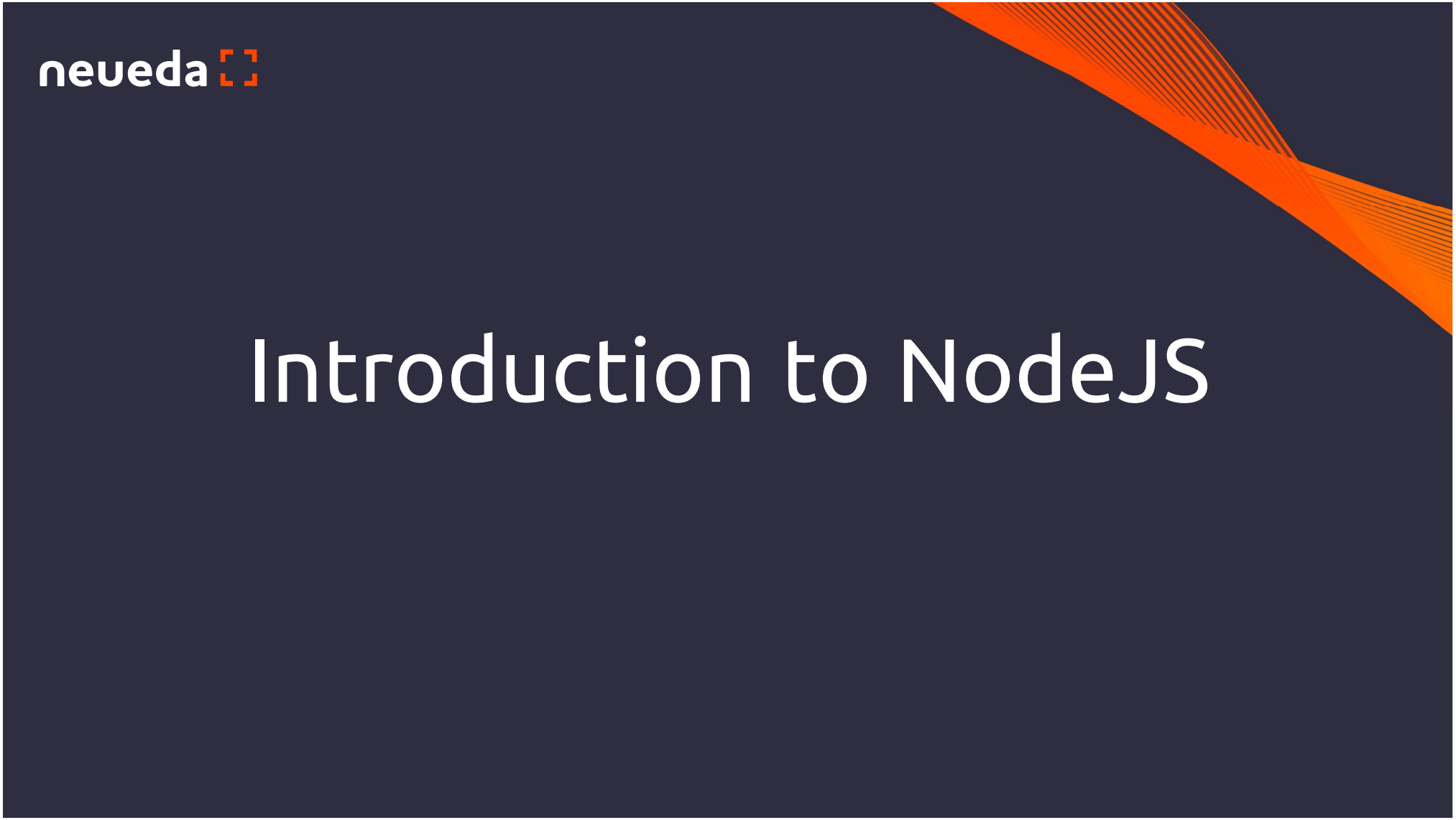
Neueda Technologies

Working with JavaScript

Contents


neueda
Introduction to NodeJS

Objectives

- History of NodeJS
- NodeJS Core Features
- Installing NodeJS
- NodeJS Package Management

neueda

History of NodeJS

- Up until 2009 JavaScript was largely a technology used in the client side elements of Web applications
- In 2009, NodeJS was released that provided a framework that allowed server side applications to also be developed using JavaScript
- This meant that for the first time, an entire application could be built using JavaScript
  - Both the back end and the front end

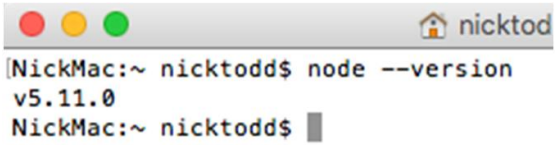
NodeJS Core Features

- Node JS uses the Google V8 JavaScript Engine
- The architecture is event driven
- The I/O can be asynchronous
- The event driven and asynchronous nature of the framework is ideal for building scalable and performant Web applications
- The Framework allows you to create additional libraries and then these can be shared and managed using the Node Package Manager (npm)

Installing NodeJS

- NodeJS can be installed on Windows / Mac / Linux
- Install files can be downloaded from
  - https://nodejs.org/en/download/stable/

- Once installed, node will be available from the command line or console



neueda

Hello World in Node

- Since NodeJS is a runtime for JavaScript, a NodeJS application is a JavaScript application

```
console.log("hello world from Node!");
```

```
E:\Dropbox\Conygre\UIUX\Demos\ecma6_general>node helloworld.js
hello world from Node!

E:\Dropbox\Conygre\UIUX\Demos\ecma6_general>
```

Referencing Packages

- When a script requires a package
  1. Install it with the Node Package Manager
  2. Reference it in the code using either **require** or **import**

- Require is the ECMA 5 way to do it

```
const yourNameForThePackage = require("package_name")
```

- Using import is the ECMA 6 way to do it and also allows you to import specific components from the module

```
import {store} from  './actions';
```


neueda

Basic Web Server Example

- Below is another example
  - This time a simple Web server

```
var http = require("http"); //load the package
http.createServer(function (request, response) {
   response.writeHead(200, {'Content-Type': 'text/plain'});
   response.end('Hello World\n');
}).listen(8081);
```

- The example above creates a function that will be executed when HTTP requests come in on port 8081
- To launch the example, type
  - `node basicserver.js` at a console

Installing Node Packages

- Packages are installed using **Node Package Manager** (npm)

```
$ npm install express
```

- Package along with any dependencies are placed into a folder called **node_modules** within your project
- Each module has a JSON configuration file called **package.json**

package.json

- Each library will have dependencies, contributors and license agreements
  - This is all specified in **package.json** which is a config file that each library will have
- Much of it is self explanatory
  - One particularly important section is where the dependencies that the package has
  - These then also get downloaded upon install

demos/nodejs/node_modules/express/package.json

neueda

Dependencies

- Below is a snippet from a NodeJS package dependencies section

```
 "dependencies": {
    "accepts": "~1.2.12",
    "array-flatten": "1.1.1",
    "content-disposition": "0.5.1",
    "content-type": "~1.0.1",
    "cookie": "0.1.5",
    "cookie-signature": "1.0.6",
    "debug": "~2.2.0",
}
```

- Version numbers can have symbols next to them such as
  - ^ Compatible with
  - ~ Approximately equivalent to

Summary

- History of NodeJS
- NodeJS Core Features
- Installing NodeJS
- NodeJS Package Management

neueda


neueda
JavaScript Essentials

Contents

- Working with variables
- Using operators
- Performing tests
- Performing loops
- Defining functions

neueda

Working with Variables

- Overview of JavaScript variables
- Declaring variables
- JavaScript primitive data types
- Assigning to undeclared variables
- Doing arithmetic

Overview of JavaScript Variables

- As with algebra (x, y, z), JavaScript variables are used to hold values or expressions
  - A variable can have a short name, like x, or a more descriptive name, like `userName`

- Rules for JavaScript variable names:
  - Variable names are case sensitive (y and Y are two different variables)
  - Variable names must begin with a letter or the underscore character

Declaring Variables

- To create (or "declare") variables in JavaScript, use the `let` keyword
  - You can declare variables with no initial value (empty initially)

```
let x;
```

  - Or you can assign values to the variables when you declare them:

```
let x = 5;
```

- We will see more on this later using **let** and **var** and we will discuss scoping rules

JavaScript Primitive Data Types

- JavaScript defines a small set of primitive types:
  - string, number, boolean

```
let userName   = "Andy";
let monthName  = "December";

let dayOfMonth = 3;
let height     = 1.67;
let favColor   = 0xFF0000;

let isWelsh    = true;
let canSing    = false;
```

- JavaScript also supports the concept of objects
  - See later in course for details

Assigning to Undeclared Variables

- If you assign values to variables that have not yet been declared…
  - The variables will automatically be declared

```
let x = 5;
let userName = "Andy";
```

- The following groups of statements have the same effect:

```
x = 5;
userName = "Andy";
```

Doing Arithmetic

- You can do arithmetic using variables and constants
  - JavaScript supports all the operators you might expect
  - E.g. + - * /

```
let radius = 10;
let area = 3.14 * radius * radius;
console.log("Area of circle is " + area);
```


neueda

2. Using Operators

- Arithmetic operators
- Assignment operators
- String handling
- Comparison operators
- Logical operators
- The conditional operator

neueda

Arithmetic Operators

- Arithmetic operators are used to perform arithmetic between variables and/or values
- Imagine we've already set y= 5, the following table explains the arithmetic operators

Operator	Description	Example	x value
+	Addition	x = y + 2;	7
-	Subtraction	x = y – 2;	3
*	Multiplication	x = y * 2;	10
/	Division	x = y / 2;	2.5
%	Modulus (remainder)	x = y % 2;	1
++	Increment by 1	x = ++y;	6
--	Decrement by 1	x = --y;	4

Assignment Operators

- Assignment operators are used to assign values to JavaScript variables
- Imagine we've already set x=10 and y=5, the following table explains the assignment operators

Operator	Description	Example	Same as…	x value
=	Assignment	x = y;		5
+=	Add and assign	x += y;	x = x + y;	15
-=	Subtract and assign	x -= y;	x = x - y;	5
*=	Multiply and assign	x *= y;	x = x * y;	50
/=	Divide and assign	x /= y;	x = x / y;	2
%=	Modulus and assign	x %= y;	x = x % y;	0

11

neueda

String Handling

- Use + to add strings (variables and/or string variables)

```
txt1 = "Hello";
txt2 = "world";
txt3 = txt1 + " " + txt2 + "!!!";
```

- Use += to concatenate text to a string variable

```
let output = "<h1>";
output += "This is message ";
output += "to the user";
output += "</h1>";
```

String Concatenation

- If you mix-and-match numbers in a string concatenation, everything gets stringified

```
let swansGoals = 0;
swansGoals++;
swansGoals++;
swansGoals++;
let output = "Swansea scored " + swansGoals + " against Cardiff :-)";
```

String Concatenation using Template Literals

- You can also concatenate strings that include JavaScript variables using templating
- The syntax uses
  - Back ticks (`)instead of quotes around the text
  - To place in a variable, use the **${**stickYourExpressionHere**}** syntax

```
var a = 3, b = 3.1415;
console.log(`PI is roughly ${a} or more roughly ${b}`);
```


neueda

Comparison Operators

- Comparison operators are used to compare two variables and/or values
- Imagine we've already set y=5, the following table explains the comparison operators

Operator	Description	Example	Result
>	Is greater than?	y > 8	false
>=	Is greater than or equal?	y >= 8	false
<	Is less than?	y < 8	true
<=	Is less than or equal?	y <= 8	true

```
if (userAge >= 18)
  alert("Adult");
```

Equality and Identity Operators

- Equality operators compare two variables and/or values to see if they have the same value
  - Will coerce (i.e. convert) values to same type if necessary

Operator	Description	Example	Result
==	Equality test	5 == "5"	true
!=	Non-equality test	5 != "5"	false

neueda

Strict Checking

- Identity operators compare two variables and/or values to see if they have the same value
- Will NOT coerce (i.e. convert) values to same type

Operator	Description	Examples	Result
===	Identity test	5 === "5" 5 === 5	false true
!==	Non-identity test	5 !== "5" 5 !== 5	true false

neueda

Logical Operators

- Logical operators are used to combine multiple test conditions
- Imagine we've already set x=10 and y=5, the following table explains the logical operators

Operator	Description	Example	Result
&&	Logical and	(x < 20 && y > 1)	true
||	Logical or	(x == 20 || y == 20)	false
!	Logical not	!(x == y)	true

```
if (userAge >= 18 && userAge <= 30)
  alert("You are eligible to go on an 18-30
holiday!");
```


neueda

The Conditional Operator

- The "conditional operator" is an if-test in a single expression
  - General syntax:

```
(test-expression) ? true-expression : false-expression
```

  - Example:

```
let favTeam;

… code to ask user for favourite team …

let msg = (favTeam == "Swansea") ? "Good choice" : "Bad choice";

console.log("Your team is " + favTeam + "[" + msg + "]");
```

Flow Control

- Overview of conditional statements
- if statements
- if…else statements
- if…else…if statements
- switch statements
- Aside: Handling errors

neueda

Overview of Conditional Statements

- Very often when you write code, you want to perform different actions for different decisions
  - You can use conditional statements in your code to do this

- JavaScript has the following conditional statements:
  - **if**
    - Use this statement to execute some code only if a specified condition is true
  - **if...else**
    - Use this statement to execute some code if the condition is true and another code if the condition is false
  - **if...else...if**
    - Use this statement to select one of many blocks of code to be executed
  - **switch**
    - Use this statement to select one of many blocks of code to be executed

if Statements

- Use `if` statements to execute some code only if a specified condition is true

```
if (condition)
{
  code to be executed if condition is true
}
```

- Example

```
let d = new Date();
let time = d.getHours();

if (time < 12)
{
  document.write("<b>Good  morning!</b>");
}
```


neueda

if…else Statements

- Use `if…else` statements to execute some code if a condition is true, otherwise execute other code

```
let d = new Date();
let time = d.getHours();

if (time < 12)
{
  document.write("<b>Good morning!</b>");
}
else
{
  document.write("<b>Good day!</b>");
}
```


neueda

if…else…if Statements

- Use `if…else…if` statements to select one of several blocks of code to be executed

```
…
if (time < 12)
   …
else if (time < 17)
   …
else
   …
```


neueda

switch Statements (1 of 2)

- Use a `switch` statement to select one of many blocks of code to be execute, based on the value of a variable

```
switch(n) {
case 1:
  execute code block 1
  break;
case 2:
  execute code block 2
  break;
default:
  code to be executed if n is different from case 1 and 2
}
```

switch Statements (2 of 2)

- You can have any number of case branches
- Each case value must be a constant value
- Use break to prevent "fall-through" to next branch
- The default branch is optional

```
let d = new Date();
theMonth = d.getMonth();
switch (theMonth) {
case 11:
case 0:
case 1:
  document.write("Winter");
  break;
case 2:
case 3:
case 4:
  document.write("Spring");
  break;
case 5:
case 6:
case 7:
  document.write("Summer");
  break;
default:
  document.write("Autumn");
  break;
}
```


neueda

Handling Errors

- Imagine you perform an operation that might fail
  - E.g. make a call to a server
  - E.g. open a database connection

neueda

Try / Catch

- JavaScript uses exceptions to indicate error conditions
  - You can enclose code-that-might-fail in a `try` block
  - Then define a `catch` block to catch the exception (and maybe display info about the error to the user)
  - Optionally define a `finally` block (executed unconditionally)

neueda

Try / Catch Example

```
try {
  // Some code that might cause an exception…
}
catch (ex) {
  console.log("Exception occurred: " + ex);
}
finally {
  console.log("Code here will always be executed - good place to do tidying
up!");
}
```

Performing Loops

- Overview of loop statements
- for statements
- while statements
- do…while statements
- for…in statements
- Jump statements

neueda

Overview of Loop Statements

- Often when you write code, you want the same block of code to run several times
  - You can use loop statements in your code to do this

- JavaScript has the following loop statements:
  - **for**
    - Loops through a block of code a specified number of times
  - **while**
    - Loops through a block of code while a specified condition is true
    - The test about "whether to continue" is at the start of the loop
  - **do...while**
    - Loops through a block of code while a specified condition is true
    - The test about "whether to continue" is at the end of the loop
  - **for...in**
    - Loops through a collection of items, to perform some task on each item

for Loops

- Use `for` loops when you know in advance how many times the script should run

```
for (initialization; test; update)
{
  code to be executed
}
```

- Example

```
let i = 0;
for (i = 0; i < 5; i++)
{
  console.log("The number is " + i);
}
```

while Loops

- Use `while` loops to loop through a block of code while a specified condition is true
  - Note: the test condition is at the *start* of the loop body

```
while (condition)
{
  code to be executed
}
```

```
let i = 0;
while (i < 5)
{
  document.write("The number is " + i + "<br/>");
  i++;
}
```


neueda

do...while Loops

- Use **do...while** loops to loop through a block of code while a specified condition is true **(loops at least once)**
  - Note: the test condition is at the end of the loop body

```
let i = 0;
do
{
  document.write("The number is " + i + "<br/>");
  i++;
}
while (i < 5);
```

for…in Loops

- Use `for…in` loops to loop through all the items in a collection (e.g. an array)
  - Perform some task on each item

```
for (variable in collection) {
  code to be executed
}
```

```
let players = new Array();
players[0] = "Leon Britton";
players[1] = "Nathan Dyer";
players[2] = "Joe Allen";

let c;
for (c in players) {
  document.write("Great Swans player: " + players[c] + "<br/>");
}
```


neueda

Jump Statements

- Use **break** to immediately terminate a loop:
- Use **continue** to abandon the current iteration

```
let i;
for (i = 1; i < 20; i++)
{
  if (i % 10 == 0)
    break;

  if (i % 3 == 0)
    continue;

  console.log("i is " + i + "<br/>");
}
console.log("The end");
```


neueda

Defining Functions

- Overview of functions
- Defining functions
- Simple example
- Returning a value
- Local vs. global variables
- Additional considerations

Overview of Functions

- A function contains code that will be executed by an event or by a call to the function
- You may call a function from anywhere within the code

neueda

Defining Functions

- To define a function:
  - Use the `function` keyword
  - Define parameters (i.e. inputs) in brackets, ()
  - Implement the function body in braces, {}

```
function functionName(parameter1, parameter2, ... , parameterN) {
  function body code
}
```


39

Function Parameters

- Function parameters are named, but not typed
  - The parameter types are determined at run time, when you pass argument values into the function

```
function functionName(parameter1, parameter2, ... , parameterN) {
  function body code
}
```


neueda

Invoking Functions

- To invoke (i.e. call) a function:
  - Use the name of the function
  - Pass argument values inside the brackets, ()

```
functionName(value1, value2, ... , valueN);
```

Returning a Value

- You can return a value from a function
  - Using the `return` keyword

```
function constructMessage(firstName, lastName) {
  return "Hello " + firstName + " " + lastName;
}

function displayMessage(firstName, lastName) {
  let msg = constructMessage(firstName, lastName);
  console.log(msg);
}

constructMessage(“Nick”, “Todd”);
displayMessage(“Nick”, ”Todd”);
```

Scoping

- Variables declared without use of **let** are always global regardless of where you declare them

```
function sayHello(name) {
    console.log(name)
    anotherGlobalVariable = name;
}

globalNameVariable = "Nick"
sayHello(globalNameVariable)
console.log(anotherGlobalVariable)
```

Using var

- Using the var keyword limits the scope to the function if declared in a function
  - The variables are local to the function
  - Only accessible in function
  - Created on declaration, destroyed on exit

```
function sayHello(name) {
    console.log(name)
    var localVariable = name;
}

globalNameVariable = "Nick"
sayHello(globalNameVariable)
console.log(localVariable) // FAILS!
```


neueda

Using let

- Consider the following

```
function sayHello(name) {
    for(var i=0; i<3; i++) {
        console.log(i)
    }
    console.log(i) // does this work? It does!
```

- Using var scopes to the function
- If we use **let**, it scopes to the curly brackets – much better!

```
function sayHello(name) {
    for(let i=0; i<3; i++) {
        console.log(i)
    }
    console.log(i) // does this work? Not any more
```


neueda

Using const

- **const** is an alternative to the let keyword – variables defined with **const** cannot be reassigned.

```
let name = "Nick";
name = "Dave”;

const age = 27;
age = 28; //this line won’t work!
```


neueda

Parameter Passing is Optional

- The number of arguments you pass into a function doesn’t have to match the number of parameters in the function
  - If you pass too few arguments, the value of any parameters you haven't supplied is undefined
  - If you pass too many arguments, the surplus arguments are just ignored
- This means JavaScript doesn't support overloading
  - If you define two functions with the same name but different numbers of parameters…
  - Then the 2nd function definition replaces the 1st function definition

Default Values

- Since ECMA6, functions can also provide default values for when the value is not passed in

```
function sayHello(name="unknown") {
    console.log(name);
}

sayHello(); // name will be unknown in this case
```


neueda

Assigning Variables from Objects

- There are also some convenient syntaxes for assigning variables from objects
- For example, if you have a complex object structure, and you want two variables that refer to two bits of the structure you can do this

```
let complexObject = {
    name: "Fred",
    age: 30,
    address: {
        line1: "1 High Street",
        line2: "Bristol"
    }
}
let {name, address} = complexObject;
console.log(name);
console.log(address.line2);
```


neueda

Spread Syntax

- Another convenient syntax is referred to as spread syntax used a lot in React applications
- It allows for arrays or objects to be expanded when referenced

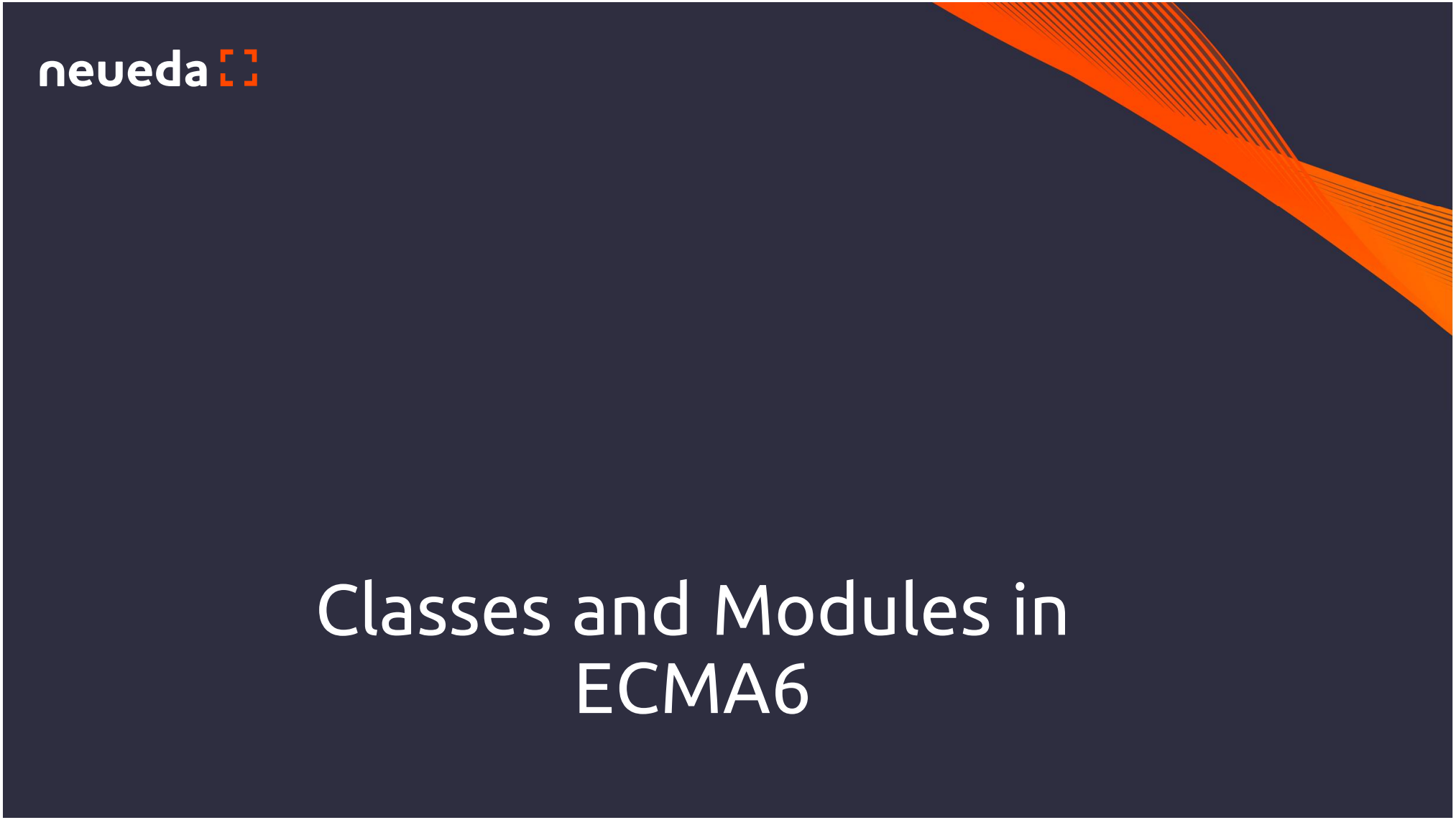
```
let complexObject = {
    name: "Fred",
    age: 30,
    address: {
        line1: "1 High Street",
        line2: "Bristol"
    }
}
function takeInObjectAndAugmentIt(someObject) {
    return {
        ...someObject,
        country: "UK"
    }
}
console.log(JSON.stringify(takeInObjectAndAugmentIt(complexObject)));
```

```
{"name":"Fred","age":30,"address":{"line1":"1
High Street","line2":"Bristol"},"country":"UK"}
```


neueda

Summary

- Working with variables
- Using operators
- Performing tests
- Performing loops
- Defining functions


neueda
Classes and Modules in ECMA6

Objectives

- Browser Support
- Writing classes
- Instantiating classes
- Inheritance
- Defining Modules

neueda

Browser Support

- The following Web page shows current browser support
  - https://kangax.github.io/compat-table/es6/
- In summary, these features **will work** in current versions of
  - Chrome / Firefox / Safari / Edge
- These features will **not work** in
  - Internet Explorer 11

neueda

Defining Classes

- Classes defined in a similar way to Java/C#

```
class Car {

        constructor(make, model) {
            this.make = make;
            this.model = model;
            this.speed = 0;
        }

        accelerate(){
            // must use the this keyword to access the property
            this.speed++;
        }

}
```


neueda

Creating Instances

- Instances are created using the **new** keyword
- Properties can be accessed in the same way as ECMA 5

```
class Car {
    …
}

var car = new Car("BMW", "5 series");
car.accelerate();
console.log(car.speed);
```


neueda

Getters and Setters

- JavaScript also has the ability to define properties a bit like C#
  - The constructor sets a field with an _ in front of the name
  - The **get** and **set** keywords are used by methods with the chosen property name
  - The property is then accessed using the name of the get/set functions

```
class Dog {
        constructor(name) {
           this._name = name;
        }

        get name() {
           return this._name;
        }

        set name(newName){
           if (newName) {
              this._name = newName;
           }
        }
     }

     var doggie = new Dog("Fido");
     console.log(doggie.name);
     doggie.name = “Barnie”;
```


neueda

Inheritance

- Inheritance is very similar to Java/C#

```
class SportsCar extends Car {
        constructor(make, model, turboBoost) {
            // call superclass constructor
            super(make,model);
            this.turboBoost = turboBoost;
        }
        // method overriding
        accelerate() {
            super.accelerate(); // call superclass method
            this.speed = this.speed * this.turboBoost;
        }
}

sportsCar = new SportsCar("Maserati", "4200", 4);
sportsCar.accelerate();
console.log(sportsCar.speed);
```

Static Methods

- Classes can also have static methods
  - Static properties are not possible

```
class Car {

        static bogStandardCar() {
            return new Car("Ford", "Fiesta");
        }

        constructor(make, model) {
            this.make = make;
            this.model = model;
            this.speed = 0;
}
var ordinaryCar = Car.bogStandardCar();
```

Defining Modules

- There are a number of ways of defining a module
  - Exporting specific functions and variables in a JavaScript file
  - Exporting all functions and variables in a JavaScript file
  - Export a mixure of Functions and variables from multiple files from one file - mixins

neueda

Exporting Examples

```
// exporting specific functions
export function makePayment(amount) {
    return "payment made for " + amount;
}

export function issueRefund(amount) {
    console.log("refund issued for " +
amount);
}
```

```
// exporting one thing
class Account {
}

// more things in the file not exported

export default Account
```


neueda

Summary

- Writing classes
- Instantiating classes
- Inheritance
- Defining Modules

neueda


neueda
Lambda Functions

Objectives

- What is a Lambda
- Defining a Lambda in ECMA6

neueda

What are Lambdas?

- A lambda is a syntax allowing for the definition of a method with no name ie. anonymous method
- They make many tasks syntactically easier
  - Looping code
  - Filtering code
  - Event handling
  - Asynchronous response handling

What is a Lambda

- A Lambda is essentially an anonymous function written in a specific way
- Consider iterating over a simple array and having a function to process each item

```
function processItem (nextItem) {
    console.log(nextItem);

}
arrayOfItems = ["item1", "item2", "item3"];
arrayOfItems.forEach(processItem);
```

- This could be simplified with a lambda function

```
arrayOfItems.forEach((item) => console.log(item));
```

Lambda Syntax

- The Lambda syntax is as follows

```
(any parameters) => method body;
```

- So, in the example, forEach function takes in a function to process each item

```
(item) => console.log(item);
```

- The method body will require {} if there is more than one line of code

```
() => {
    console.log("curlies required as ");
    console.log("more than one line");
};
```

Lambda Parameters and Return Values

- For one statement Lambdas the return is implied so the result of a Lambda is implicitly returned without the keyword
- If you have more than one statement then a return statement is required

Using lambda syntax to define functions

- The lambda syntax can be used as an alternative syntax to define regular functions.
- The following two examples are equivalent

```
function addNumbers (a,b) {
   return (a + b);
}
```

```
const addNumbers = (a,b) => (a + b);
```


neueda

Summary

- What is a Lambda
- Lambdas in JavaScript

neueda


neueda
Module Management

Objectives

- Node Application Structure
- Creating Node Projects
- Node Scripts
- Specifying Dependencies
- Specifying Dependency Versions
- Restoring Dependencies

neueda

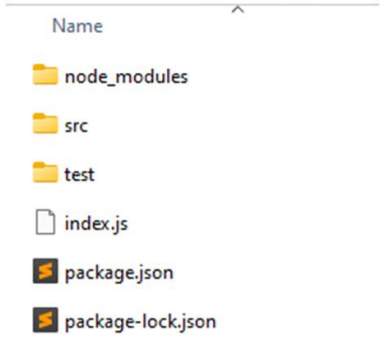
Node Application Structure

- Complex applications require considerations around
  - Code structuring
  - Test location
  - Dependency location
  - What goes in source control

neueda

Node Project Management

- Node projects are structured as follows:
  - The root folder contains 2 configuration files in JSON format.
  - Dependencies (other Javascript libraries) will be stored in a sub-folder called **node_modules**
  - The entry-point is called **index.js**
  - Code files are normally stored in a sub-folder called **src** (for “source”)
  - Test files are often stored in a sub-folder called **test**



neueda

Creating Node Projects

- New node projects can be created by running the **npm init** command
- This command should be executed in the target project folder

```
workspace> mkdir sample_app

workspace> cd sample_app

workspace/sample_app> npm init
```

- The npm command will execute node scripts – it will prompt to download a script from the node package manager registry if not already available on the local machine.

Creating Node Projects

- In the root folder for your new project, run
  - **npm init**

```
workspace/sample_app> npm init

This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible
defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
```


neueda

npm init Utility (1)

- **Package Name**: The init utility prompts for a package name, defaulting to the folder name containing your project
- **Version**: The version of your project. Defaults to 1.0.0
- **Description**: Description of the project
- **Entry point**: The name of the main script file – this will be executed when the module is run

```
package name: (sample_app)
version: (1.0.0)
description: sample application
entry point: (index.js)
test command: jest
git repository:
keywords: sample application
author: Nick Todd
license: (ISC)
```

npm init Utility (2)

- **Test command**: The test command is the command used to run the tests
  - In the example, jest has been used
- **Git Repository**: The location of the GIT repository (if relevant)
- **Keywords**: These help people find your script using **npm search**
- **Author**: The project author
- **License**: What software license you are using

```
package name: (sample_app)
version: (1.0.0)
description: sample application
entry point: (index.js)
test command: jest
git repository:
keywords: sample application
author: Nick Todd
license: (ISC)
```

The package.json

- Running the init utility creates the **package.json** file

```
{
  "name": "module_structure",
  "version": "1.0.0",
  "description": "An example of Module Structuring",
  "main": "index.js",
  "scripts": {
    "test": "jest"
  },
  "keywords": [
    "example",
    "nodejs"
  ],
  "author": "Nick Todd",
  "license": "ISC"
}
```

- You will need to create the **index.js** file manually.

Executing a node application

- Create a text file called **index.js** in the root folder

```
console.log("This is index.js");
```

- The application can be run with the command: **node index.js**
- You can create a script in the package.json file to specify that when the application is run, it is index.js that should be executed. This script is normally called start.

```
"scripts": {
   "start": "node index.js",
   "test": "jest"
 },
```

- The application can now be run with the **npm start** command.

Making the application into a module

- Configuring the application as a module makes it easier to write code – it allows you to import functions from other modules
- Edit the **package.json** file and add in the following line:

```
{
  "name": "sample_app",
  "version": "1.0.0",
  "type": "module",
  "description": "sample application",
```


neueda

Structuring a node application

- With the exception of the **index.js** file, most code will be in a folder called **src** (or subfolders)
- Index.js will contain code that imports and executes functions from these files.
- Create the following code structure:

src/numberFunctions.js

```
export const checkNumber = (value) => {
  return ("This function has not yet been
implemented.");
}
```

index.js

```
import {checkNumber} from './src/numberFunctions.js';

console.log(checkNumber(7));
```

- Run the application with the **npm start** command.

neueda

Adding Dependencies

- Dependencies can be added to your project using **npm install**
- The **install** command will
  - Download and add the required library to a **node_modules** folder
  - Update package.json with an entry specifying the dependency
  - Update / Create a **package-lock.json** file

neueda

Install Example

- As an example, the **prime-number-check** module will be installed
  - **npm install prime-number-check**
- This adds the following entry to package.json

```
"dependencies": {
   "prime-number-check": "^1.0.1"
 }
```

- The **^** sign means the latest version 1 can be used
  - A **~** can also be used which would mean any 1.0.x version can be used

The package-lock.json File

- An additional file is also created / updated called **package-lock.json**
- This file describes the exact versions of all your dependency trees

neueda

package-lock.json Example

```
{
  "name": "sample_app",
  "version": "1.0.0",
  "lockfileVersion": 3,
  "requires": true,
  "packages": {
    "": {
      "name": "sample_app",
      "version": "1.0.0",
      "license": "ISC",
      "dependencies": {
        "prime-number-check": "^1.0.1"
      }
    },
    "node_modules/prime-number-check": {
      "version": "1.0.1",
      "resolved": "https://registry.npmjs.org/prime-number-check/-/prime-number-check-1.0.1.tgz",
      "integrity": "sha512-
Wa6LwvNlkuh3rmbjdsheMAtR0SifxeGbdpHGEvCz3VOe6uivf1shXxzA9KlSAbyuiixyaiYELhJ5NbdVyy1wOw=="
    }
  }
}
```


neueda

Benefits of package-lock.json

- This file provides
  - A snapshot of every dependency and version
  - If you maintain this file in source control you can see your history of dependency changes
  - Allows you to avoid checking the node_modules folder into source control since it can be regenerated from the package-lock.json file
    - /node_modules will normally be included in .gitignore

Using the dependencies

- We can import functions from npm modules – Node knows where to find them so we just reference the module name.
- Edit the src/numberFunctions.js file:

```
import  isPrime from 'prime-number-check';

export const checkNumber = (value) => {
  return  isPrime(value) ? "The number is prime" : "The number is not prime";
}
```

- Run the application with the **npm start** command.

neueda

Dev Dependencies

- Not all packages will be required in production, such as test libraries
- Libraries such as these can be added with a flag
  - npm install jest **–D**

- The –D makes the module a **devDependency**

neueda

Jest Installation

- Once Jest is installed, it will be in your devDependencies section

```
"devDependencies": {
  "jest": "^23.6.0"
}
```

- Although less common, you can also use
  - **-O** which means optional
  - **-no-save** which means don't save in the dependency list

neueda

Creating a test file

- Jest will look for all files in the project called xxx.test.js and execute any tests contained in there.
- Create the following file:

test/prime.test.js

```
const isPrime =  require('prime-number-check');

test ('test that 3 is prime', () => {
   expect(isPrime(3)).toBe(true);
})
```

- Execute the tests with **npm test**

Locking Down Dependencies

- It also possible to provide a copy of package-lock.json called **npm-shrinkwrap.json**
  - This LOCKS the dependencies to the specified versions
- This file can be used if you wish to publish your package
- It is not suggested that you use this in your own libraries as it prevents people from controlling dependencies
- However it is used in global command line tools based on Node

neueda

Restoring Dependencies

- Once checked out of source control your node_modules folder will not be present
- Dependencies can be restored using
  - **npm install**

neueda

Summary

- Node Application Structure
- Creating Node Projects
- Node Scripts
- Specifying Dependencies
- Specifying Dependency Versions
- Restoring Dependencies

neueda


neueda
Asynchronous JavaScript

Objectives

- History of asynchronous JavaScript
- Callbacks
- Promise and Fetch
- Async and Await
- Generators

neueda

Asynchronous Defined

- Here is a good definition from Wikipedia
  - “In multithreaded computer programming, **asynchronous method** invocation (AMI), also known as **asynchronous method** calls or the **asynchronous** pattern is a design pattern in which the call site is not blocked while waiting for the called code to finish. Instead, the calling thread is notified when the reply arrives.”

Basic Callbacks

- A simple practice in JavaScript is to use callback functions as parameters

```
module.exports = function (inputValue, callback) {
   // do some processing on the input value
   // could take a while
   var resultOfLotsOfEffort = null;
   try {
      // not really much effort but just pretending
      var resultOfLotsOfEffort = inputValue;
   }
   catch (e) {
      // problem so call back with the exception
      // use the return keyword so the rest of the method doesn't run
      return callback(e);
   }
   // finally done, so callback with the result
   return callback(null, resultOfLotsOfEffort);
 }
```

Calling the Callback

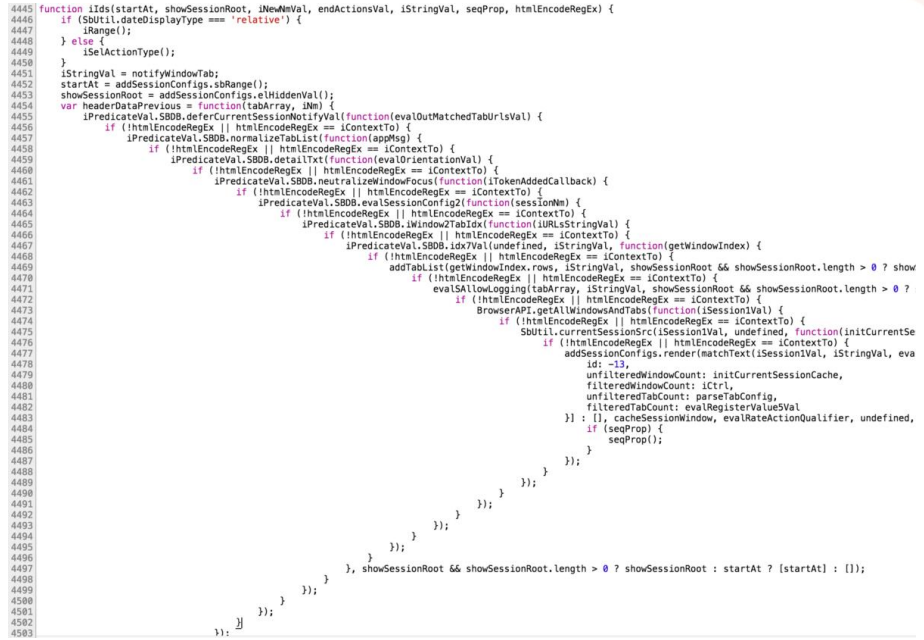
- The Callback can then be invoked from some other script

```
const callbacks = require("./callbacks");

function handleTheResultWhenWeGetIt(err, result) {
    if (err) {
        console.log("looks like the function didn't work");
    }
    else {
        console.log("got a result! " + result);
    }
}
callbacks("hello, please call back when you are done", handleTheResultWhenWeGetIt);
```

Limitations of Callbacks

- Cannot return values, instead you just invoke the callback function
- Cannot throw an exception only pass one back as a parameter
- Code can get very
  - Complicated
  - Hard to maintain



```
function iIds(startAt, showSessionRoot, iNewNmVal, endActionsVal, iStringVal, seqProp, htmlEncodeRegEx) {
    if (SbUtil.dateDisplayType === 'relative') {
        iRange();
    } else {
        iSelActionType();
    }
    iStringVal = notifyWindowTab;
    startAt = addSessionConfigs.sbRange();
    showSessionRoot = addSessionConfigs.elHiddenVal();
    var headerDataPrevious = function(tabArray, iNm) {
        iPredicateVal.SBDB.deferCurrentSessionNotifyVal(function(evalOutMatchedTabUrlsVal) {
            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                iPredicateVal.SBDB.normalizeTabList(function(appMsg) {
                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                        iPredicateVal.SBDB.detailTxt(function(evalOrientationVal) {
                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                iPredicateVal.SBDB.neutralizeWindowFocus(function(iTokenAddedCallback) {
                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                        iPredicateVal.SBDB.evalSessionConfig2(function(sessionNm) {
                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                iPredicateVal.SBDB.iWindow2TabIdx(function(iURLsStringVal) {
                                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                        iPredicateVal.SBDB.idx7Val(undefined, iStringVal, function(getWindowIndex) {
                                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                addTabList(getWindowIndex.rows, iStringVal, showSessionRoot && showSessionRoot.length > 0 ? show
                                                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                        evalSAllowLogging(tabArray, iStringVal, showSessionRoot && showSessionRoot.length > 0 ?
                                                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                                BrowserAPI.getAllWindowsAndTabs(function(iSession1Val) {
                                                                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                                        SbUtil.currentSessionSrc(iSession1Val, undefined, function(initCurrentSe
                                                                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                                                addSessionConfigs.render(matchText(iSession1Val, iStringVal, eva
                                                                                                    id: -13,
                                                                                                    unfilteredWindowCount: initCurrentSessionCache,
                                                                                                    filteredWindowCount: iCtrl,
                                                                                                    unfilteredTabCount: parseTabConfig,
                                                                                                    filteredTabCount: evalRegisterValue5Val
                                                                                                }] : [], cacheSessionWindow, evalRateActionQualifier, undefined,
                                                                                                    if (seqProp) {
                                                                                                        seqProp();
                                                                                                    }
                                                                                                });
                                                                                            }
                                                                                        });
                                                                                    }
                                                                                });
                                                                            }
                                                                        });
                                                                    }
                                                                }, showSessionRoot && showSessionRoot.length > 0 ? showSessionRoot : startAt ? [startAt] : []);
                                                            }
                                                        });
                                                    }
                                                });
                                            }
                                        });
                                    }
                                });
                            }
                        });
                    }
                });
            }
        });
```

neueda

Enter Promises

- Promises were introduced in JavaScript in 2012 and make writing callbacks clearer
- The concept is not new, and was invented in 1976

- ***"A promise represents the eventual result of an asynchronous operation"***

neueda

Promise Syntax

- A Promise object is created with up to two functions as parameters
  - **onFulfilled** A function for when it returns successfully
  - **onRejected** A function for when it does not

neueda

Callback Done as a Promise

- Below is the code example from earlier done as a promise

```
module.exports = function (inputValue) { // note no callback now
    // note function returns a promise
    return new Promise(function(resolve, reject) {
        // could take a while
        var resultOfLotsOfEffort = null;
        try {
        // not really any effort, but just pretending
        var resultOfLotsOfEffort = inputValue;
        }
        catch (e) {
            // problem so call back on the reject function
            return reject(e);
        }
        // finally done, so callback on the resolve function
        return resolve(resultOfLotsOfEffort);
    });
};
```

Calling the Function returning a Promise

- The calling code is now much cleaner

```
const callbackdoneasapromise = require("./callbackdoneaspromise");

callbackdoneasapromise("hello")
        .then((result) => console.log(result) ) // handle the success
        .catch((err) => console.log(err)) ; // handle the failure
```


neueda

REST API Call Example

- The **node-fetch** API is a good example of promises
- The node-fetch API is can be used to make REST API calls via the **fetch()** function
- The **fetch()** method is a method which returns a promise
  - The simplest use of fetch() is to give it a URL
- This returns a promise
  - The methods to handle the response is passed to a **then()** method of the promise

Node Fetch Structure

- Below is the structure of the node-fetch API in action

```
var fetch = require('node-fetch');

fetch(someJSONURL).
        then(function() { /* do this with the response */},
            function() { /* do this if it doesn't work */);
```


neueda

Chaining Promises

- When one promise returns you can pass the result to another promise
  - For example, you can extract JSON from the Response from the first promise using the json() method
    - This is another Promise!

```
fetch(someRequest).then(function(response) {
    return response.json(); // this is a promise of JSON
}).then(function(dataAsJson) { // the response is now the JSON object
    // do something with the JSON response
});
```


neueda

The Complete Example

- Below is a complete browser based example of a Fetch being used with a Promise to retrieve a JSON food menu

```
$(document).ready(function() {

    fetch('burgerRestaurantMenu.json').then(function(result) {
        return result.json(); // a promise of JSON
    }).then(function(menu) { // the menu is now the JSON
        $("#restaurantName").html(menu.RestaurantDescription.Name);
        $("#firstMenuItem").html(menu.RestaurantMenuCategories[0].Items[0].Title);
    });
});
```

Async/Await

- Async / Await is an alternative syntax for a promise
- A function with **async** in front of it means that the function will return a **promise**.
  - If an async function returns something that **isn't** a promise, javascript will convert whatever it returns into a **resolved** promise.

neueda

Async/Await

- The await keyword will pause the execution of the function until the promise assigned to it resolves.

```
let value = await SomePromise
```

- The await keyword can **only** be used from within an async function

neueda

Async Await Example

- Below is an example calling our code that returns a promise but using async/await
  - Note the use of try/catch for errors now instead of another function

```
// using the promise with async / await
async function callTheFunctionUsingPromisesUsingAsyncAwait() {
    try {
        var result = await callbackdoneasapromise("hello");
        console.log(result);
    }
    catch(err) {
        console.log("looks like the promise didn't work out");
    }
}

callTheFunctionUsingPromisesUsingAsyncAwait();
```

Async/Await With REST

- await waits for the promises to resolve before setting the variables

```
var fetch = require('node-fetch');

async function myAsyncFunction(){
  let response = await fetch('http://date.jsontest.com/');
  let jsonData = await response.json();
  console.log('The date is: ' + jsonData.date)
  return jsonData;
}

myAsyncFunction();
```

```
Output: The date is: 08-16-2018
```

Parallel Promises with Promises.all()

- As well as chaining promises you can trigger a group of promises in parallel

```
let [return1, return2] = await Promise.all([method1ReturnsAPromise(), Method2ReturnsAPromise()]);
```

- You have effectively created a super promise!
  - Once they all return, you will have an array of resolved promises!

neueda

Generators

- Generators are functions whose execution can be **suspended** and **resumed**.
- Generators are initialised in the **suspended** state
- Every time a generator is called with the .next() function, it will run from the last **yield** point, until the next one

neueda

Generator Example

- The keyword **function*** is used to create a generator
- They are called using the **.next()** function

```
function* generatorFunction(){
    console.log('this is the first time i have been executed');
    yield null;
    console.log('this is the second');
    }

    var generator = generatorFunction();

    console.log('activating once');
    generator.next();
    console.log('activating second time');
    generator.next();
```

```
activating once
this is the first time I have been executed
activating second time
this is the second
```


neueda

yield and next()

- The yield keyword suspends the function until a call to **next()** is made
- The **yield** keyword must always act on *something*, even if that something is **null**

```
function* generatorFunction(){
    console.log('this is the first time i have been executed');
    yield null;
    console.log('this is the second');
    }

    var generator = generatorFunction();

    console.log('activating once');
    generator.next();
    console.log('activating second time');
    generator.next();
```

Assigning values with next()

- In the example below, the yield line blocks *before* the assignment of the variable
- The second next() call results in the parameterPassedToNext variable being assigned
  - The value of the assignment will be parameter passed to next()

```
function* generatorFunction(){
  console.log("I've not yielded yet");
  parameterPassedToNext = yield console.log('I will print this, then yield');
  console.log(parameterPassedToNext);
}

var generator = generatorFunction();

generator.next();
generator.next('This text was passed into the yield value');
```


neueda

Complete Example

```
function* generatorFunction(){
  console.log("I've not yielded yet");
  parameterPassedToNext = yield console.log('I will print this, then yield');
  console.log(parameterPassedToNext);
}

var generator = generatorFunction();

generator.next();
generator.next('This text was passed into the yield value');
```

```
Output:
I've not yielded yet
I will print this, then yield
This text was passed into the yield value
```

Uses of Generators

- Consider a function that returns numbers in some kind of series
  - You can just call yield when you need the next one
  - The function itself contains an infinite loop but with a yield within it

```
function* fibonacciGenerator () {
  var current = 0, next = 1, swap
  while (true) {
    swap = current, current = next
    next = swap + next
    yield current
  }
}
```

```
fibo = fibonacciGenerator();
for (var i=0; i<10; i++) {
  console.log(fibo.next().value);
}
```


neueda

Summary

- History of asynchronous JavaScript
- Callbacks
- Promise and Fetch
- Async and Await
- Generators

RESTful Web Services

Objectives

- Introducing REST
- The principles of REST
- Implementing a REST Web service

neueda

Introducing REST

- REST stands for **Representational State Transfer**
- REST is a style of architecture
  - The Web is a large set of resources
  - As a client selects a resource, they are then in a particular state
  - The client can then select a different resource, and be in a different state as they now have a different resource
- For example, drilling into eBay to find a specific item

3

neueda

Identifying Resources

- One key concept in REST is the concept of a resource
- A resource always has a simple URL
  - http://www.conygre.com/courseList
- At the end of the URL will be URLs for other resources allowing you to get more information (and therefore change state)

neueda

Resource Example

- The courseList resource could be an XML document, containing links to further information for each individual course

http://www.conygre.com/courseList

```
<courses xmlns="http://www.conygre.com/courses"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <course id="XMLOV" xlink:href="http://www.conygre.com/site/xml/xmloverview"/>
  <course id="J2EEJBOSS" xlink:href="http://www.conygre.com/site/xml/j2eeJBoss"/>
  <course id="RUP" xlink:href="http://www.conygre.com/site/xml/rup"/>
  <course id="XSLT" xlink:href="http://www.conygre.com/site/xml/xslt"/>
</courses>
```


neueda

REST URLs

- Once important principle in REST is that the URLs are logical names for resources, not physical locations
- The actual location should be resolved by technology on the server
  - Such as a Java Servlet or ASP.NET routing rule

neueda

The REST Principles

- There is a request and a response
- Resources are all named through URLs
- Resources provide links to other resources
- There is a standard interface into the application using one of the HTTP methods

HTTP Method	Role in REST
GET	Retrieve data
POST	Add data
PUT	Edit data
DELETE	Remove data

REST and Web Services

- Web services that use REST tend to have the following characteristics
  - Clients send HTTP requests using the four HTTP methods
  - Simple parameters can be passed in the URL
  - Complex parameters are passed in the header as XML or JSON

Designing a RESTful API

- APIs can be developed using REST
- For example
  - GET www.meals.co.uk/restaurants?location=bristol
    - Get a list of Bristol restaurants
  - GET www.meals.co.uk/restaurants/2554
    - Get details for the Burger Joint Restaurant (restaurant #2554)
  - GET www.meals.co.uk/orders/454
    - Get the details for order number 454
  - DELETE www.meals.co.uk/orders/454
    - Remove order number 454
  - POST www.meals.co.uk/orders
    - (provide a JSON food order in the request header)
    - Create a new order

9

neueda

Security with REST

- With the URLs for RESTful APIs very simple and even guessable, it can be important to secure the API
- Approaches can include
  - Providing a user token to use with each request
  - Enforcing client authentication

Benefits of REST

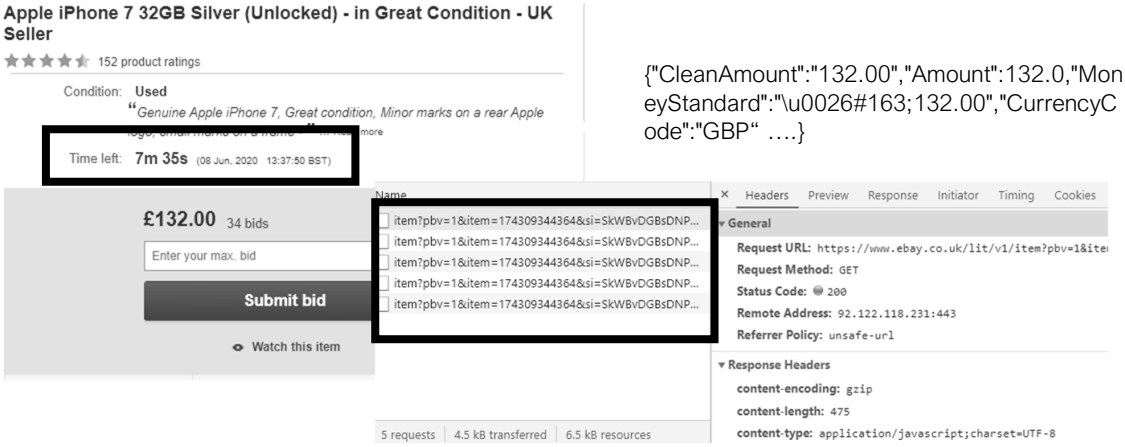
- REST based services are simple to implement
- Frameworks exist for multiple languages and platforms
  - Django / Flask for Python
  - ASP.NET API for .NET
  - Spring MVC for Java
  - Express for NodeJS
- REST based services are easier to invoke from browser based clients
- REST based services are ideal for simple request/response Web services

11

neueda

Ebay Example

- The pricing is updated by frequent calls to a REST API to get the latest price



12

neueda

How Mature is Your REST?

- Implementations of REST APIs vary in terms of the their level of maturity
- The Richardson Maturity Model defines four levels of REST
  - Level 0 – Just use HTTP as a RPC protocol
  - Level 1 – Just use POST or GET but also differentiate resources
  - Level 2 – use the appropriate HTTP verbs
  - Level 3 – within responses, provide URLs to be used for subsequent requests
- For more information, visit
  - http://martinfowler.com/articles/richardsonMaturityModel.html

Level 3 and HATEOAS

- With Level 3 services, the response from a request includes the URIs that can be used for subsequent requests
  - HATEOAS – Hypertext As The Engine Of Application State
- If you retrieved a list of restaurants, each restaurant would include the URL to get the menu
  - This means that the service provider could change those restaurant URLs if it wanted to

neueda

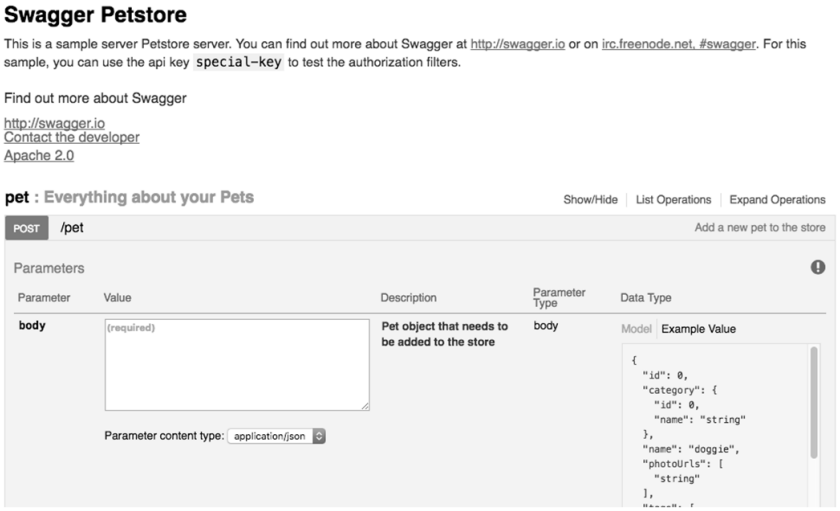
Self Describing APIs

- When you get to Level 3 based services, the APIs become self describing
- Once you know the entry point, the rest of the API is apparent within the responses
- Even better, is to use a documenting technology such as OpenAPI (formerly Swagger)
- Using OpenAPI, you can autogenerate client code in multiple languages, and the documentation allows people to understand the JSON structures and API calls
    - https://www.openapis.org/

15

neueda

OpenAPI Example



16

neueda

Summary

- Introducing REST
- The principles of REST
- Implementing a REST Web service

neueda


neueda
Server Side JavaScript

Objectives

- Running a Web server using the http library
- Creating a Rest API using ExpressJS
- Connect to a MySQL database using mysql

Basic Web Server Example

- To see how simple it is to get started with Node below is a 'HelloWorld' Web server

```
import http from 'http';

http.createServer(function (request, response) {
   response.writeHead(200, {'Content-Type':
'text/plain'});
   response.end('Hello World\n');
}).listen(8081);
```

- The example above creates a function that will be executed when HTTP requests come in on port 8081

Basic Web Server Example

To set up this example:

- Create a new folder called serverSideJS
- Run the npm init command to set up a new node application
- Configure package.json to include “type”: “module” and a start script to execute a file called basicserver.js
- nun the command **npm install http** to get the required dependency
- Create a file called basicserver.js

```
import http from 'http';

http.createServer(function (request, response) {
   response.writeHead(200, {'Content-Type':
'text/plain'});
   response.end('Hello World\n');
}).listen(8081);
```

- Run the application with **npm start**
- Visit http://localhost:8081 in your browser

4

neueda

NodeJS Libraries

- To get the full benefit from NodeJS additional libraries can be installed and used to complement the functionality
- Additional libraries are installed manually and then referenced in the application
- ExpressJS is a popular framework used in Web application development
    - http://expressjs.com/
- body-parser framework that will help us parse JSON received in the body of an http request
    - https://www.npmjs.com/package/body-parser
- mysql will help us connect to and use a MySql database
    - https://www.npmjs.com/package/mysql

- These are installed through NPM

Set up the project & database

- Before we start, set up the database by running the following command:

```
mysql -uroot –pc0nygre -h127.0.0.1 < databasesetup.sql
```

- Now get the required node dependencies with

```
npm install express body-parser mysql
```

- Create a new javascript file called **expressserver.js**, and edit the start script in package.json to run this file.

RESTful Server Example

- We will now build a RESTful server example providing a Create / Read / Update / Delete (CRUD) type functionality
- The example will work with a simple music catalog contained in a MySQL database.
- The first part of the NodeJS files loads the relevant libraries and initialises them

```
import express from 'express';
import bodyParser from 'body-parser'
import mysql from 'mysql'

//set up and configure express
const app = express();
app.use(bodyParser.json());

//set up and intialize the database connection
const connection = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "password",
  database: "albums"
});
```

RESTful Server Example

- We will create functions to handle the different **verbs** (GET, POST, PUT etc) to the different **endpoints** (URLs).
- At the end of the code, we star the server

```
//methods to handle requests to different
endpoints

...


//start the server
app.listen(8081, () => {
   console.log("Server is running.");
});
```


neueda

Working with the database

- Use the **.createConnection** function to create a MySQL connection object
- Use the **.query** function of the connection object to execute some SQL. The function takes 2 parameters – the SQL and a **handler function**
- The handler function takes 3 parameters, error, results and fields – **results** contains the response from the database.

```
const connection =
mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "c0nygre",
  database: "albums"
});

connection.query("SELECT * FROM albums",
    (error,results,fields) => {
      res.json(results);
    }
);
```


neueda

The Basic GET Method

- The basic GET function obtains the data from the database and returns the contents
- app.**get** – this is handling a GET request
- (**"/albums",** - this is the endpoint
- **(req, res) => {}** – this is a function that can reference the http request and http response objects.
- We call **res.json(...)** to provide the response.

```
app.get("/albums", (req,res) => {
  connection.query("SELECT * FROM albums",
    (error,
    results,
    fields) => {
      res.json(results);
    });
});
```


neueda

GET By Id

- A second GET function handles requests for individual albums by id
- Note that the connection.query function always returns an array so we extract the first item to return to create JSON of a single object.

```
app.get('/albums/:id', (req, res) => {
  connection.query(`SELECT * FROM albums where id = ${req.params.id}`,
    (error,
    results,
    fields) => {
      res.json(results[0]);
    });
});
```


neueda

DELETE an Album

- The method to delete an album is shown below
- Note that there is no data to return so we call the .end function of the response object to pass in a value and ensure a 200 status code.

```
app.delete('/albums/:id', (req, res) => {
    connection.query(`DELETE FROM albums where id = ${req.params.id}`,
    (error,
    results,
    fields) => {
      res.end("ok");
    });
});
```


neueda

POST a New Album

- Below is a POST method handler for new albums to be added to the file

```
app.post("/albums", (req, res) => {
  let sql = "INSERT INTO albums(title,artist,price,tracks)";
  sql += `VALUES
('${req.body.title}','${req.body.artist}',${req.body.price},${req.body.tracks})`
  connection.query(sql, function(error, results, fields) {
    res.end("added new item");
  });
});
```

PUT an Album Update

- This function handles an update for an order

```
app.put("/albums/:id", (req,res) => {
  let sql = `UPDATE albums set title =
'${req.body.title}',artist = ${req.body.artist} ,`;
  sql += `price = ${req.body.price},tracks =
${req.body.tracks}) where id = ${req.params.id}`;
  connection.query(sql, function(error, results, fields) {
    res.end("item updated if it exists");
  });
})
```


neueda
Testing with Jest

Objectives

- What is Jest
- Setting up Jest
- Matchers
- Basic Testing
- Testing Asynchronous Code
- Setup and Teardown

neueda

What is Jest

- Jest is a JavaScript testing framework developed by Facebook
- Jest is compatible with many Javascript libraries and frameworks such as
  - ReactJS
  - NodeJS

neueda

Setting up Jest with npm

- Create your node package file with

```
npm init
```

- Install jest as a dependency

```
npm install –D jest
```

- Configure your package.json to use Jest for testing

```
"scripts": {
        "test": "jest"
 },
```

- To run your test run the following in your project directory

```
npm test
```

Simple Jest example

```
module.exports = {
  sum: function(a,b) {
    return a+b;
  }
}
```

FileToTest.js

```
const testFile = require('./FileToBeTested');

test('Checks that the sum function returns 4 when given 2 and 2 as parameters', () => {
  expect(testFile.sum(2,2)).toBe(4);
});
```

FileToTest.test.js

```
Alex$ npm test

PASS  ./FileToBeTested.test.js
√ Checks that the sum function returns 4 when given 2 and 2 as parameters(5ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        12.918s
Ran all test suites.
```


neueda

Jest Test Files

- For every *file*.js you make, you then also write a *file*.test.js
- Every function you test in your test file must be exported from the file you are testing using:
  - module.exports = {}
- The test file must require() the file that will be tested

FileToBeTested.js

```
module.exports = {
  sum: function(a,b) {
    return a+b;
  }
}
```

FileToBeTested.test.js

```
const testFile =
require('./FileToBeTested');

test('Checks that the sum function
returns 4 when given 2 and 2 as
parameters', () => {
expect(testFile.sum(2,2)).toBe(4);
});
```


neueda

test() Function

- test() takes up to 3 parameters:
  - name of the test [type: String]
  - function that runs the test
  - timeout for that function

- From example on last slide:

```
test('Checks that the sum function returns 4 when given 2 and 2 as parameters', () => {
        expect(testFile.sum(2,2)).toBe(4);
},0);
```


neueda

expect()and matchers

- The expect function takes a function as a parameter and is always followed by a matcher
- The matcher describes what the function within the expect phrase should return
- In the last example the matcher used was .toBe(4) because we expect that sum(2,2) will return 4

neueda

Other matchers

List of matchers

- Errors
  - toThrow*(error type* or *error string)*
- Arrays
  - toContain()
- RegEx Strings
  - toMatch(/*RegEx*/)
- Numbers
  - toBeGreaterThan(*number)*
  - toBeGreaterThanOrEqual(*number*)
  - toBeLessThan(*number*);
  - toBeLessThanOrEqual(*number*)
  - toBeCloseTo(0.3) [for floating point rounding errors]
- Bools
  - toBeTruthy/toBeFalsy [anything an if statement treats as true or false resp.]
  - toBeDefined/toBeUndefined
  - toBeNull

- not [can be inserted before any matcher to invert it]
  - e.g. expect(sum(2,2)).not.toBeGreaterThan(4);

neueda

Asynchronous Testing

- Consider the following asynchronous test
  - This test will fail because the test will complete *before* the callback is called

```
test('The received data contains the word async', () => {
   function callback(data) {
      expect(data).toMatch(/async/);
   }
   getData(callback);
});
```

- To correctly test callbacks like this pass the keyword done into the anonymous test function, then call done() at the end of the callback

```
test('The received data contains the word async', (done) => {
   function callback(data) {
      expect(data).toMatch(/async/);
      done();
   }
   getData(callback);
});
```


neueda

Testing Promises

- To test a promise resolved to a certain value or object, use resolves before your matcher
  - Jest will then automatically wait for the promise to resolve before ending the test
  - If the promise is rejected then test will fail
- If you expect your promise to be rejected then use rejects instead of resolves
  - If the promise resolves the test will fail
- Alternatively, use async/await which is easier to read
  - See next page

Async/Await

- Declare the anonymous function passed to the test() to be an async function

```
test('The received data contains the word async', async () => {
    function callback(data) {
        await data = getData();
        expect(data).toMatch(/async/);
    }
});
```

- Await always returns a promise so you can combine this with resolves/rejects

```
test('The received data contains the word async', async () => {
    function callback(data) {
        await expect(getData()).resolves.toMatch(/async/);
    }
});
```

Before and After

- As with other testing frameworks there is the ability to have repeated setup and teardown
  - **beforeEach**
  - **afterEach**

```
beforeEach(() => {
  console.log("before each test")
});

afterEach(() => {
  console.log("after each test")
});
```


neueda

One Time Setup

- As well as beforeEach and afterEach you also have
  - **beforeAll**
  - **afterAll**

```
beforeAll(() => {
  // do this once only at the start
});

afterAll(() => {
  // do this once only at the end
});
```


neueda

Scoped Blocks

- It is also possible to group tests together into scoped blocks with their own ‘befores’ and ‘afters’

```
beforeEach(() => console.log(‘before each'));
afterEach(() => console.log(‘after each'));
test('', () => console.log(in the test'));
describe(‘Now in a scoped block', () => {
  beforeAll(() => console.log(‘before all in the block'));
  beforeEach(() => console.log(‘before each in the block'));
  test('', () => console.log(‘test now in a block'));
});
```


neueda

Summary

- What is Jest
- Setting up a node environment for Testing
- Matchers
- Basic Testing
- Testing Asynchronous Code
- Setup and Teardown

neueda

ECMA6 Exercises

Course Labs Contents

Conygre IT Limited –ECMA6 Labs

Introduction to your computer

Software

The following software should be installed on your operating system.

- NodeJS recent version
- A suitable editor such as Visual Studio Code, IntelliJ, or any other preferred IDE

Chapter 1: Introduction to NodeJS

Aims

In this lab, your aim is to gain familiarity with using the NodeJS environment. We will write a simple script that will output some text to the console.

Your First NodeJS Program

1. Using a text editor like notepad, create a new file and save it as **myfirstnode.js**.
2. Now we can insert some code to print something out to the console.

   ```
   console.log("Hello from my first nodejs program!");
   ```

3. Run your first nodejs program using a terminal in the same folder as your file. To do this, type the following at the command line;

   node myfirstnode.js

4. It should print out you're the text that is in your console.log statement.

Conygre IT Limited –ECMA6 Labs

NodeJS – The Basics

The Aims

This lab will introduce you to using simple variables, and basic operators.

Remember – JavaScript is case sensitive!

Declaring and initialising variables

1. Within the script from the last lab exercise, you will declare some variables to represent the car you drive, or if you do not have a car – a car you would like to drive! Firstly declare two variables called **make** and **model**, then declare a variable to be the **engineSize**, and declare a variable to be the **gear** your car is in.

2. Now initialise those variables with appropriate values. Put in some code so that the program will print out the values of these variables, things like;

   ```
   The make is x
   The gear is y
   The engine size is z
   ```

   You will need to use the string concatenator (+).

3. Fix any errors and run your script.

Looping and Branching

The Aims

This lab will introduce you to using the various flow control constructs of the JavaScript language.

Part 1 Using if / else

1. In your previous script, firstly, put in some logic to print out either that the car is a powerful car or a weak car based on the engine size, for example, if the size is less than or equal to 1.3.
2. Now, using an if / else if construct, display to the user a suitable speed range for each gear, so for example, if the gear is 5, then display the speed should be over 45mph or something. If the gear is 1, then the speed should be less than 10mph etc.

Part 2 Looping

3. We will now need to generate a loop which loops around all the years between 1900 and the year 2000 and print out all the leap years to the command console. You can use either a for or while loop to do this.
4. Once you have done this, set it so that after 5 leap years have been displayed, it breaks out of the loop and prints ‘finished’.

Part 3 Using switch / case

5. Now re-write part 1 to use a switch / case construct. It should do exactly the same thing as the if / else if construct. Don’t forget to use break.

Part 4 (Optional)

6. This will possibly require some research by you to find out how to do this. Create yourself an array.
7. Now modify your loop above, so that it no longer displays the first five years, but stores the first 10 in your array instead.
8. Now provide another loop to print out the values in the array. Use the length property of the array object to specify how many times to loop

Introduction to Objects and Classes

The Aims

This lab will introduce you to defining classes with instance methods and variables, and then instantiating and manipulating the resulting objects from a script.

Defining a class

We are going to create a new class called Account, and save it in a file called **Account.js.**

1. Create a new JavaScript file called Account.js and define a class within it.
2. Provide two properties called **_balance** and **_name** which you will have to set up in a constructor.
3. Now provide get and set blocks for your properties.
4. Define a new method called **addInterest**, which does not take in any parameters or return any value, but increases the balance by 10%. We will be using this method later.

Instantiating the class

5. Create another script called **TestAccount.js**.
6. Within this script, create a new Account object called **myAccount**, and then give it a name and a balance. Set name to be your name, and you can give yourself as much money as you like.
7. Now print out the name and balance to the screen using the get methods. Place appropriate text before the values using the string concatenator (+).
8. Run your program. It should print out your name and balance variables.
9. Call the **addInterest** method and print out the balance again. It should be different when you run it.

Conygre IT Limited –ECMA6 Labs

Working with Arrays

Aims

You will now take the previous exercise a little further and modify the code to work with an array of accounts rather than single account references.

For information about working with arrays in JavaScript, you can review the documentation at:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Creating an array of Accounts

1. Within the same script that you used in the previous exercise, declare an array of accounts called **arrayOfAccounts**.
2. Create two other arrays, which contain values for the names and balances of your account objects. Do this using two array initialisers. Make up values for these two arrays. Something like:

   ```
   amounts = [23,5444,2,345,34];
   names = ["Picard", "Ryker", "Worf", "Troy", "Data"];
   ```

3. Using a for loop, populate the object referenced by arrayOfAccounts with account objects specifying a name and a balance using values from your predefined arrays.
4. Within the loop print out the name and balance from each account object.
5. Finally, within the loop, call the **addInterest** on each object in the array. Print out the modified balances.

Conygre IT Limited –ECMA6 Labs

Working with Strings

The Aims

This lab will introduce you to using the string methods found in JavaScript. Documentation referring to the String methods can be found here:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Part 1 Manipulating Text

1. Create a new script file called teststrings.js.
2. Using a variable with the value ‘example.doc’, change the text to example.bak.
3. Create two string variables that are same and test them see if they are equal. If they are not equal show which of them lexicographically further forward (in the dictionary!).
4. Find the number of times "ow" occurs in "the quick brown fox swallowed down the lazy chicken"
5. Check to see whether a given string is a palindrome (eg "Live not on evil")

Part 2 Formatting Text

1. Print today's date in various formats.

Part 3 Optional Splitting Text

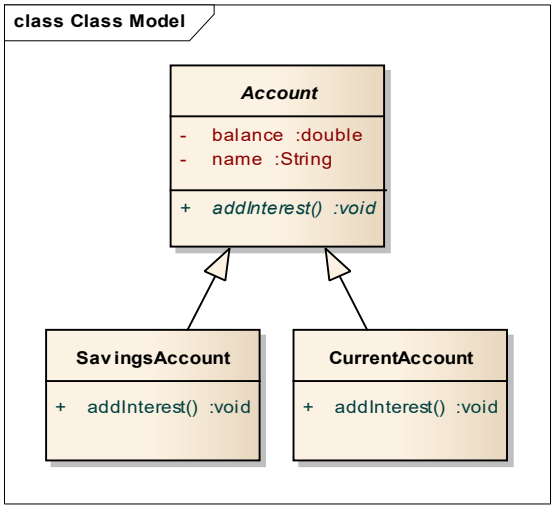
1. Copy a page of news or other text from your favourite website. Split the text into sentences, count and display them. Next split each sentence into words, count and display the words. Finally split the whole text into words and count them, checking that this is consistent with the results of the previous step.

Inheritance

The Aims

We all know that you cannot simply walk into a bank and open an 'account'. It has to be a particular 'type' of account. We are going to modify our classes so that we have a number of subclasses of the Account class, and we will be instantiating them.

.



Part 1: Defining the Subclasses

We will firstly define two subclasses of our class called **Account**. One will be **SavingsAccount**, and one will be **CurrentAccount**.

1. Define a new Javascript file called **SavingsAccount.js** and define a SavingsAccount that extends Account.
2. Define a constructor that takes two arguments for name and balance.
3. In the constructor you have defined, pass the two parameters to the superclass constructor using the *super* keyword.
4. Define a second class called **CurrentAccount**, in a new file repeating the steps above.
5. Now, in your two subclasses, override the addInterest method.
6. Multiply the balance by 1.1 in the current account and multiply the balance by 1.4 in the savings account.

Part 3: Instantiating our classes

1. Define a new javascript file called *TestInheritance.js*.
2. Declare an array called *accounts*, and initialise it with an Account object, a SavingsAccount object, and a CurrentAccount object. You do not need a loop for this bit.

   These objects should have balances of 2, 4, and 6 respectively. The names can

be whatever you like.

3. Now loop through the array, and call addInterest on each of the elements. Which addInterest will be called? There are three in total. One in each of the subclasses, and one in Account.

Streams

The Aims

In this chapter you will process a file path and check that it is a directory, and if it is, list the contents of a directory. You will then create a file copying program that will read a file and copy the contents in a different location.

Part 1 Using Basic Streams

1. Create a new script called *DirList.js*.
2. Using suitable nodejs modules and methods, check that the directory exists, and print out a message to say if it does or not.
3. Now check that the path is a directory and not just a simple file. If it is not a directory, display a message and exit.
4. Finally, list the contents of the directory on the console.

Part 2 Using the Stream Classes

What we will do in this part of the practical is implement a file copying program, using readers and writers.

1. Create a new script called *FileCopier.js*.
2. Create two file streams, one for reading which refers to an actual file, and one for writing which refers to an as yet non-existent file.
3. Read the input file and write it to an output file initially using a pipe.
4. Test your code and check that it works. Once you are satisfied that it works, refactor it to use event handlers and chunking to deal with the writing instead.

Creating a Basic Alexa App

The Aim

In this chapter you will develop a basic understanding for the workflow involved in creating a simple Alexa app. The app will ask the name of the user and speak it back to them in a greeting.

Part 1 Creating the Interaction Model

First, you will set up the Alexa app functionality - what the user will say to invoke the app and what intents and slots will be required.

1. First go to the **Alex Skills Kit** and log in. [https://developer.amazon.com/alexa-skills-kit]
2. Then hover over **Your Alexa Consoles** in the top right and click on **Skills**
3. Click on **Create Skill** and call it something like 'Simple Greeter'
4. Set the **Default Language** to your preference and make sure that **Custom** is the selected model
5. Click **Create Skill**
6. When asked for a template select **Start from Scratch** and click **Choose**.

Here you are shown your console for the Interaction Model. We will need to give our skill an **Invocation Name** (The name that Alexa looks for in user input to start our app) as well as some **Intents** and **Slot Types**.

7. Click on **invocation** in the left-hand-side panel
8. In the **Skill Invocation Name** box, set it to something like '**simple greeter**' (There are some rules on what an invocation name can contain which are in a green box below the text box)
9. Click on **Save Model** above.
10. Click on **Intents** in the **left-hand-side** panel.
11. Click **Add Intent.**
12. Call your intent something sensible like **greet.**

Now you have to come up with some phrases that could be used to start your greet intent. For example: 'my name is {name}'. *The value in the curly brackets signifies that the value is going to be something which belongs in the **name** slot.*

13. Add a few more phrases that could initiate your intent. The more phrases, the better.
14. If you used a value in curly brackets, then the console will have created a corresponding slot for you, but you will need to define a slot **type**.
15. In this case our slot is **name** so our slot type will be AMAZON.GB_FIRST_NAME.
16. Select **AMAZON.GB_FIRST_NAME** in the slot type **drop-down**.
17. Now we are ready to move onto creating the logic for the app to use.

Part 2 Creating the Lambda

In this section you will create the lambda function that will be called by the Alexa app that handles all of the logic. For this app it will be fairly straight-forward.

1. Click the **Endpoint** section on the left hand side of Alexa console
2. Then selected the **AWS Lambda** radio button
3. Now head to the AWS console (https://aws.amazon.com/lambda/)
4. Log into the console
5. Select **lambda** from the services drop down (top left)
6. Click on **create function**
7. Click on **serverless application repository**
8. Then search for the **alexa-skills-kit-nodejs-factskill**
9. Select it, then give it a name (bottom right)
10. Click **deploy**

You will not be using the code that gets automatically generated, but it is helpful to use this approach, as it generates all the roles and permissions for you. Later you will do this yourself.

11. Once your code has been deployed you will see a green box telling you that you can now use it.
12. Click on **Test App.**
13. Find your function in the list. It will be called something like: aws-serverless-repository-alexaskillskitnodejsfact-xxxxxxxxxxxxx.