {
  name: 'Shaun',
  id: 456321,
  age: 39
}

function myFunc ( {id, name, age} = options ) {
  console.log( name, id, age )
}

myFunc( Student )
```

In the above code, we **destructure** the options and pull out the name, id, and age from the passed object. Keep in mind, we MUST pass in an object that has those values, or we will wind up with variables being undefined.

Also notice that the order of the destructuring isn't the same as the order of the keys in the object. This is important because when destructuring an object is uses the keys to match up the variables. That means we don't have the flexibility in naming the variables like we did in the array. They MUST be the same as the key names.

[You can read more about desctructuring. It is a very power feature that helps make code a lot more human readable.]

Classes

Classes is so important it gets its own heading. Prior to ES6, we had to this fun:

```
// This is basically our class and constructor
function Student ( id, name, age ) {
  // these are the properties
  this.id = id;
  this.name = name;
  this.age = age;
}

// This is a method of the class
Student.prototype.output = function () {
  console.log( this.id, this.name, this.age );
};

// here's instantiation of the class
( new Student( 1234, 'Shaun', 39 ) ).output();
```

Clear as the windows in a sketchy bar, right? This is prototyping. A very different approach to building 'classes' in Javascript. If you're looking for a more traditional version of classes, look no further than our friend, ES6:

```
class Student {
  constructor ( id, name, age ) {
    this.id = id
    this.name = name
    this.age = age
  }

  output () {
    console.log( this.id, this.name, this.age )
  }
}

( new Student( 1234, 'Shaun', 39 ) ).output()
```

Annnnnnd, if you want inheritance, you can use the keyword **extends**:

```
class StudentGroup extends Student {}
```

ES6 and ES7 and ES8 are making huge changes to how we code Javascript. These massive changes are going to make a more approachable language for the layman and will definitely increase its popularity.

Author: Shaun McKinnon

BREAK — 00:15

Promises — 00:20

https://medium.com/dev-bits/writing-neat-asynchronous-node-js-code-with-promises-32ed3a4fd098

Let's change our original insult code so that it uses promises:

```
// Promises
const request = require ( 'request' )

function getInsult ( url ) {
  let options = {
    url: 'https://insult.mattbas.org/api/insult',
    headers: { 'User-Agent': 'request' }
  }

  // 1
  return new Promise( ( resolve, reject ) => {
    // 2
    request.get( options, ( err, resp, body ) => {
      if ( err ) {
        // 3
        reject( err )
      } else {
        // 4
        resolve( body )
      }
    })
  })
}



getInsult( url ).then( ( result ) => {
  console.log( result )
})
```

So now you're probably saying "Great!" so we still have 3 nested levels of callbacks. What exactly did this solve? Well, one, we converted all our functions to arrow functions. Whoop. We now have responses based on if our promise is rejected or resolved. Ok...Promising (see what I did there?). We also now have the ability to chain events.

```
getInsult( url )
.then( ( result ) => {
  console.log( result )
})
.then(() => console.log('This is next'))
.then(() => console.log('And then this one'))
```

And we can sequence promises:

```
let insultOne = getInsult()
let insultTwo = getInsult()
let insultThree = getInsult()

Promise.all( [insultOne, insultTwo, insultThree] ).then( ( result ) => console.log( result ) )
Promise.all( [insultTwo, insultOne, insultThree] ).then( ( result ) => console.log( result ) )
Promise.all( [insultThree, insultTwo, insultOne] ).then( ( result ) => console.log( result ) )
```

The **Promise.all()** function takes a list of promises, in any given order, and returns a new promise once they are completed.

It is important to understand that resolved or rejected promise is immutable. The value will never change.

Author: Shaun McKinnon

Node: A little deeper — 00:20

The File Server

Node allows us to read files:

```
const fs = require( 'fs' )

fs.readFile( 'index.html', ( err, data ) => {
  if ( !err ) console.log( data )
})
```

The above will read the file, and log the data if present. It is important to note, that if no encoding is defined (the **utf-8** arg), it will return a raw buffer.

Using that information, we can read a file and send its contents to our client in our response instead of just raw text. We'll have to make some changes though (and we might as well use Promises):

```
const connect = require( 'connect' );
const app = connect();
const fs = require( 'fs' )

function getFile ( path ) {
  return new Promise( ( resolve, reject ) => {
    fs.readFile( path, ( err, data ) => {
      resolve( data )
    })
  })
}

let home = ( req, res ) => {
  getFile( 'index.html' )
  .then( ( data ) => {
    res.writeHead( 200, {'Content-Type': 'text/html'})
    res.write( data )
    res.end();
  })
};

app.use( '/', home );

app.listen( 3001 );

console.log( 'Server running at http://localhost:3001' );
```

There you have it: a way to send a file to the browser. We have some gotchyas though; if we link to a CSS file, it won't be handled correctly. That is because we are only writing a header for text/html and not really caring about any other requests that may come through. We should fix this:

```
const connect = require( 'connect' );
const app = connect();
const fs = require( 'fs' )
const url = require( 'url' )

function getFile ( path ) {
  return new Promise( ( resolve, reject ) => {
    fs.readFile( path, 'utf-8', ( err, data ) => {
      resolve( data )
    })
  })
}

const mime = {
  html: 'text/html',
  css: 'text/css',
  js: 'application/javascript',
  jpg: 'image/jpeg',
  png: 'image/png'
}

let home = ( req, res ) => {
  let path = (url.parse(req.url)).pathname
  path = '.' + ( ( path != '/' ) ? path : '/index.html' )
  let ext = path.split('.').pop()
  let type = mime[ext] || 'text/plain'

  getFile( path )
  .then( ( data ) => {
    res.writeHead( 200, {'Content-Type': type})
    res.write( data )
    res.end();
  })
};

app.use( '/', home );

app.listen( 3001 );

console.log( 'Server running at http://localhost:3001' );
```

Here we're adding a lookup object so we can find the mime type. The line **let ext = path.split('.').pop()** is splitting **path** at the **.** period into an array and then popping off the last element. The last element (always hopefully) should be the file extension. Then we give the extension as a key and grab the mime type, or just return 'text/plain'.

Our function hasn't changed, but we've abstracted the arguments we're providing. This will allow us to attach images, html, css, and javascript to our HTML page if we choose to.

Author: Shaun McKinnon