# NM2207

Session 09 Codealong

# Overview of what we will do today:

* Javascript basics
* The fetch API
* Importing data

**Warm up**

* forEach

The forEach() method is a useful way to loop through an array and perform some operation on each element. It takes a callback function as its argument, which is executed for each element in the array. The callback function takes up to three arguments: the current element being processed, the index of the current element, and the array itself. However, the second and third arguments are optional and can be omitted if not needed.

We can use the forEach() method to loop through an array. The callback function passed to forEach() takes one argument, which is the current element being processed in the array.

We can also use the forEach() method to loop through the array and perform some operation on each element. For instance, we can push it to a new array.

* split and slice

The split() method is a useful way to split a string into an array of substrings based on a delimiter. The slice() method is a useful way to get a portion of an array without modifying the original array. These methods can be combined with other array methods like map(), filter(), and reduce() to perform more complex operations on strings and arrays.

We can use the split() method to split a sentence into an array of words. The split() method takes a delimiter as its argument, which in this case is a space character " ". The result is an array of words.

Next, we can use the slice() method to get a portion of the array. The slice() method takes two arguments: the starting index (inclusive) and the ending index (exclusive). So for example, we can get the first three words of the array using words.slice(0, 3). The result is an array containing the first three words of the sentence.

* push

The push() method is a useful way to add new elements to the end of an array. It can be used with a single element or multiple elements, and it modifies the original array. If you want to add elements to the beginning of an array, you can use the unshift() method instead.

We can use the push() method to add an element to the end of the array. The argument passed to push() is the element we want to add to the end of the array. The result is an array with the new element added to the end.

We can also use the push() method again to add multiple elements to the end of the array. We pass multiple arguments to push() separated by commas. The result is an array with the new elements added to the end.

* const

Using const for variables that won't change improves the readability and maintainability of the code. It also helps prevent unintended errors that can occur when a variable is accidentally reassigned.

In general, it is a best practice to use const for variables that won't change and use let for variables whose values will change during the program execution. If you need to reassign a value to a variable declared with const, you will get a TypeError in strict mode.

**Part 1**

**Using an API**

We’re back to hard-core JS coding this week. Remember our bar chart on deepfake sharing from Week 7? We had to manually type into the array ourselves. In this part, you will learn how to read and visualize data from an online file, where the data is already made available to you.

**Using the Fetch API**

In order to grab data from online files, we will have to use the Fetch API, which is a JavaScript tool that allows us to fetch resources from the Internet.

This is how your fetch function will look like for this session.

Text

Description automatically generated

There’s a lot of things going on here, so let’s break it down line-by-line.

**Lines 1-7**

Line 1, we are creating a variable called data and assigning it the result of the fetch function. Now, what does the fetch function do? The fetch function needs 2 arguments, *resource*, and *options*. *Resource* refers to the string URL of whatever you’re fetching from. In this case, it’s a CSV file somewhere on the Internet. *Options* refers to an object, containing information about the resource that you need to tell the place you’re fetching from. This argument is **optional**. Because we’re fetching a simply csv file, nothing more complicated than that, we only pass 1 argument – the resource.

Line 2 is therefore our resource, the URL of our .csv file:

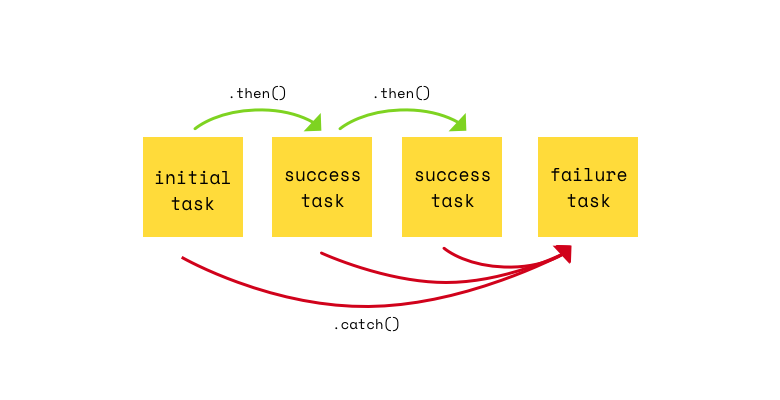
<https://2207-resources.s3.ap-southeast-1.amazonaws.com/sharing_deepfakes.csv>

Line 3 is where the fetch function ends.

Our fetch function always returns what we call a *Promise*. Essentially, that means JS is looking for your file, and promises that it will get back to you whether the file can be found. That’s when your promise is “resolved”. When that happens, you can use the .then() method to then grab the data from the promise. Line 4 is therefore an extension of our fetch function.

The then() method is a useful way to handle the resolved value of a promise. It allows us to perform some operation on the resolved value and chain multiple operations together. If you need to handle errors that may occur during the execution of the promise, you can use the catch() method to handle them. The then() method takes a callback function as its argument, which is executed when the promise is resolved. It is accepting another nameless function as an argument. This nameless function has one argument, *response*, which is **passed down** from the result of the previous fetch function (which is the csv file we asked for). We then convert the response to readable **string** using .text(), and return the result in Line 5.

Well, the thing is, .text() returns another *Promise*. And when you have a *Promise*, you need another .then() method to grab the data resolved by the promise. This is called promise chaining.



Line 7 is another extension. The nameless function in the .then takes in an argument called *data* that is **passed down** from our previous result. Try doing console.log(data) after Line 7. What do you see?

Now that we have the proper data, we have to manipulate it so that we can feed it to Chart.js for visualization.

**Part 2 – Programmatically Splitting Data**

Text

Description automatically generated

**Lines 8-15**

The magic happens here. For now, we’ll create an empty array, ***table***, in line 8, that will eventually become a two-dimensional array, similar to an excel file with rows and columns.

The thing about .csv files, is that it returns all your column data separated by commas. And rows are separated by line-breaks (which are represented by invisible characters \r\n).

So, the first part of our .csv would look something like this:

Country,USA,China,Singapore,Indonesia,Malaysia,Philippines,Thailand,Vietnam\r\n

Age,0.05,0.184,0.136,0.074,0.18,0.078,0.162,0.106\r\n

If you want to split this into its respective rows, you have to do **string-splitting**. This can be accomplished with the .split() function, which is what is happening in Line 9. We are splitting the entire CSV string into its respective rows, by identifying the parts where “\r\n” appears.

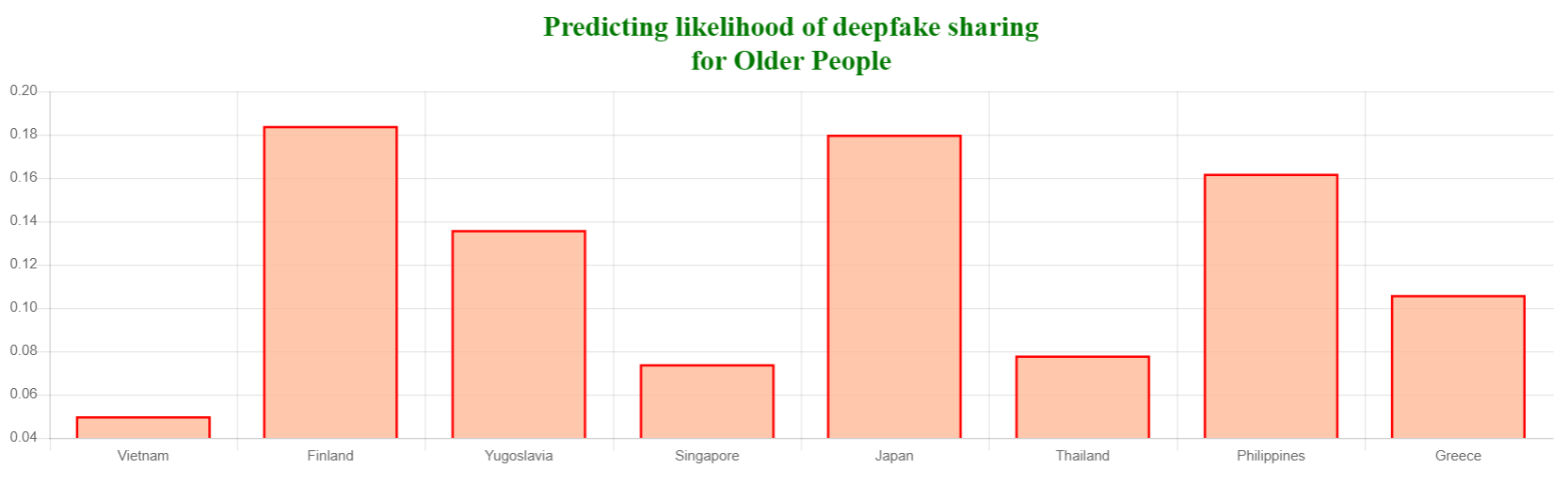
Now that you have your rows, you want to split them into their respective columns, so each piece of data is its own cell. Looping over each row in Line 11, we start splitting each row in Line 12 into an array of values by cutting the data where a comma appears. Then we push our new row of values into the table array (Line 13).

Try looking at what the console prints in Line 15, you should see that ***table*** is now an array of 13 different arrays (rows). Each array should have 9 values (columns). The first value is the title of the entire row (e.g., ‘Country’, ‘Age’, ‘Education’).

**Part 3 – Visualizing Data**

Now let’s prepare the data for visualization into Chart.js. This is what your code should look like.

For now, we’re interested in the first 2 rows of our data. If you look at the table printed in Part 2, the first row represents the list of countries, while the second row represents the probability of deepfake sharing among older people. We want to print a bar chart that looks like this:



In this case, the *labels* of the chart are the list of countries, and the *values* (or *dataset*) is the list of probabilities.

But remember how each row has its title as its first value? Like “country” or “age”? We don’t need that for our bar chart. So to remove the first value of an array in JavaScript, we can use the [.slice() function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice). The first argument for the .slice() function tells JS where to cut the array. We want to cut the array after the first index (to remove the title), so we give it an argument of 1.

Therefore, our list of countries (without the title ‘Country’) would be table[0].slice(1).

You can now do the same thing for the ‘Age’ array and remove its title.

Both these rows are then assigned to variables labelCountry and beingOld respectively in Lines 17 and 18.

Text

Description automatically generated

If you remember from Week 7, we need to give Chart.js a standard object that represents our data in order to visualize it. For a bar chart, we need to give it a list of labels, and provide it with a dataset corresponding to the list of values to print for each label. You can now fill in the dataObj accordingly (lines 19-30).