CONTENTS

QuickLabs Environment Set-Up

Code Editing

1. Open **VSCode** (or download and install if not present).
    - Use the *desktop shortcut* to open the **VSCode** download page:
    - For **Windows** users download the **64-bit System Installer**.
2. Check for *updates* and download and install if necessary:
    - For **Windows** Users click **Help - Check for updates;**
    - For **MacOS** Users click **Code - Check for updates.**
3. Using **File - Open,** navigate to the **QuickLabs** folder and click **Open.** This will give you access to all of the **QuickLab** files and solutions needed to complete the **QuickLabs.**

NodeJS

1. Navigate to https://nodejs.org/en/download to open the **NodeJS** download page.
2. Download and install the **LTS** version for the operating system you are working in:
    - For **Windows** users, download the **Installer file (.msi);**
    - For **MacOS** users, download the **Installer file (.pkg).**
3. Check that the installation has worked by opening a terminal and typing:

```
node -v
```

and then

```
npm -v
```

This is the end of Quick Lab Environment Set Up

Quick Lab 1 - JavaScript Types

Objectives

- To be able to create and log out variables
- To be able to manipulate strings
- To be able to use Node.js to run JavaScript files

Activity – Part 1 – Numbers

1. In **VSCode**, open the file **numberTypes.js** from the **QuickLabs/ql01-JavaScriptTypes/starter** folder.
2. Add a value to `numTest` – make it have at least 4 digits after a decimal point.
3. Open **VSCode**'s in-built console by selecting **View → Integrated Terminal** (*or use the shortcut key or icon on the bottom bar*).
4. Ensure that the terminal is pointing to the **QuickLabs/ql01-JavaScriptTypes/starter** folder and execute the file using node:

```
node numberTypes.js
```

5. Node executes the file and prints any `console.log` commands to the terminal.
6. Note that there is a second value of `undefined` – this is the current value of `twoDecimalPlaces`.
7. Set the value of `twoDecimalPlaces` to be `numTest` with the function `toFixed` called on it – this should have the value `2` passed as an argument:

```
let twoDecimalPlaces = numTest.toFixed(2);
```

8. Save the file and re-run the code using the node command and observe the output now.

You will notice that the values are printed in different colours (on some operating systems).

9. Add another `console.log` under the last one to log out the **type of** the variable `twoDecimalPlaces`.

```
console.log(typeof twoDecimalPlaces);
```

Activity – Part 2 – Strings

1. Open **stringTest.js** from **QuickLabs/ql01-JavaScriptTypes/starter.**

2. Add a value to the declaration of `indexOfM` variable that sets it to a call to `indexOf` on `stringTest` with an argument of `` `m` ``:

```
let indexOfM = stringTest.indexOf(`m`);
```

3. Save the file and run it using node:

```
node stringTest.js
```

You will see a value of `3`, examine the string and you will see that the `m` is the fourth character, so there are three characters before the first `m`.

4. Change the `m` within the `indexOf` method call to a capital `M`, save and observe the output in the console again.

This time, the console.log will return a `-1` value. The `-1` value is telling us that there is no match within the string at all proving that string searches are case sensitive. What if we convert the string to upper case?

5. Before the `indexOfM` line, add the following code:

```
stringTest = stringTest.toUpperCase();
```

6. Save and observe the output in the console again.

The output will, once again, give a value of `3`. Behind the scenes, the string is an indexed collection of characters and the search function is making its way through the letters character by character until it makes a match. With that concept in mind, we will use the principals to learn how to slice a string.

7. Add the following code under the last line, then capture start and end in a `console.log`, save and observe the output.

```
let start = stringTest.indexOf("MODEL");
let end = stringTest.lastIndexOf('MAJOR');
```

This time, we have matched based upon words, but you could search for file paths or extensions; for instance, if we were reading from a form. The two integer values held can be used to create a substring from the longer one using string's substring method.

8. Add the following lines of code to the end of your code, then save and observe the output.

```
let subStr = stringTest.substring(start, end);
console.log(subStr);
```

The console should now return a value of `"MODEL OF A MODERN"`.

This is the end of QuickLab 1

Quick Lab 2 - Arrays

Objectives

- To investigate JavaScript arrays and their functions

Activity

1. In **VSCode**, open **index.js** from **QuickLabs/02_JavaScriptArrays/starter/src.**
2. Run the file using `node array.js` and observe the browser to check the output - *you should see details of an array.*
3. Access the index of the array that contains the string "your" and log the array element to the console.

```
console.log(quote[2]);
```

4. Save the file and observe the browser to check the output.
5. Using the `pop` function, remove the string `` `friend` `` from the end of the array.
6. Using the `push` function, add the string `` `father` `` to the end of the array.
7. Log the array to the console again.
8. Save the file and observe the output.
9. Use the `unshift` function to add the string `` `Luke` `` to the start of the array.
10. Log the array to the console again.
11. Save the file and observe the output.

There are two things wrong with the output. The first is that it the string is concatenated by commas and the second is that the 'quote' is actually a misquote! We're going to generate an output in a different way by looping through the array and creating a new string. We're going to fix the misquote by detecting the erroneous word in the array and replacing it with the correct word! Let's do that first.

To do this, we are going to detect if indeed the erroneous word is in the array. If it is, we are going to find the index that the word is at and then use this information to replace that index with the correct word.

12. Declare a variable called `erroneousWord` and set it to a string with the misquoted word from the array (it's `Luke` if you didn't know!).
13. Set a variable called `lukeIsHere` using the `find()` function to see if the quote array contains the `erroneousWord`. The

code is:

```
let lukeIsHere = quote.find(n => { return n === erroneousWord});
```

The syntax inside the find function will feel a little alien at the moment but go with it as it is explained later in the course.

14. Declare a variable called `lukeIsAt` without assigning it.

15. If `lukeIsHere` has been set to `true`, find the index that the `erroneousWord` sits at using the `findIndex()` function and set `lukeIsAt` to the value of the `index`. The code is:

```
let lukeIsAt = quote.findIndex(n => { return n === erroneousWord});
```

16. Still inside the `if` block, use the value of `lukeIsAt` to set that index in the quote array to the string `` `No` ``.

```
if (lukeIsHere) {
    lukeIsAt = quote.findIndex(n => {
        return n === erroneousWord
    });
    quote[lukeIsAt] = "No";
}
```

17. Log out the array and ensure that the expected result is outputted in the console.

18. Declare a variable called `output` and set it to be an empty string.

19. Create a `for` loop that:

20. Loops through the quote array.

21. Executes when the loop counter is less than the `length` of the array.

22. Adds an exclamation mark to the `output` string, if we are at the last element in the array.

23. Adds a comma and a space to the `output` string, if the current element is `` `No` ``.

24. Otherwise adds a space to the `output` string.

```
for (let i = 0, j = quote.length; i < j; i++) {
    if (i === j - 1) {
        output += quote[i] + '!';
    } else if (quote[i] === 'No') {
        output += quote[i] + ', ';
    } else {
        output += quote[i] + ' '
    }
}
```

25. Log out the `output` string.

26. Save the file and then check your console output to ensure that the correct quote is displayed.

```
No, I am your father!
```

This is the end of Quick Lab 2

Quick Lab 3 - Objects

Objectives

- To understand how to declare and destructure objects

Activity

1. In **VSCode, objects.js** from **QuickLabs/ql03_JavaScriptObjects/starter.**

For clarity of instructions, the steps to save and observe the browser have been omitted after each instruction that affects the output.

2. Create a new `Object` called `darthVader` and add the following *key/value* pairs to it:
    - `allegiance` - `Empire`;
    - `weapon` - `lightsabre`;
    - `sith` - `true` (boolean value).
3. Access the properties that you have just declared by logging out the following details:
    - DarthVader's `allegiance`;
    - Darth Vader's `weapon`;
    - If Darth Vader is a `sith`;
    - The value of `Jedi` from Darth Vader;
    - The number of properties Darth Vader has (see the line of code below for this)

```
console.log(Object.keys(darthVader).length);
```

Quick explanation - Object.keys is a function that takes an object and returns an array of the keys in it. By appending .length to it, we return the number of keys in the object.

- Add *key/value* pairs to `darthVader` that:
- Sets a key of `children` to `2`;
- Sets a key of `childNames` to the array `['Luke', 'Leia']`;

and then log the `children` property and the *value of the first element* in the `childNames` array.

4. Iterate over `darthVader` using a `for…in` loop that uses both the *key* of each pair, logging out each pair's key and its value.
5. Manipulate the object by:
    - *Changing* the value of `allegiance` to `The light side` and log out `darthVader`;

- *Deleting* the key/value pair `children` and log out `darthVader`;

Hint: use the code below:

```
delete darthVader.children;
```

- *Destructuring* the object, setting a variable for each of the keys in the object to the corresponding value in the object:

```
let{allegiance, weapon, sith, childNames} = darthVader;
```

  - Log each individual variable out to ensure that they have been set.

- *Clearing* the object and logging it out.

This is the end of QuickLab 3

Quick Lab 4 - Collections

Objectives

- To be able to create and manipulate a Map in JavaScript

Activity

1. In **VSCode**, open **maps.js** **QuickLabs/ql03_JavaScriptMaps/starter**.

For clarity of instructions, the steps to save and observe the browser have been omitted after each instruction that affects the output. Run the command `node maps.js` after saving each time to see the output.

2. Create a new `Map` object called `hanSolo` and add the following key/value pairs to it:
    - `vehicle` - `Millenium Falcon`;
    - `bff` - `Chewbacca`;
    - `sweetheart` - `Leia`.
3. Access the properties that you have just declared by logging out the following details:
    - The `size` of the map `hanSolo`;
    - Han Solo's `vehicle` name (***HINT*** - use the `Map.get()` method);
    - If Han Solo has a `sweetheart` (***HINT***: use the `Map.has()` method);
    - If Han Solo is a (`has`) `Jedi`.
4. Add another *key/value* pair to `hanSolo` *that sets a key* `son` *to* `Ben` *and log this new property to the console.*
5. Iterate over `hanSolo` using a `for…of` loop that uses both the key and the value of each pair, logging out each pair.
6. Manipulate the map by:
    - *Changing* the value of `bff` to `Luke` and log out `hanSolo`;
    - *Deleting* the key/value pair `son` and log out `hanSolo`;
7. *Clearing* the `Map` and logging it out.

This is the end of QuickLab 4

Quick Lab 5 - Functions

Objectives

- To investigate JavaScript function scope
- To create functions that return data

Activity – Part 1 – Defining and using Functions and understanding scope

1. In **VSCode**, open the file **functionsAndScope.js** from the **QuickLabs/ql05-JavaScriptFunctions/starter** folder.
2. Declare a function called `findMovie` that takes an argument called `movieTitle`.
3. In the *body of the function* create a `for…of` loop of the `movies` array where:
    - The loop body should:
        - Check to see if the current *movie title* is the same as the `movieTitle` passed into the function and *if it is*, log out details of the movie in a suitable string;
        - Log out the value of `movie` before the loop's closing brace;
        - The value of `movie` should be logged before the function closes.
4. Call the `findMovie` function with an argument of `` `Star Wars` ``.
5. Log out the value of `movie`.
6. At this point, save your file and check the output.

The expected outcome is that there is a Reference Error - but which console.log is, or console.logs are, causing it/them?

7. Comment out the offending `console.log(s)` and check your output.

You should see all 5 movies logged, with the string you wrote for a found movie being outputted before the movie object for it is logged itself.

Two of the `console.log` statements added produced a ***Reference Error.*** This is because of the scope of the variable `movie`. As it is declared as part of the `for…of` loop, its scope is limited to inside the body of this block (i.e. between the { } that immediately follows the for). As long as execution remains inside this loop, the variable `movie` is in scope.

Once the loop finishes and execution returns to the level above (i.e. back to the body of the function) and the variable `movie` is no longer in scope and therefore referring to it in the code causes the

Reference Error. It follows that if `movie` is not available here, it will also not be available after the line that calls the function has completed execution, again causing a ***Reference Error***.

Note: Because `movieTitle` is part of the function block, it is accessible throughout the execution of the function, including inside any blocks that are used within the function body (i.e. in the `for` and `if` blocks).

Note: Because the `const` `movies` is declared at script level (i.e. inside this file) and at the top of it, it is available to all blocks of code that live inside this file.

8. Under the last line, define a variable called `movie` set to the value of `` `Thor: Ragnorok` ``.
9. *Uncomment* the `console.logs` of movies and *add another log* of the value of `movie` under the declaration of the variable from step 11.
10. Observe the results.

What you should see this time is the return of the ***Reference Error***, suggesting that movie is the problem.

11. Change the declaration of `movie` to have the `var` keyword in front of it (rather than `let`) and make sure that all `console.logs` are *uncommented*.
12. Observe the results.

What you should see this time is an undefined value again followed by the set value. The differences are all to do with concepts called hoisting and 'temporal dead zones'. More details of which can be found, with a good explanation of `let` at:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

13. Add a call to the `findMovie` with the argument set to `movie`, saving and observe the output.

There is no output from the function call for `findMovie` using the defined movie variable. This is because there is no movie in the movies array with the title `` `Thor: Ragnorok` `` and therefore the loop completes without ever entering the `if` condition.

Activity – Part 2 – Make a function return data

Functions rarely just execute code and then the program continues. It is more usual that a function will manipulate some data and then return some data which can then be used further. This part of the exercise will allow you to experiment with returning data from a function.

1. Open **QuickLabs/ql05-JavaScripFunctions/starter/functions2.js.**
2. Declare a function called `returnMovie` that takes `movieTitle` as an argument and has a function body that:
    - Uses a `for…of` loop on the `movies` array with a loop body that:
    - Checks to see `if` the `title` property of the current movie matches the `movieTitle` supplied to the function;
        - If it does, it should simply `return` the current `movie`;
    - Logs out the current value of `movie`;
    - Logs out `` `Any text, any text at all` ``.
3. In the body of the script, declare a variable called `myMovie` and set it to the result of calling `returnMovie` with an argument of `` `Avengers: Infinity War` ``.
4. Log out the value of `myMovie`, save and observe the output.

If you have created your `returnMovie` function you should observe the following:

- Each of the movies that appear BEFORE the selected movie are logged out as the loop has executed for each of these movies;
- The movies that are AFTER the selected movie are not logged out because the presence of the return statement stops the execution of the loop and indeed the function (so that "Any text, any text at all" is also not shown);
    - The execution 'returns' to its call point with the value of whatever is returned.

5. Access the properties of `myMovie` to produce and log a string as a sentence with them in it, saving and observing your output.

What happens if we try to pass a movie title that doesn't exist in the movies array into returnMovie? Let's find out!

6. Declare a variable `myOtherMovie` and set its value to a call to `returnMovie` with an argument of `` `Thor: Ragnorok` ``.
7. Log out the value of `myOtherMovie` and observe the output.

The first thing that we notice is that the whole of the movies array has been logged out and the text `Any text, any text at all`. This is because the title was not found, and the function completed its execution fully and never returned a value…or did it?

The next thing that we notice is that the console.log of `myOtherMovie` has outputted undefined. It looks like we've never set the value of `myOtherMovie` because we haven't! Let's fix that…

8. Comment out the logging of `Any text…` and add a line that returns the string `` `Movie not found` ``.

9. Save and observe the output.

The logging of `myOtherMovie` now outputs `Movie not found`.

The code is still not very reusable as if I want to log out the details of a movie, I have to supply the string inside a console.log. Also, what happens if it is already a string (because it is a movie not in the array)? Our output would be very messy! Step up another function!

10. Create a function called `myMovieDetails` that takes a variable `anyMovie` as an argument.

11. Check that the `typeof` `anyMovie` is an `'object'` and return a suitable string if it is and simply return `anyMovie` if it isn't.

12. Inside a console.log, call `myMovieDetails` with an argument of `myOtherMovie`.

13. Observe the results.

It should output: Movie not found.

Can we use a function as the argument to another function? Yes we can!

14. Repeat the last instruction instead passing in `returnMovie` with an argument of `` `Jaws` `` as the argument to the `myMovieDetails` function.

15. Observe the results.

It should output the details for Jaws in your defined string.

This is the end of Quick Lab 5

Quick Lab 6 - Structural and Attribute Directives

Objectives

- To be able to use the class syntax in JavaScript and instances of it.

Activity

1. Open **QuickLabs/ql06-JavaScriptClasses/starter/motorvehicle.js.**
2. Create a class called `MotorVehicle` with a constructor that takes the following arguments:
    - `make`; `model`; `wheels`; `engineSize`.
3. Inside the `constructor`, set the values of each - use the 'private' notation.
    - Add a `_speed` property set to `0`.
4. Add a 'getter' for each of the 5 properties.
5. Add an `accelerate` method that takes an argument of `time` and sets the `speed` to:

```
this._speed = this._speed + ((0.25 *
this._engineSize/this._wheels) * time);
```

6. Add a `brake` method that takes and argument of `time` and sets the `speed` if it is greater than 0 to:

```
this._speed = (this._speed - ((0.3 * this._engineSize/this._wheels)
* time) > 0)

this._speed = this._speed > 0 ? this._speed : 0
```

7. Create an instance of the `MotorVehicle` using `myMake`, `myModel`, `4`, and `2000` as constructor arguments:

```
const myVehicle = new MotorVehicle(`myMake`,`myModel`, 4, 2000);
```

8. Log out `myVehicle` and then it's `speed`.
9. Make the vehicle `accelerate` for `10`:

```
myVehicle.accelerate(10);
```

10. Log out its `speed` again - it should be `1250`.
11. `brake` for `5` and log out the `speed` - it should be `500`.
12. `brake` again for `5` and log out the `speed` again - it should be `0`.

This is the end of Quick Lab 6

Quick Lab 7 - Inheritance

Objectives

- To be able to use the class extends syntax in JavaScript

Activity

1. Open **QuickLabs/ql07-JavaScriptInheritance/starter/motovehicle.js**.
2. Under the `MotorVehicle` class, create another class called `Car` that `extends` `MotorVehicle`.
3. Add a `constructor` that takes the arguments:
    - `make`; `model`; `engineSize`; `doors`; `satNav` set as `false` by default; `wheels`.
4. Inside the constructor:
    - Make a `super` call with `make`; `model`; `wheels` set as `4` by default; `engineSize`.
    - Set the values of `doors` and `satNav`.
    - Add 'getters' for `doors` and `satNav`.
    - Add a 'setter' for `satNav`, taking `satNav` as an argument and using this to set the value.
5. Create a new instance of the `Car` class, called `myCar`.
6. Use any values you want here - don't include a value for wheels.
7. Log out `myCar`.
8. Add another **class** called `Motorbike` that `extends` `MotorVehicle`.
9. Add a constructor that takes the arguments:
    - `make`; `model`; `engineSize`; `driveType`; `wheels`.
10. Inside the constructor:
    - Make a `super` call with `make`; `model`; `wheels` set as `2` by default; `engineSize`.
    - Set the value of `driveType`.
    - Add a 'getter' for `driveType`.
    - Override the `MotorVehicle` implementation of `accelerate` by adding an `accelerate` method to the `Motorbike` class:

```
accelerate(time) {
  this._speed = this._speed + ((0.5 * this._engineSize /
    this._wheels) * time);
}
```

11. Make a new instance of `Motorbike` using any values you like - we used Kawasaki Ninja with a 650 sized engine and a chain drive-type.

12. Compare the implementations of the `accelerate` method for the `Car` instance and the Bike instance.

13. We suggest you `accelerate` both for `10` and see who won the speed trap race in a console.log!

This is the end of Quick Lab 7

Quick Lab 8 - Asynchronous JavaScript - JSON

Objectives

- To be able to create a properly formed JSON file.
- To be able to install and run json-server

Activity

1. In **VSCode**, create a file **reactrangers.json** in the **QuickLabs/ql08-JSON/starter** folder.
2. Start the file with an opening and closing set of **curly-braces**:

```
{

}
```

3. Add a **key** of `results` with the value of an *empty array*:

```
{
  "results": [

  ]
}
```

4. Inside this array add at least 2 objects (separated by a comma) that have 5 key/value pairs (**id** should increment with each). The object should look like the example below:

```
{
  "id": 1,
  "home": "React Rangers",
  "away": "Angular Athletic",
  "homeScore": 2,
  "awayScore": 0
}
```

5. Save the file.
6. In **VSCode's** terminal window, initialise another terminal by clicking the ✚ button.
7. Install **json-server** globally using:

```
npm i json-server -g
```

8. Ensure that the terminal is pointing to the **src** folder for this exercise, then spin up **json-server** using the command:

```
json-server reactrangers.json
```

9. Open your browser at:

```
http://localhost:3000/results
```

10. You should see the data from the file presented on the screen.

This is the end of Quick Lab 8

Quick Lab 9 - Asynchronous JavaScript - Promises

Objectives

- To understand how Promises work.

Activity

1. In **VSCode**, open the file **QuickLabs/ql09-Promises/starter/promises.js**.
2. Create a function called `runPromise()`.
3. Inside the function, declare a variable called `aPromise` that is a new `Promise` whose `constructor` has an *arrow function* that:
    - Takes `resolve` and `reject` as arguments

```
let aPromise = new Promise((resolve, reject) => {
}
```

    - Has a function body that:
        - Declares a variable called `delayedFunc` that is set as follows:

```
…
  let delayedFunc = setTimeout(() => {
    //whether it resolves or rejects is unknown
      let randomNumber = Math.random();
      (randomNumber < 0.5) ? resolve(randomNumber) :
        reject(randomNumber);
  }, Math.random() * 5000); //function returns in: 0-5s
…
```

The fact that we have used `setTimeout` here and the final argument `Math.random() * 5000` (which generates a random number between 0 and 1 and multiplies it by 5000) means that the *arrow function* will execute somewhere between 0ms and 5000ms. The arrow function itself generates a random number between 0 and 1 and the `Promise` is resolved if the number is less than 0.5 and rejects otherwise.

4. Call `aPromise` with a `.then` chain and set `data` to be the resolved value and log this out with `` `Resolved: ` `` as a prefix.
5. Add a `catch` block and set `error` to be the rejected value and log this out with `` `Rejected: ` `` as a prefix.
6. Run the file several times using node and ensure that the promise both resolves and rejects.

This is the end of Quick Lab 9

Quick Lab 10 - Asynchronous JavaScript - Fetch

Objectives

- To be able to use the Fetch API to be able to send and receive data.

Activity – Part 1 – GET Requests

Node.js does not natively support the `fetch` function, so we are going to install an **npm** package to allow us to use it here. The package is called **node-fetch** and is installed with the command:

```
npm i node-fetch --save
```

1. Execute this command with the terminal pointing to **QuickLabs/ql10-fetch/starter**.
2. Create a file in the same folder called **fetch.js**.
3. At the top of the file, `require` `fetch` from the `node-fetch` package:

```
const fetch = require(`node-fetch`);
```

4. Declare a variable called `reactRangersResults` and set this to be a call to `` `http://localhost:3000/results` `` via fetch.

```
let reactRangersResults =
  fetch(`http://localhost:3000/results`);
```

5. Chain a call to `then` with a callback arrow function that passes in `results`, returning `results.json()`:

```
.then(results => results.json())
```

6. Chain another call to `then` with a callback arrow function that passes in `results` returning a `console.log` of `results`:

```
.then(results => console.log(results))
```

7. Chain a call to `catch` with a callback arrow function that passes in `error` returning a `console.log` that logs out the `error` with prefix text `` `There was an error: ` ``:

```
..catch(error => console.log(`There was an error: ${error}`));
```

8. Save the file and make sure that your **json-server** from **QuickLab 8** is running in a terminal.
9. With **json-server** still running, execute this file. You should see the results logged. Try stopping **json-server** (by binning its terminal) and re-run this file. You should receive an error message.

Activity – Part 2 – POST Requests

To submit data to a service, a POST request can be made (for new data). POST requests should be sent to the same address as a GET request to retrieve all data.

1. Under the previous code, make a function called `sendData` that receives no parameters.
2. Declare a `const` called `resultToSend` with key/value pairs `home`, `away`, `homeScore` and `awayScore`, giving them values – we used `React Rangers`, `Vue United`, `4` and `1`.
3. Declare a variable called `addResult` that is a call to `fetch` with a **`url`** of `` `http://localhost:3000/results` `` and a *configuration object* that has the following key/value pairs:
    - `method` set to `POST`;
    - `body` set to `resultToSend` put through the `JSON.stringify` function;
    - `mode` set to `cors`;
    - `headers` set as shown:

```
"headers": {
  "Content-Type": "application/json"
}
```

4. Chain a call to `then` that has a callback arrow function that passes in `postResult` and returns `postResult` with a call to `json()`:
5. `.then(postResult => postResult.json())`
6. Chain another call to `then` that has a callback arrow function that passes in `postResult` and returns a `console.log` of it:

```
.then(postResult => console.log(postResult))
```

7. Chain a call to `catch` that has a callback arrow function that passes in `error` and *logs it out*:

```
.catch(error => console.log(`There was an error: ${error}));
```

8. Call the function `sendData()`.
9. Save the file and run it (ensuring that your json-server is still running).

You should see the `resultToSend` object logged back to you. Check your actual **`reactrangers.json`** file. It should have the new result saved to it. Try stopping json-server and check that an error is displayed.

This is the end of Quick Lab 10

Quick Lab 11 - Asynchronous JavaScript - async/await

Objectives

- To be able to use async/await to be able to send and receive data.

Activity – Part 1 – GET data

1. Point the terminal at **QuickLabs/ql11-asyncAwait/starter** and run the command `npm i`

This will install the dependencies for this project folder and (only) includes the node-fetch package. A **package.json** file can be found in the starter folder and this tells **npm** what to install. This should be committed as part of the repo. The resulting **node_modules** folder should not!

2. Create a new file in the **starter** folder called **asyncAwait.js**.
3. Set a `const` `fetch` to `require` `node-fetch`.
4. Declare a variable called `results`, set to an empty object.
5. Declare a `const` called `getReactRangerResults` that is an `async` arrow function that takes no parameters. The body of the function should:
    - Surround the following in a `try` block:
        - Declare a `const` called `reactRangersResultsData` set to `await` a call to `fetch` for `` `http://localhost:3000/results` ``;
        - Sets `results` to `await` `json()` being called on `reactRangersResultsData`.
        - Logs out `results`
    - Add a `catch` block that receives an `error` and logs it out.

```
const getReactRangersResults = async() => {
  try {
    const reactRangersResultsData =
      awaitfetch(`http://localhost:3000/results`);
    results = await reactRangersResultsData.json();
    console.log(results);
  } catch(error) {
      console.log(`There was an error: ${error}`);
  }
}
```

6. Call the `getReactRangersResults` function.

7. Save the file and run it, ensuring that you have your **json-server** running.

The console should display an *array of results objects*. Try stopping **json-server** to see the catch block execute.

Activity – Part 2 – POST data

1. Add a second `async` arrow function called `sendReactRangersResult` – it does not have any parameters – and a function block surround the following in a `try` block:
    - Declares a `const` called `resultToSend` with key/value pairs `home`, `away`, `homeScore` and `awayScore`, giving them values – we used `React Rangers`, `Vue United`, `4` and `1`.
    - Sets a `const addedResult` to `await` call to `fetch` with a **url** of `` `http://localhost:3000/results` `` and a *configuration object* that has the following key/value pairs:
        - `method` set to `POST`;
        - `body` set to `resultToSend` put through the `JSON.stringify` function;
        - `mode` set to `cors`;
        - `headers` set as shown:

```
"headers": {
  "Content-Type": "application/json"
}
```

    - Sets `result` to wait a call to `json()` on `addedResult`;
    - Logs out `result`
    - Has a `catch` block for an `error` to log it out.
2. Make a call to `sendReactRangersResult()`;
3. Save the file and run it, ensuring that **json-server** is running. Check that the *new result* is added to your **reactrangers.json** file. Stop **json-server** and check that an error is caught.

This is the end of Quick Lab 11

Quick Lab 12 - "Hello World" TypeScript

Objectives

- To be able to write and compile a simple TypeScript file and run the outputted JavaScript.

Activity

1. On the command line, using the cd command, navigate to the **QuickLabs/ql12-TypeScriptHelloWorld/starter** folder.
2. Install TypeScript, globally on your machine using the command:

```
npm i -g typescript
```

3. In the starter folder, create a new file called **helloWorld.ts**.
4. Add the following lines of code into it:

```
let world = `World`;
console.log(`Hello ${world}`);
```

5. Save the file.
6. Back on the command line, compile the TypeScript file to JavaScript using:

```
tsc helloWorld.ts
```

Notice that, if the compiler is able to run error free, it appears nothing has happened.

7. Check the starter folder, you should see a new file **helloWorld.js** - refresh the view if not.
8. Examine the contents of the file **helloWorld.js**.

Notice how all of the ES2015+ syntax has been replaced by older syntax? This is because part of the compilation process deals with the transpilation to ES3!
Run the file by typing the following command into the command line:

```
node helloWorld.js
```

9. Run the same command but use **helloWorld.ts**.

The file should run the same - this is because Node has found only JavaScript in the file.

10. Change the declaration of world to the following:

```
let world: string = `World`;
```

11. Run the compiler again and check the outputted JavaScript file - you should see that the type declaration has been removed.

12. Run both files with node again - the TypeScript file should now fail as it is not pure JavaScript.

This is the end of Quick Lab 12

Quick Lab 13 - TypeScript Dev Environment

Objectives

- To be able to create a project environment for TypeScript development.

Activity

1. On the command line, using the **cd** command, navigate to the **QuickLabs/13-TypeScriptDevEnvironment/starter** folder.
2. Initialise an npm project, accepting all defaults by using the command:

```
npm init -y
```

3. Install **typescript** and the **webpack plugin ts-loader** for the project using:

```
npm i --save-dev typescript ts-loader
```

4. Install Webpack, its CLI and the development server using:

```
npm i --save-dev webpack webpack-cli webpack-dev-server
```

5. Amend the **package.json** file scripts section so that the following replace the "test" script:

```
"build": "webpack --mode production",
"dev": "webpack --mode development",
"start": "webpack-dev-server --mode development --open",
"check-types": "tsc"
```

6. Configure webpack by creating the file **webpack.config.js** and putting the following code inside it:

```
module.exports = {
  entry: './src/index.ts',
  resolve: {
    extensions: ['.ts', '.js'],
  },
  module: {
    rules: [
      {
        test: /\.ts$/,
        use: {
          loader: "ts-loader"
        }
      }
    ]
  }
```

```
};
```

The entry part of this file tells Webpack how to get into our application. The resolve object tells Webpack to use both **.ts** and **.js** files imported. The module object tells Webpack to use the **ts-loader** when bundling **.ts** files.

When the build or dev commands are used, Webpack will create a bundled JS file called main.js (minified for build because of --mode production) and place it in a folder called dist. When using the development server, a virtual file is created and held on it. The check-types script will simply run the compiler and highlight any TypeScript errors.

7. Next, configure TypeScript by creating a **tsconfig.json** file and putting the following inside it:

```
{
  "complierOptions": {
    "sourceMap": true,
    "target": "es5"
  }
}
```

This config enables TypeScript and Webpack to allow you to debug the TypeScript files in the browser (use the Sources tab and then **webpack:// → . → src** and then debug from the **.ts** file). It also means that the outputted JavaScript is only ES5 compliant. If you need to support ES3 browsers, change this and look into Polyfills!!!

8. Create the entry file, **index.ts** in a new folder called **src**.

9. Re-write or copy the contents from QuickLab 12's TypeScript file in **index.ts** and save.

10. Create a new file in the **index.html** folder called **index.html**.

11. Add a skeleton HTML page that as a script tag with a source of **main.js**.

This file **src** will need to be modified when going into a production environment. Presently, it enables the development server to run the bundled file.

12. On the command line, run the project using:

```
npm start
```

Your browser should spin up now - check the console and also find the file to debug.

13. Experiment with the build and dev commands (insert **run** after the npm command if it doesn't work out of the box).

Run the check-types script and see the results - i.e. the new files in the src folder!

This is the end of Quick Lab 13

Quick Lab 14 - TypeScript Tuples

Objectives

- To experiment with TypeScript's Tuples.

Activity

1. In **VSCode**, open the file **tuple.ts** from the **QuickLabs/ql-14TypeScriptTuples/starter** folder.
2. Define a *tuple* called `person` that has 3 types allowed: `string`, `number` and `boolean`.
3. Try to define the `person` with *values in the wrong order* - note the errors that are given by **VSCode**.
4. Log out `person`.
5. Use the TypeScript compiler to compile the file: `tsc tuples.ts` - ignore the warnings.
6. Run the compiled JavaScript file using: `node tuples.js` and verify that the JavaScript has been executed
7. Define a `person` correctly and check the logging.
8. Try to add another element to the array and note the errors:

```
person.push(() => console.log(`Hello World`);
```

9. Call for person[3] to be executed as a function:

```
person[3]()
```

10. Check the output of logging.

REMEMBER THE ONLY LESSON YOU NEED ABOUT USING TYPESCRIPT!

This is the end of Quick Lab 14

Quick Lab 15 - Type Assertion and Unknown

Objectives

- To use the unknown and type assertion when working with variables.

Activity

1. In **VSCode**, open the file **typeAssertion.ts** from the **QuickLabs/15-TypeScriptTypeAssertion/starter** folder.
2. Define a variable called **something** with a type of **unknown** and initially set it to the **string 1234**.
3. Log out the result of the Boolean expression `something == 1234`.
4. Log out the result of the Boolean expression `something === 1234`.
5. Log out the result of the Boolean expression `something !== 1234`.
6. Log out the result of the Boolean expression `something >= 1000`.
7. Log out the result of asking for the `length` of `something`.
8. Note the error here - check the output of the compiled JavaScript by saving and compiling the TypeScript and running the JavaScript through node!
9. Add the `as number` type assertion to all of the expressions above.
10. Save the file and compile it.
11. Run the JavaScript file in node.

What do you notice about the output of `something as number === 1234`? Can you explain this?

This is the end of Quick Lab 15

Quick Lab 16 - TypeScript Classes

Objectives

- To experiment with classes and access modifiers

Activity

1. In **VSCode**, open the file **index.ts** from the **QuickLabs/ql16-TypeScriptClasses/starter/src** folder.
2. Run **npm install** on the command line and start the application running.
3. Declare a `class` called `Vehicle` and set it to have `private` properties of `make` and `model`, set as `strings` and a `number` `private` property of `speed` set initially to `0`.
4. Provide `get` methods for `make`, `model` and `speed`.
5. Provide a `set` method for `speed` that:
6. Takes a parameter `delta` of type `number` to represent the change in speed;
7. Checks to see if the new speed (by applying the change speed to the current speed) is *greater than* `0` - if it is set the new speed to the *calculated value*, otherwise set speed as `0`.
8. Make an *instance* of a `Vehicle` and ensure that the *methods* and *modifiers* work as expected.
9. `Extend` the `Vehicle` with a class called `RoadVehicle` that has its own `private` property of `wheels` (it should be a `number`).
10. Provide a `getter` for `wheels`.
11. Create an *instance* of a `RoadVehicle` and check that all of its properties can be accessed in the expected way.
12. Make the `Vehicle` class and the `get` and `set` for `speed` `abstract`.

Notice that you are no longer allowed to make an instance of the `Vehicle` class.

Notice that getters and setters can be `abstract` - but only as a pair.

Notice that the implementation of the `RoadVehicle` class is now incorrect.

13. Fix the `RoadVehicle` class by providing the concrete implementations of the `get` and `set` methods for `speed`. You may notice that `_speed` is *not accessible - without* making it `public`, what should you do?
14. Check that your new implementation works.

This is the end of Quick Lab 16

Quick Lab 17 - TypeScript Interfaces

Objectives

- To use an Interface with multiple classes

Activity

1. In **VSCode**, open the file **index.ts** from the **QuickLabs/ql-17/TypeScriptInterfaces/starter/src** folder.
2. Under the provided code, create an `interface` called `HasPassengers`, it should specify:
    - A `readonly` property of `passengerSeats`;
    - A method called `makeStop` that takes *2 numeric arguments* of `numberOn` and `numberOff` and specifies that it *does not return anything*.
3. Create a `class` called `SingleDeckerBus` that `extends` the `RoadVehicle` class and `implements` the `HasPassengers` interface:
    - The constructor should accept *all parameters* needed for the `RoadVehicle` class and the `HasPassengers` interface, along with a `private` property `passengersOnBoard`, initially set to `0`.
4. In the class, implement a `getter` for `passengersOnBoard` and the required `makeStop` method.
5. The `makeStop` method can be as simple as you like - we implemented it so that the bus never has a negative number of passengers or more passengers than there are seats!
6. Create an *instance* of `SingleDeckerBus` and check that the methods work and that the properties are as expected.
7. If you feel the need…
8. Create a `class` called `Train` that `extends` `Vehicle` and `implements` `HasPassengers`, adding any properties or methods that are needed to make the class function.

This is the end of Quick Lab 17

Quick Lab 18 - Modules

Objectives

- To use modules with TypeScript files

Activity

1. Split each of the classes into their own file. Remember to add `export default` to the start of the class declaration and to import any Classes or Interfaces that will now be in other files.
2. Leave only the code to execute (along with the appropriate imports) in the index.ts file.
3. When you have done, save all files, run and `npm install` and then `npm start`. You should have no errors when you inspect the console and there should be the lines of logs executed.

This is the end of Quick Lab 18