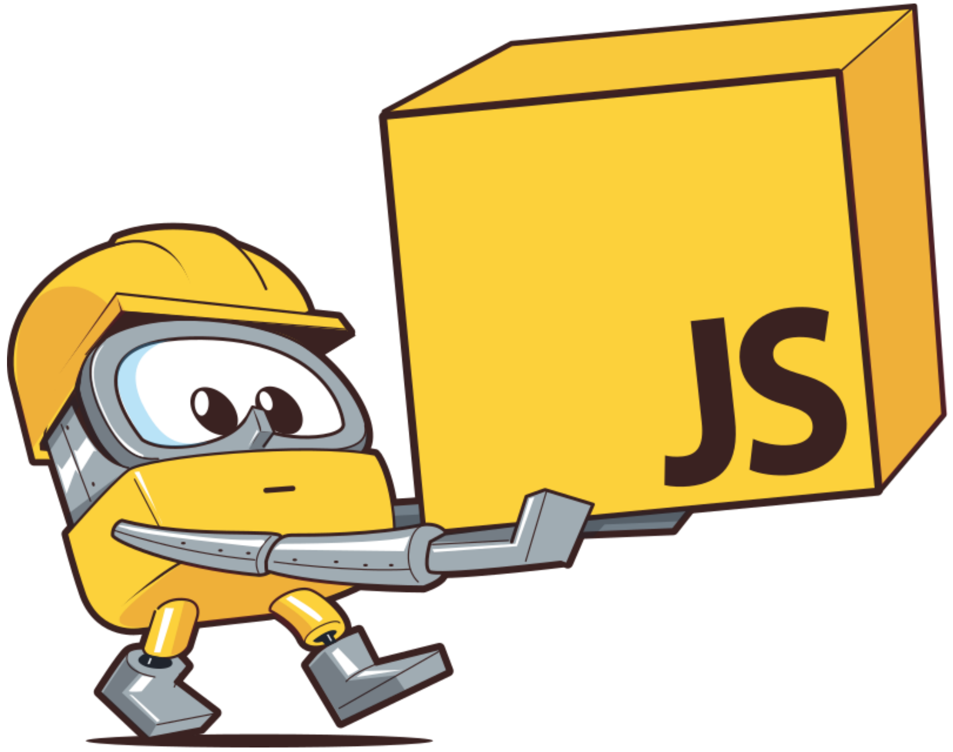
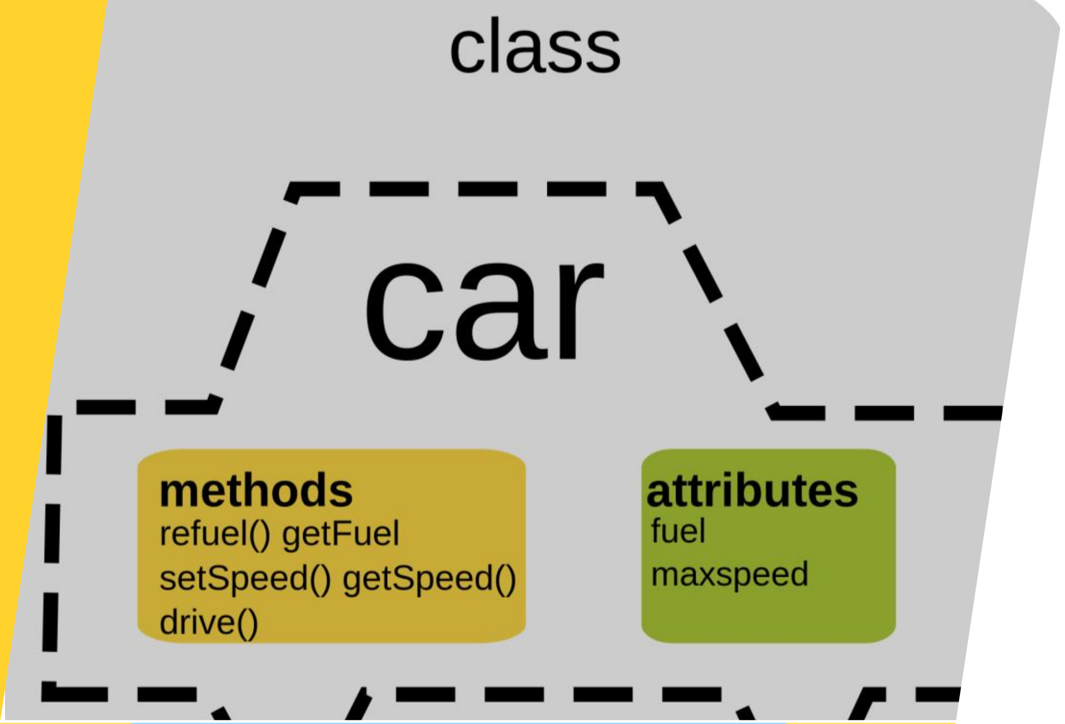
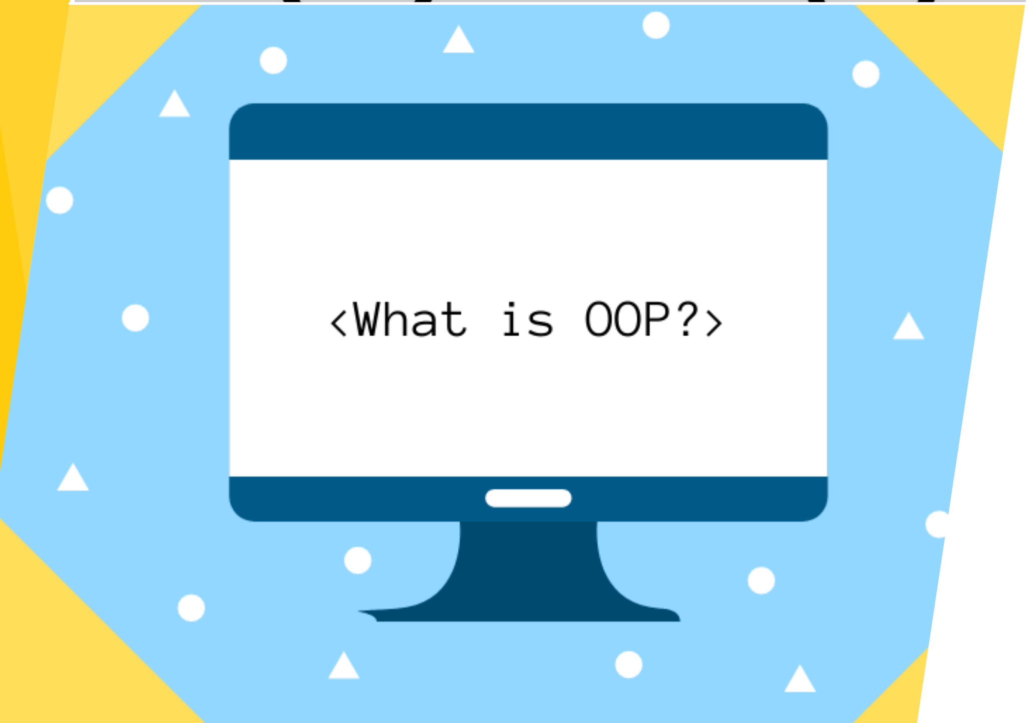


Advanced JS – Part 2





Object Oriented Programing

Focus points:

- JSON
- Asynchronous Code
- Working With APIs
- Async And Await
- Recursive Methods
- Testing

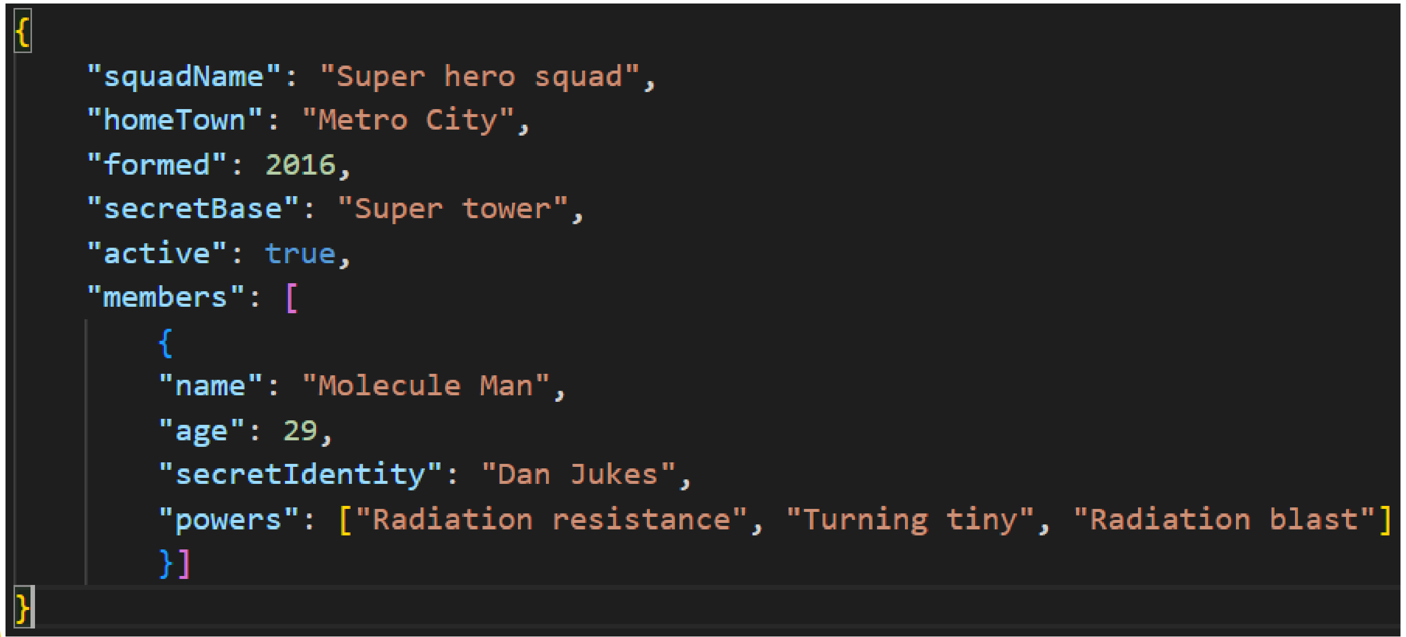
Ex. # 1.0

JSON

JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax. It is commonly used for transmitting data in web applications (e.g., sending some data from the server to the client, so it can be displayed on a web page, or vice versa).

Even though it closely resembles JavaScript object literal syntax, it can be used independently from JavaScript, and many programming environments feature the ability to **read (parse)** and **generate JSON**.

JSON exists as a string — useful when you want to transmit data across a network. It needs to be converted to a native JavaScript object when you want to access the data. This is not a big issue — JavaScript provides a global JSON object that has methods available for converting between the two.



```
{
    "squadName": "Super hero squad",
    "homeTown": "Metro City",
    "formed": 2016,
    "secretBase": "Super tower",
    "active": true,
    "members": [
        {
        "name": "Molecule Man",
        "age": 29,
        "secretIdentity": "Dan Jukes",
        "powers": ["Radiation resistance", "Turning tiny", "Radiation blast"]
        }]
}
```

Ex. # 1.0

JSON

Converting between objects and text - The **Ex. #1.0** was simple in terms of accessing the JavaScript object, because we converted the network response directly into a JavaScript object using **response.json().**

But sometimes we aren't so lucky — sometimes we receive a raw JSON string, and we need to convert it to an object ourselves. And when we want to send a JavaScript object across the network, we need to convert it to JSON (a string) before sending it. Luckily, these two problems are so common in web development that a built-in JSON object is available in browsers, which contains the following two methods:

- parse(): Accepts a JSON string as a parameter, and returns the corresponding JavaScript object.
- stringify(): Accepts an object as a parameter, and returns the equivalent JSON string.

Note: Converting a string to a native object is called **deserialization**, while converting a native object to a string so it can be transmitted across the network is called **serialization**. A **JSON** string can be stored in its own file, which is basically just a text file with an extension of **.json.**

JSON

Things to know about JSON:

- **JSON** is purely a string with a specified data format — it contains only properties, **no methods**.
- **JSON** requires double quotes to be used around strings and property names. Single quotes are not valid other than surrounding the entire JSON string.
- Even a single misplaced comma or colon can cause a **JSON** file to go wrong, and not work. You should be careful to validate any data you are attempting to use (although computer-generated **JSON** is less likely to include errors, as long as the generator program is working correctly). You can validate **JSON** using an application like JSONLint.
- **JSON** can actually take the form of any data type that is valid for inclusion inside **JSON,** not just arrays or objects. So for example, a single string or number would be valid **JSON**.
- Unlike in JavaScript code in which object properties may be unquoted, in JSON only quoted strings may be used as properties.

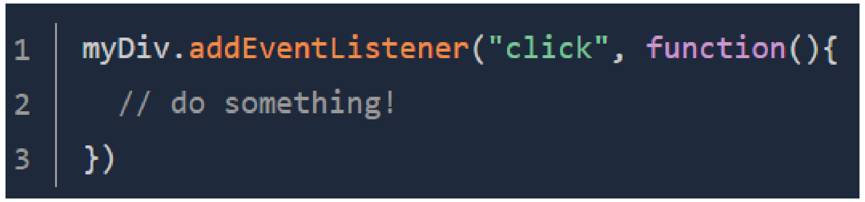
Note: Since **JSON** will most likely be used in JS in the object form, you can refresh your knowledge of loops.

Asynchronous Code

Since JavaScript is the language of the web, there are some functions that by necessity are going to take a decent amount of time to complete, such as fetching data from a server to display on your site. For this reason, JavaScript includes support for **asynchronous functions**, or to put it another way**, functions that can happen in the background while the rest of your code executes.**

Callbacks - In the recent past, the way that these were most commonly handled were with **callbacks**, and even now they are still used quite a lot in certain circumstances. **A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.**

Callbacks are simply functions that get passed into other functions. For example:



```
myDiv.addEventListener("click", function(){
  // do something!
})
```

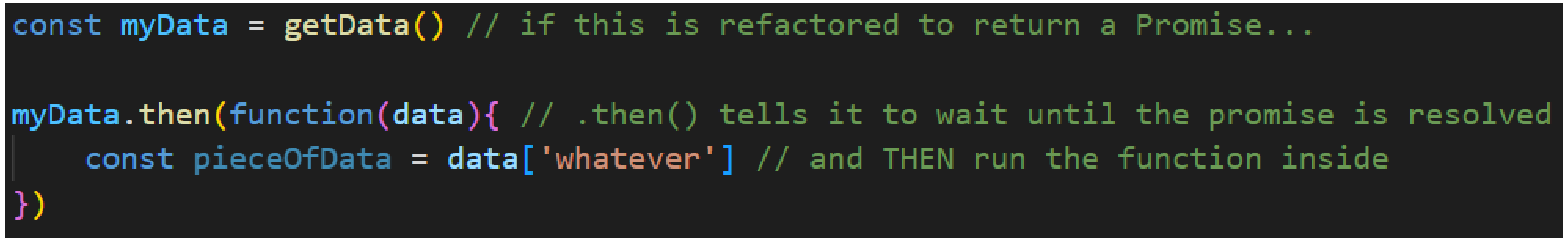
You will likely recognize this pattern as something that happens all the time in JavaScript code. Unfortunately, though they are useful in situations like the above example, using **callbacks** can get out of hand, especially when you need to chain several of them together in a specific order. The rest of this lesson discusses patterns and functions that will help keep you out of **Callback hell!**

Ex. #
2.0-1

Asynchronous Code

Promises - There are multiple ways that you can handle asynchronous code in JavaScript, and they all have their use cases. **Promises** are one such mechanism, and they're one you will see somewhat often when using other libraries or frameworks. Knowing what they are and how to use them is quite useful.

Essentially, a **promise is an object that might produce a value at some point in the future**. Here's an example:



```
const myData = getData() // if this is refactored to return a Promise...

myData.then(function(data){ // .then() tells it to wait until the promise is resolved
    const pieceOfData = data['whatever'] // and THEN run the function inside
})
```

While synchronous code is easier to follow and debug, async is generally better for performance and flexibility. Why "hold up the show" when you can trigger numerous requests at once and then handle them when each is ready?

Promises are becoming a big part of the JavaScript world, with many new APIs being implemented with the promise philosophy.

Ex. # 3.0

API

API - Servers that are created for serving data for external use (in websites or apps) are often referred to as **APIs** or **'Application Programming Interfaces'.**

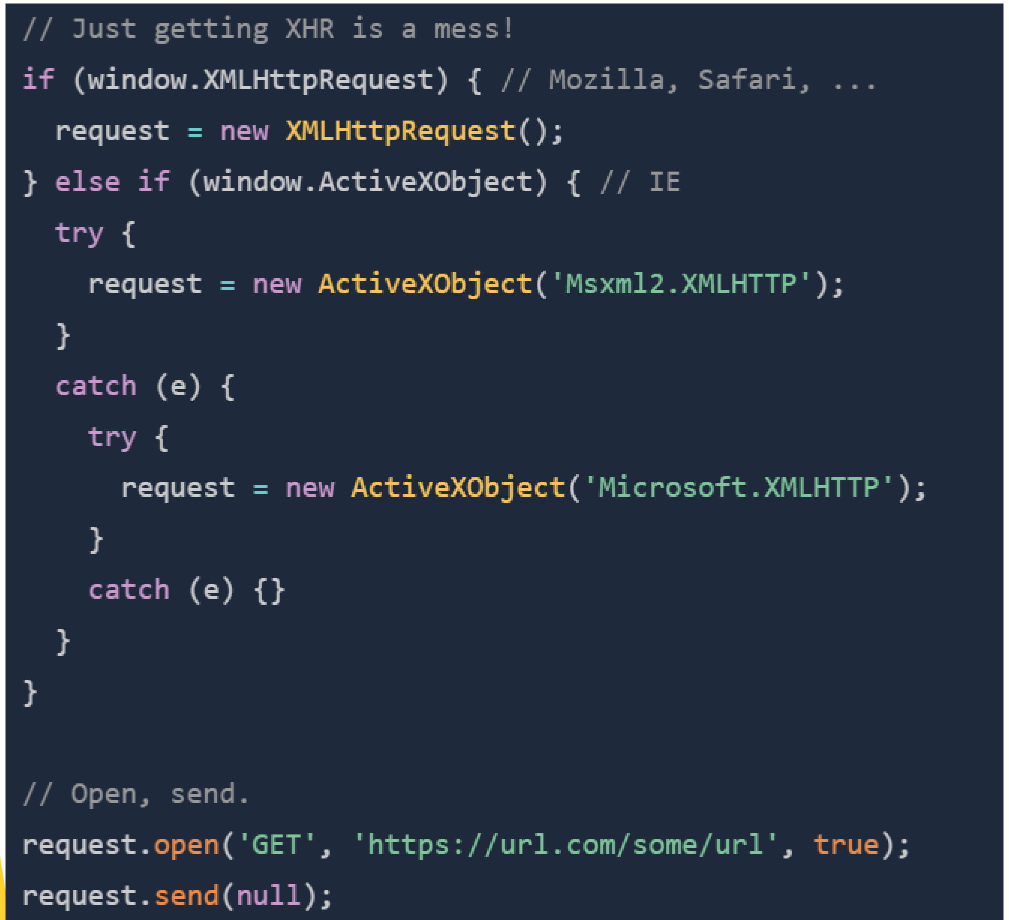
There are multiple ways of requesting data from an **API**, but all of them basically do the same thing. For the most part, **APIs** are accessed through **URLs**, and the specifics of how to query these **URLs** change based on the specific service you are using. For example, the OpenWeatherMap API has several types of data that you can request. The specifics for using any API are usually documented on the service's website.

In most cases, you will have to create an account and request an "**API key**" from the **API service** before attempting to fetch data from their endpoints. Once obtained, an **API key** will usually have to be included with every data request. As you can imagine, an API key is random and unique to you. As such, services like **OpenWeatherMap** can correlate your **API key** to your requests of their data, including how much and how often you are requesting it.

On one hand, issuing API keys allows an API service to better track abuse of their systems and data. On the other hand, it can also be a way for those services to mitigate and recuperate operating costs. While a single request to an API might cost a fraction of a penny, imagine using that API to create an amazing weather app that gets used all over the world… you could easily have thousands of people accessing that data every minute! The cost to handle that traffic could quickly balloon up to significant sums for the API service.

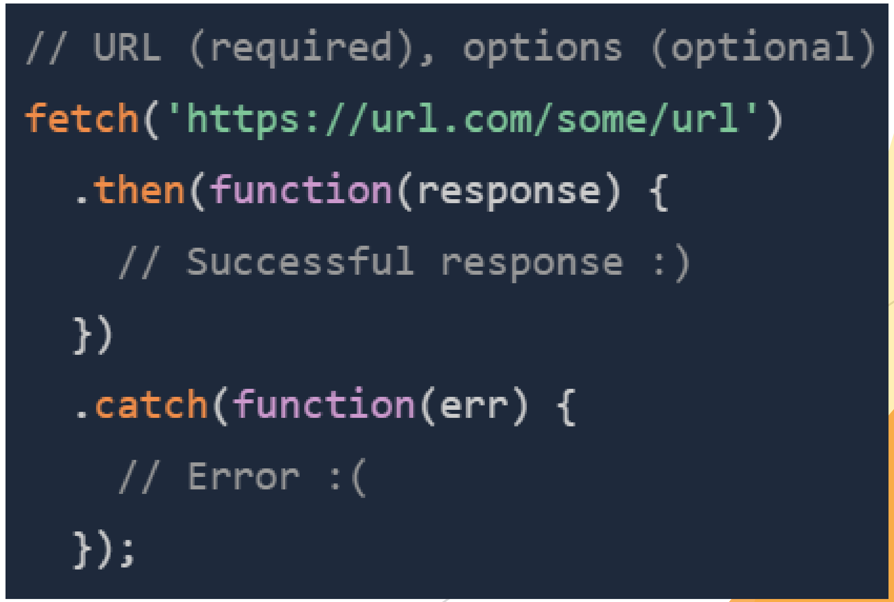
API

Fetching Data - A couple of years ago the main way to access API data in your code was using an **XMLHttpRequest**. This function still works in all browsers, but unfortunately, it is not particularly nice to use. The syntax looks something like this:



```
// Just getting XHR is a mess!
if (window.XMLHttpRequest) { // Mozilla, Safari, ...
  request = new XMLHttpRequest();
} else if (window.ActiveXObject) { // IE
  try {
    request = new ActiveXObject('Msxml2.XMLHTTP');
  }
  catch (e) {
    try {
      request = new ActiveXObject('Microsoft.XMLHTTP');
    }
    catch (e) {}
  }
}

// Open, send.
request.open('GET', 'https://url.com/some/url', true);
request.send(null);
```

More recently, however, web browsers have begun to implement a new native function for making **HTTP requests**, and that's the one we're going to use and stick with for now. Meet **fetch**:

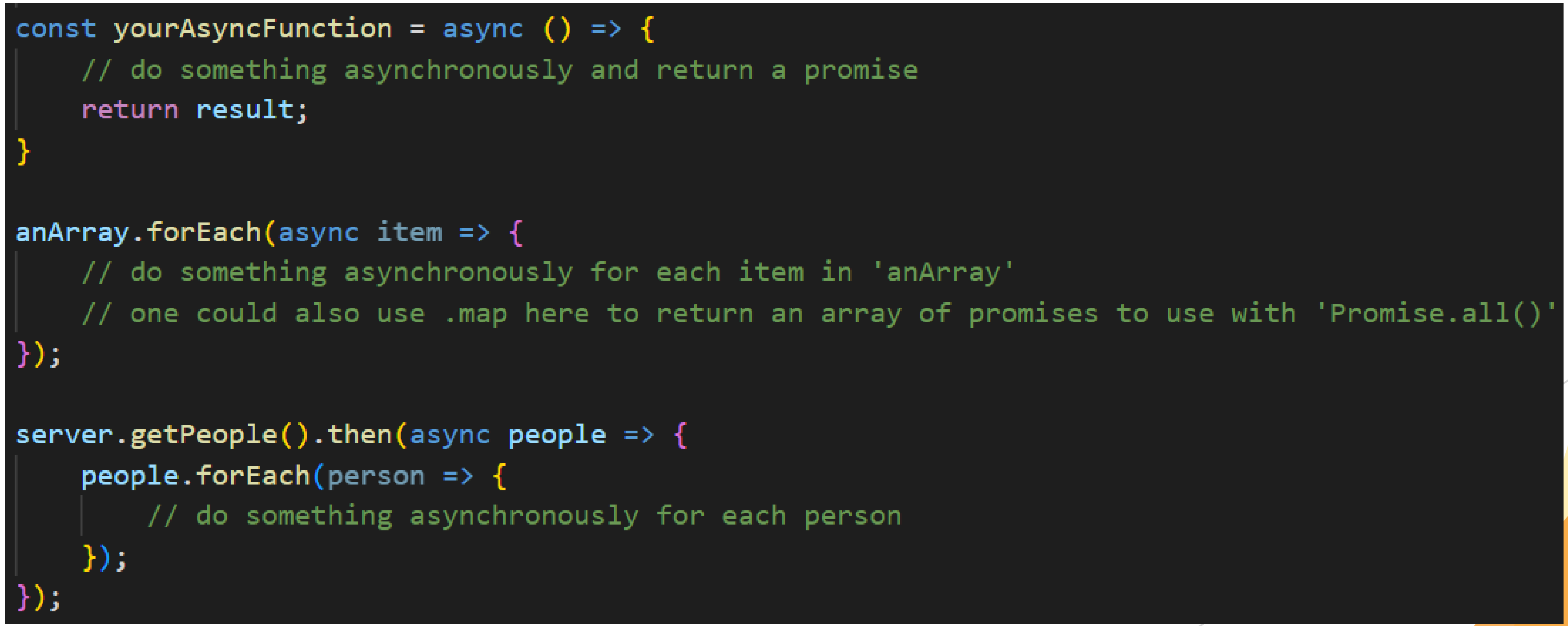


```
// URL (required), options (optional)
fetch('https://url.com/some/url')
  .then(function(response) {
    // Successful response :)
  })
  .catch(function(err) {
    // Error :(
  });
```

Async And Await

The async keyword - is what lets the JavaScript engine know that you are declaring an **asynchronous function**. This is required to use **await** inside any function. When a function is declared with async, it automatically returns a promise; returning in an async function is the same as resolving a promise. Likewise, throwing an error will reject the promise. An important thing to understand is **async functions are just syntactical sugar for promises**.

The **async keyword** can also be used with any of the ways a function can be created. It is valid to use an **async function** anywhere you can use a normal function.



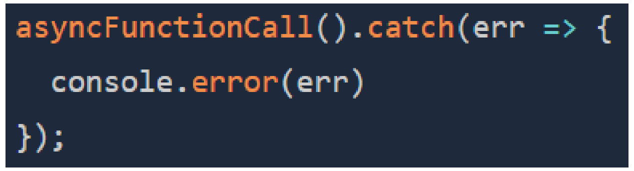
```
const yourAsyncFunction = async () => {
    // do something asynchronously and return a promise
    return result;
}

anArray.forEach(async item => {
    // do something asynchronously for each item in 'anArray'
    // one could also use .map here to return an array of promises to use with 'Promise.all()'
});

server.getPeople().then(async people => {
    people.forEach(person => {
        // do something asynchronously for each person
    });
});
```

Ex. # 4.0

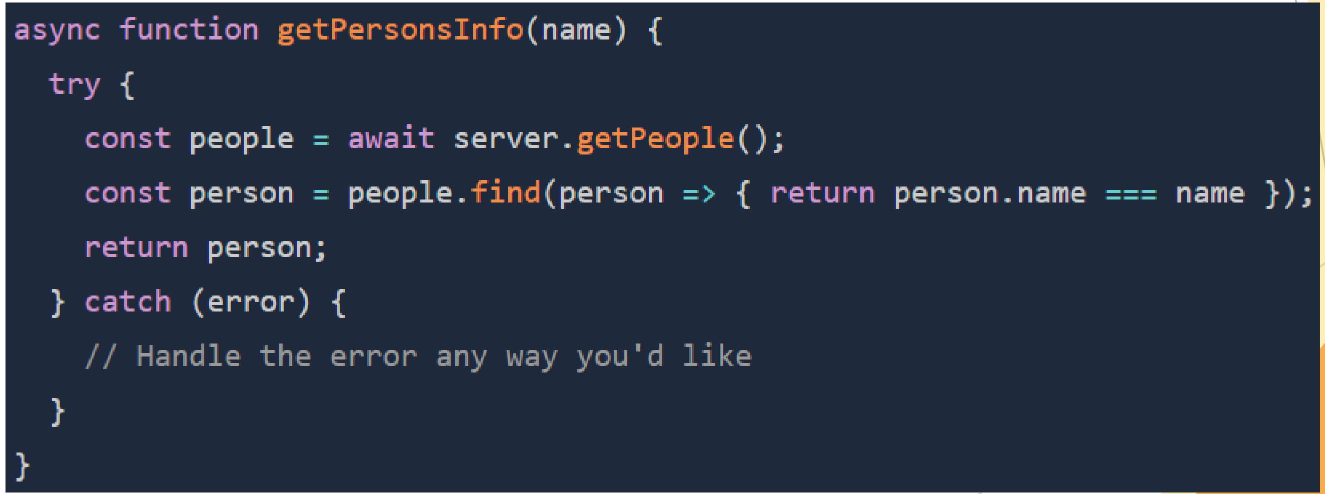
Async And Await

The await keyword - await is pretty simple: it tells JavaScript to wait for an **asynchronous action** to finish before continuing the function. **It's like a 'pause until done' keyword**. The **await keyword** is used to get a value from a function where you would normally use **.then()**. Instead of calling .**then()** after the **asynchronous function**, you would simply assign a variable to the result using **await**. Then you can use the result in your code as you would in your synchronous code.

Error Handling - Handling errors in async functions is very easy. Promises have the **.catch() method** for handling rejected promises, and since async functions just return a promise, you can simply call the function, and append a **.catch()** method to the end.



```
asyncFunctionCall().catch(err => {
  console.error(err)
});
```



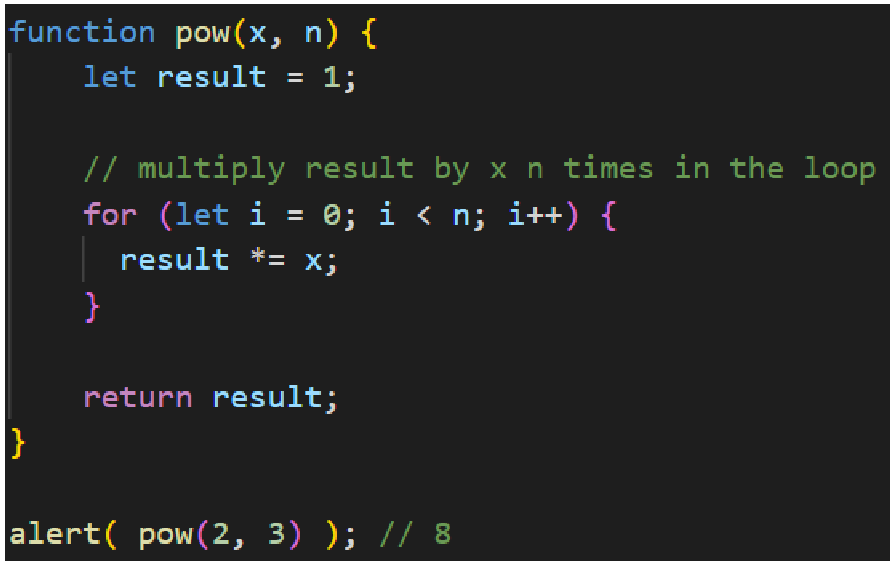
```
async function getPersonsInfo(name) {
  try {
    const people = await server.getPeople();
    const person = people.find(person => { return person.name === name });
    return person;
  } catch (error) {
    // Handle the error any way you'd like
  }
}
```

There is another way: **try/catch block!** If you want to handle the error directly inside the async function, you can use **try/catch** just like you would inside synchronous code. Doing this can look messy, but it is a very easy way to handle errors without appending **.catch()** after your function calls.

Recursive Methods

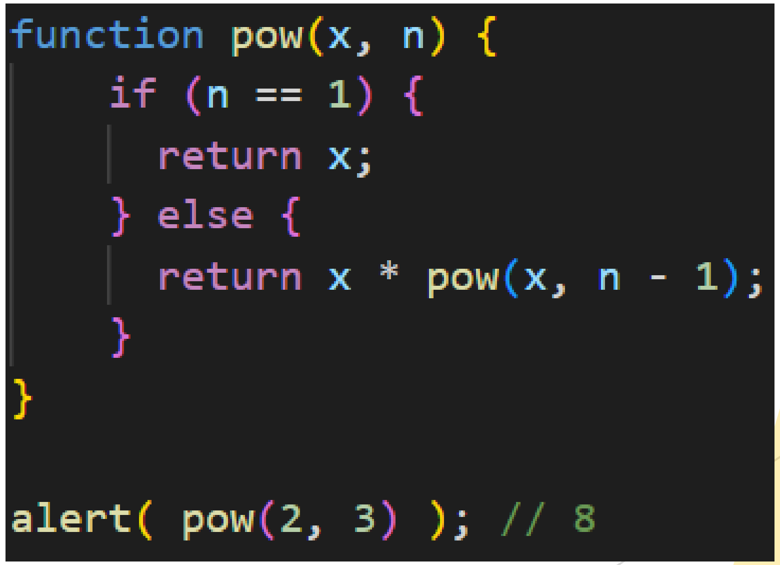
Recursion - is a programming pattern that is useful in situations when a task can be naturally split into several tasks of the same kind, but simpler. Or when a task can be simplified into an easy action plus a simpler variant of the same task. When a function solves a task, in the process it can call many other functions. A partial case of this is when a function calls itself. That's called **recursion**. There are **TWO** ways you can approach recursion:

1. Iterative thinking: the for loop:



```
function pow(x, n) {
    let result = 1;

    // multiply result by x n times in the loop
    for (let i = 0; i < n; i++) {
      result *= x;
    }

    return result;
}

alert( pow(2, 3) ); // 8
```

2. Recursive thinking: simplify the task and call self:



```
function pow(x, n) {
    if (n == 1) {
      return x;
    } else {
      return x * pow(x, n - 1);
    }
}

alert( pow(2, 3) ); // 8
```

Note: Also, check out these two videos on recursion: VIDEO 1 and VIDEO 2

Testing

Test Driven Development, or **TDD** for short, is a big deal in the modern development landscape. The main idea is simply that you start working on your code by writing automated tests before writing the code that is being tested.

There are many test-running systems available in JavaScript: **Mocha**, **Jasmine**, **Tape** and **Jest** to name a few. Fortunately, the syntax for each one is very similar. They all have their own set of special features, but the basic syntax is almost identical, so in the end it doesn't matter which one you use.

For this course, we will use **Jest.**

To set it up properly follow these instructions.

After you do the setup, you need to be aware of the following logic, **TTD** is teaching you to first write your tests before you start writing your code, I know, it's strange. **O.o** But don't worry, there is logic behind it! So, lets try to break down a test and to figure out the logic behind it.

Testing
folder

Testing

The way you write JS code will not change to much, now let us imagine we want to build an JS app, let's go trough the steps:

- First initialize **Jest** in your repository, like you were thought by the **Jest** documentation;
- Now before you start creating JS files, one thing that is worth mentioning is that you will need to create **modules,** meaning JS files that are going to export functions. You need that in so you can import them in your test JS files;
- Here the naming comes to play, you can name your test JS files however you want, but thy need to end with **.test.js (example.test.js)** It is advisable to use the same name for your JS files and your test files **(example.js/example.test.js);**
- **Jest** uses "matchers" to let you test values in different ways. Basic tests use the **test function** that takes in two arguments, **first** one is the string value used to describe the test, **second** one is usually a function called expect where you write the piece of code which should represent your "**expected result**" from the function you are writing;

Note: To use the **testing folder,** you need to go into it and initialize **Jest** over your terminal with a command: **npm install --save-dev jest,** if you are using **NPM** that is, for **Yarn** users type **yarn add --dev jest**

Ex. #
5.0-1

Testing

An important basic concept in testing is isolation. You should only test one method at a time, and your tests for one function should not depend upon an external function behaving correctly - especially if that function is being tested elsewhere. The main reason for this is that when your tests fail, you want to be able to narrow down the cause of this failure as quickly as possible. If you have a test that depends on several functions, it can be hard to tell exactly what is going wrong.

Pure Functions - There are many benefits to using TDD when you write your code. One of the biggest benefits is less obvious at first - it helps you to write better code. Tightly coupled code is hard to test! Imagine trying to write tests for a function like this **Ex. # 5.0**

Making this testable requires us to split up all the different things that are happening. First, we do not need to test the functions prompt and alert because they are built in to the browser. They are external to our program and whoever wrote them has already tested them. What we do need to test is the number logic, which is much easier if we untangle it from the other functions **Ex. #5.1**

The only thing we really need to test is the **evaluateGuess** function, which is much easier to test because it has a clear input and output and doesn't call any external functions. This implementation is much nicer as well because it's much easier to extend. If we wanted to switch out the prompt and alerts for methods that manipulate the DOM we can do that more simply now and if we want to make our code more advanced by letting the user make multiple guesses, that is also easier.

Testing

Mocking - There are two solutions to the ‘tightly coupled code’ problem.

The **first**, and best option is to simply remove those dependencies from your code as was done in the slide before, but that is simply not always possible.

The **second** option is mocking - writing “fake” versions of a function that always behaves exactly how you want. For example, if you’re testing a function that gets information from a DOM input, you really don’t want to have to set up a webpage and dynamically insert something into the input just to run your tests. With a mock function, you could just create a fake version of the input-grabbing function that always returns a specific value and use THAT in your test.

Note: Some test will require you to do some setting up before you actually stary testing, so in order to do that, we need to do a step call Setup and Teardown**.** Following this link will help you better understand what that means and how you can do that on your more complex applications.

Assignment

branko.fd.br@gmail.com

Thank you for your attention!