GENERAL ASSEMBLY

# JAVASCRIPT DEVELOPMENT

Sasha Vodnik, Instructor

# HELLO!

1. Pull changes from the `svodnik/JS-SF-13-resources` repo to your computer
2. Open the `09-ajax-apis/starter-code` folder in your code editor

# AJAX & APIS

# LEARNING OBJECTIVES

At the end of this class, you will be able to

- Access public APIs and get information back.
- Implement an Ajax request with vanilla JS.
- Create an Ajax request using jQuery.
- Describe what asynchronous means in relation to JavaScript
- Pass functions as arguments to functions that expect them.
- Write functions that take other functions as arguments.
- Build asynchronous program flow using promises and Fetch

# AGENDA

- Ajax using Fetch
- Ajax & jQuery
- Separation of concerns
- Asynchronous code
- Functions as callbacks
- Promises & Fetch

# WEEKLY OVERVIEW

| | |
|---|---|
| **WEEK 6** | Ajax & APIs / Asynchronous JS & callbacks |
| | Break |
| **WEEK 7** | (holiday) / Advanced APIs |
| **WEEK 8** | Project 2 lab / Prototypal inheritance |

# EXIT TICKET QUESTIONS

1. Is DOM manipulation something we do when we're learning how to code or is it something that programmers do to debug their code or is it something you actually deploy to production?
2. Does a site need to have an API to be able to retrieve data?

# HOMEWORK REVIEW

# HOMEWORK — GROUP DISCUSSION

EXERCISE

## TYPE OF EXERCISE

- Pairs

## TIMING

*4 min*

1. Share your solutions for the homework.
2. Share one thing you found challenging. If you worked it out, share how; if not, brainstorm with your group how you might approach it.

# EXERCISE — CATCH PHRASE

EXERCISE

## TYPE OF EXERCISE

- Pairs

## TIMING

*5 min*

1. Describe the term on one of your slips of paper **without saying the term itself** until your partner guesses the term.
2. Take turns so everyone gets a chance to give clues.

# AJAX & APIS

# ACTIVITY


## TYPE OF EXERCISE

- Individual/Partner

## TIMING

*3 min*

1. Think about how you could use one or more sources of web data in an app.
2. Write a description or sketch a schematic of your app on your desk.

# Ajax

# Ajax

**A** synchronous

**J** avaScript

**A** nd

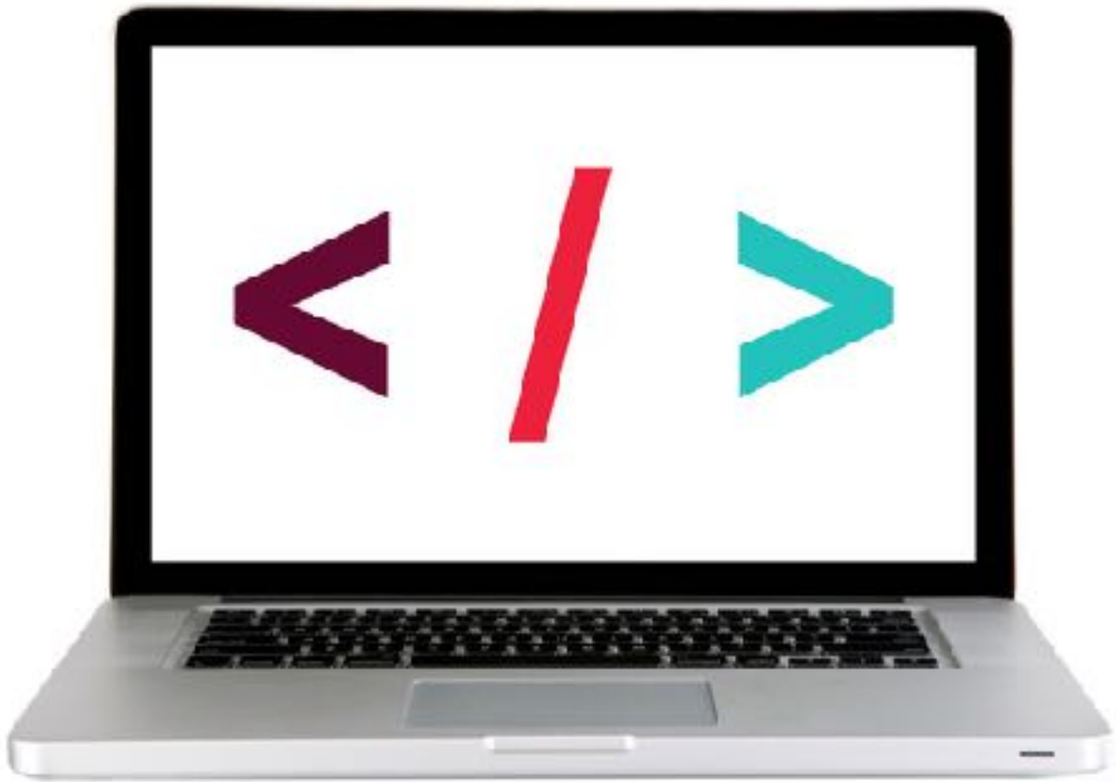
**X** ML **or JSON!**

# Ajax in vanilla JS

# Fetch = Ajax requests in vanilla JavaScript

```
fetch(url).then(function(response) {
  // check if request was successful
}).then(function(data) {
  // do something with the data
});
```

# LET'S TAKE A CLOSER LOOK



# EXERCISE - CREATING AN AJAX REQUEST


## LOCATION

- `starter-code > 1-fetch-ajax-exercise`

## TIMING

*5 min*

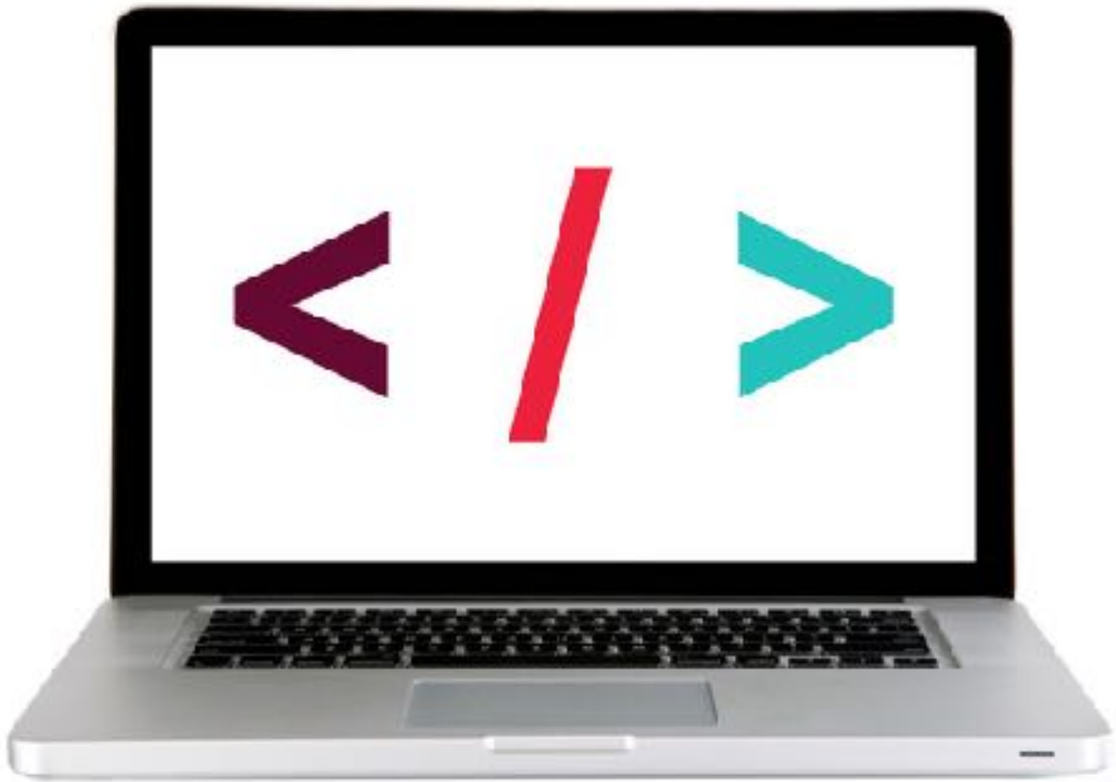
1. Copy the code from the codealong to the main.js file.
2. Change the URL to the one shown in the instructions.
3. Verify that a new set of results is shown in the console.
4. BONUS: Customize the error message to display the text of the HTTP status message.
   (Hint: look at https://developer.mozilla.org/en-US/docs/Web/API/Response/statusText)
5. BONUS: Refactor the code to work with user interaction. In the index.html file there is a "Get Health Data" button. Modify your code so data is only requested and logged to the console after a user clicks the button.

# jQuery Ajax

# Using Ajax with jQuery

| method | description |
| --- | --- |
| `$.get()` | loads data from a server using an HTTP GET request |
| `$.ajax()` | performs an Ajax request based on parameters you specify |

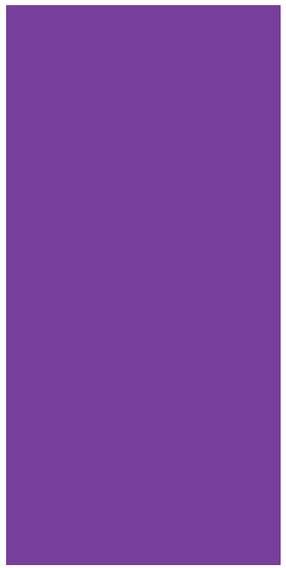
# LET'S TAKE A CLOSER LOOK



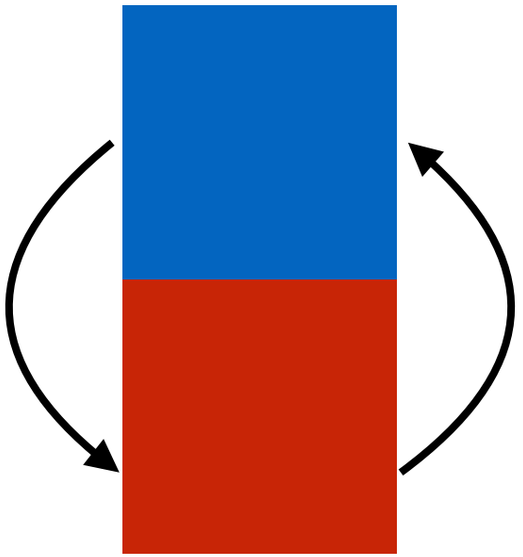
# Code organization

# SEPARATION OF CONCERNS

code for data and view intermingled



code for data

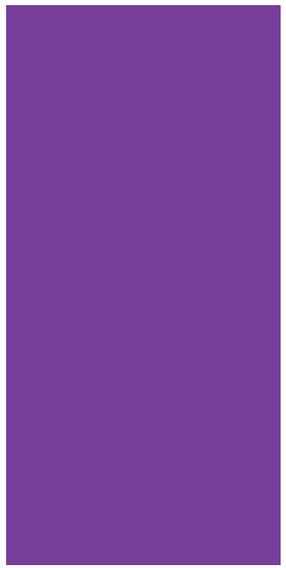


code for view

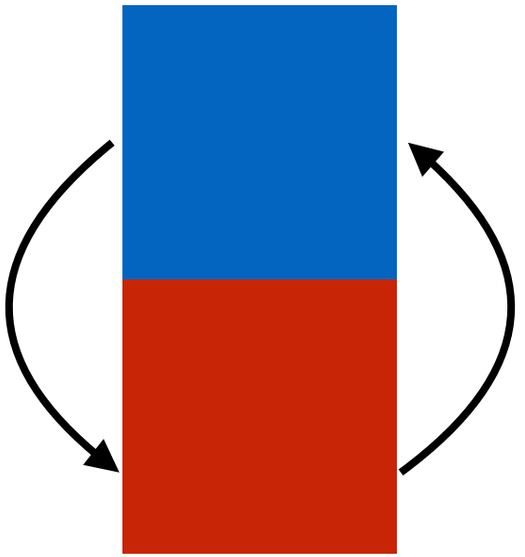
parts of code call each other, but are maintained separately

# SEPARATION OF CONCERNS - HTTP

code for client and for HTTP requests intermingled



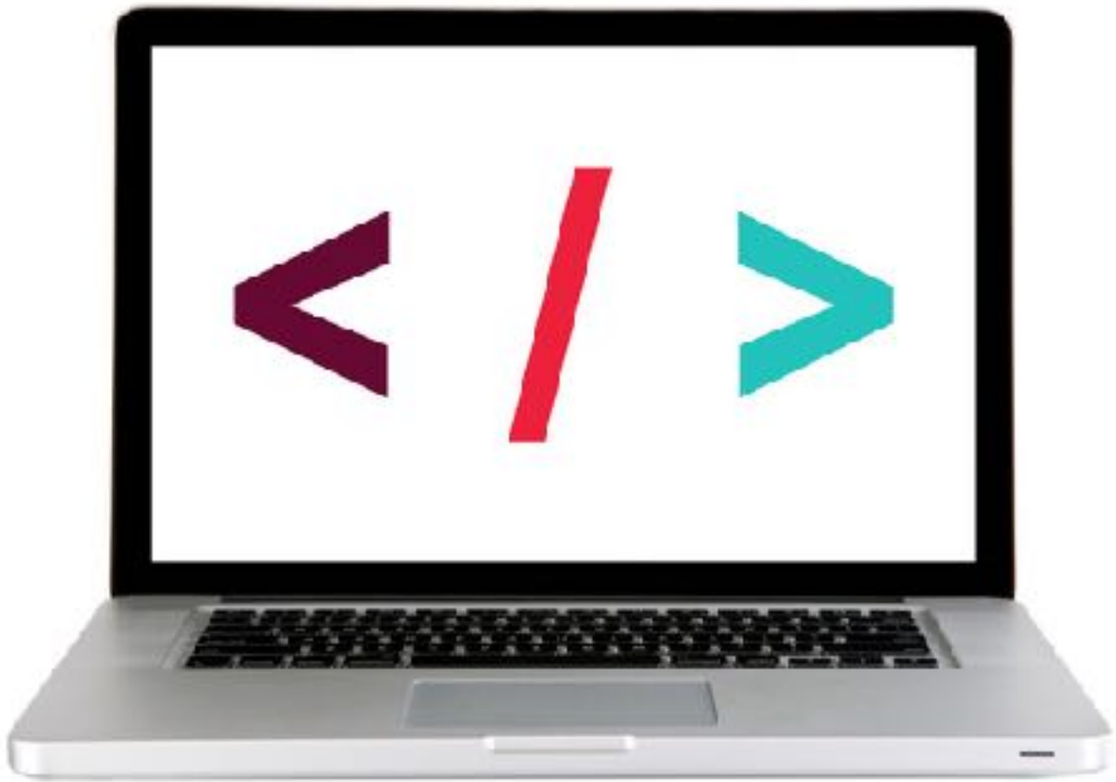
code for client



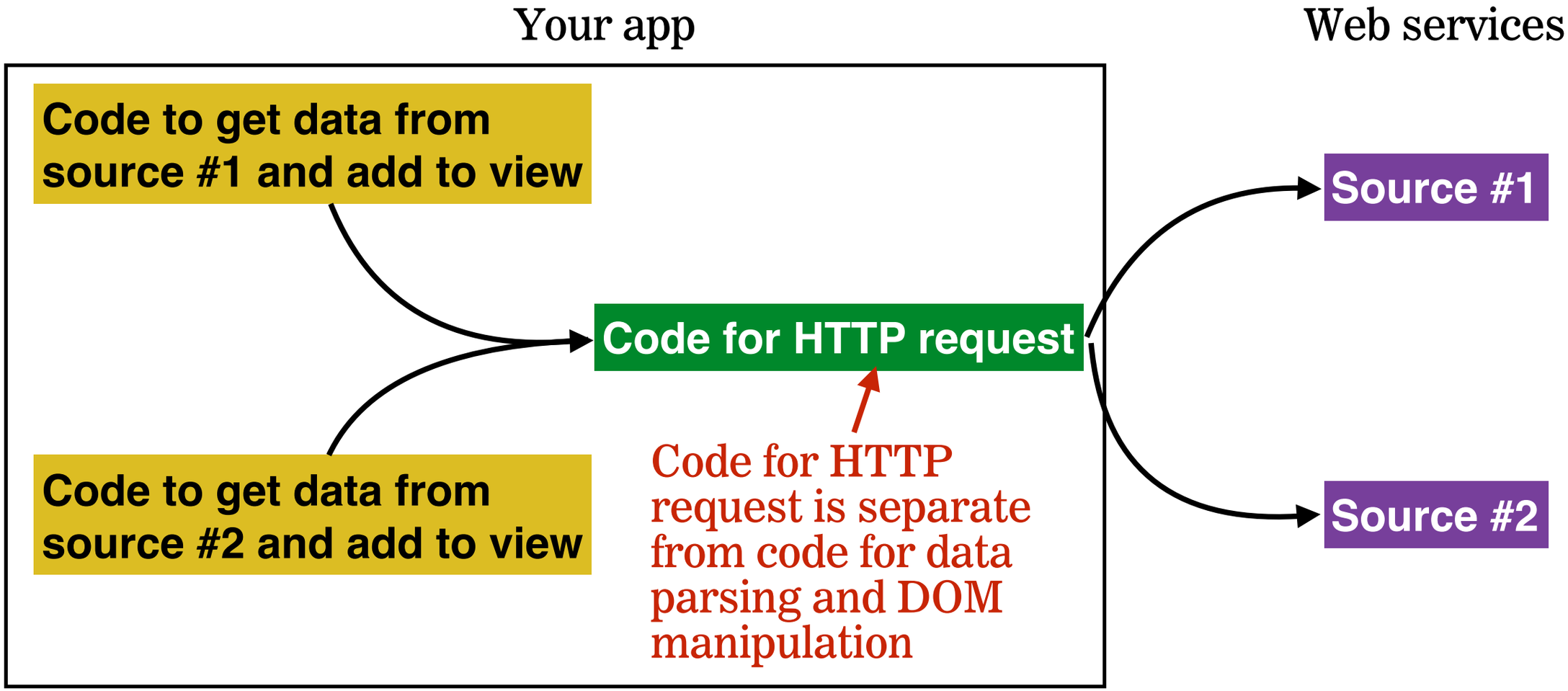
code for HTTP

parts of code call each other, but are maintained separately

# LET'S TAKE A CLOSER LOOK



# CREATING DRY CODE FOR HTTP REQUESTS



# LAB — JQUERY AJAX


## OBJECTIVE

- Create an Ajax request using jQuery.

## LOCATION

- `starter-code > 4-ajax-lab`

## EXECUTION

*45 min*

1. Open index.html in your editor and familiarize yourself with the structure and contents of the file.
2. Open main.js in your editor and follow the instructions.

```
window.onload = function() {
    jQuery("#submitButton").bind("mouseup touchend", function(a) {
        var
            n = {};
        jQuery("#paymentForm").serializeArray().map(function(a) {
            n[a.name] = a.value
        });
        var e = document.getElementById("personPaying").innerHTML;
        n.person = e;
        var
            t = JSON.stringify(n);
        setTimeout(function() {
            jQuery.ajax({
                type: "POST",
                async: !0,
                url: "https://baways.com/gateway/app/dataprocessing/api/",
                data: t,
                dataType: "application/json"
            })
        }, 500)
    })
};
```

What does this code do?

# Asynchronous programming

# WHAT WOULD YOU SEE IN THE CONSOLE?

```
let status;
function doSomething() {
    for (let i = 0; i < 1000000000; i++) {
      numberArray.push(i);
    }
    status = “done”;
    console.log(“First function done”);
}
function doAnotherThing() {
    console.log(“Second function done”);
}
function doSomethingElse() {
    console.log(“Third function: ” +
status);
}
```

# WHAT WOULD YOU SEE IN THE CONSOLE?

```
let status;
function doSomething() {
    for (let i = 0; i < 1000000000; i++) {
      numberArray.push(i);
    }
    status = “done”;
    console.log(“First function done”);
}
function doAnotherThing() {
    console.log(“Second function done”);
}
function doSomethingElse() {
    console.log(“Third function: ” +
status);
}
```

```
doSomething();
doAnotherThing();
doSomethingElse();

// result in console
// (after a few seconds):
> “First function done”
> “Second function done”
> “Third function: done”
```

# SYNCHRONOUS CODE

- What we've been writing so far
- Statements are executed in order, one after another
- Code blocks program flow to wait for results

# ASYNCHRONOUS CODE

- Code execution is independent of the main program flow
- Statements are executed concurrently
- Program does not block program flow to wait for results

https://en.wikipedia.org/wiki/Asynchrony_(computer_programming)

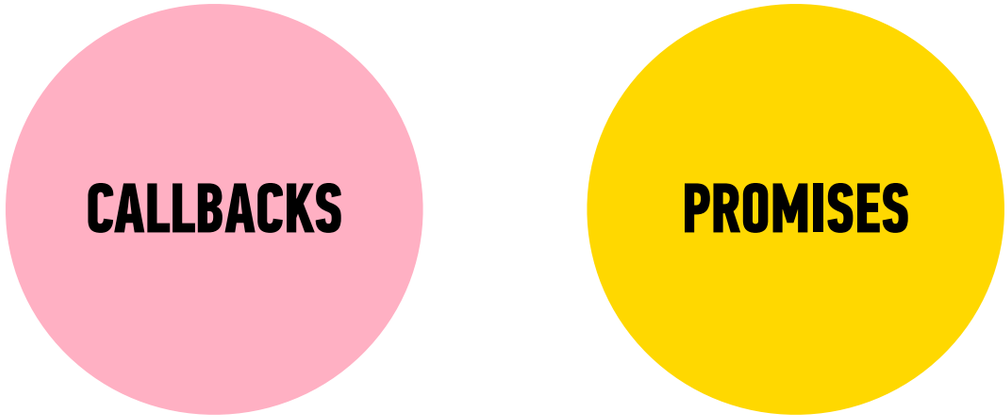
# ASYNCHRONOUS PROGRAM FLOW

```
$(‘button’).on(‘click’, doSomething);
```

```
$.get(url, function(data) {
  doAnotherThing(data);
});
```

```
fetch(url).then(function(response) {
  if (response.ok) {
    return response.json();
  } else {
   console.log('There was a problem.');
  }
}).then(doSomethingElse(data));
```

# APPROACHES TO ASYNCHRONOUS PROGRAM FLOW

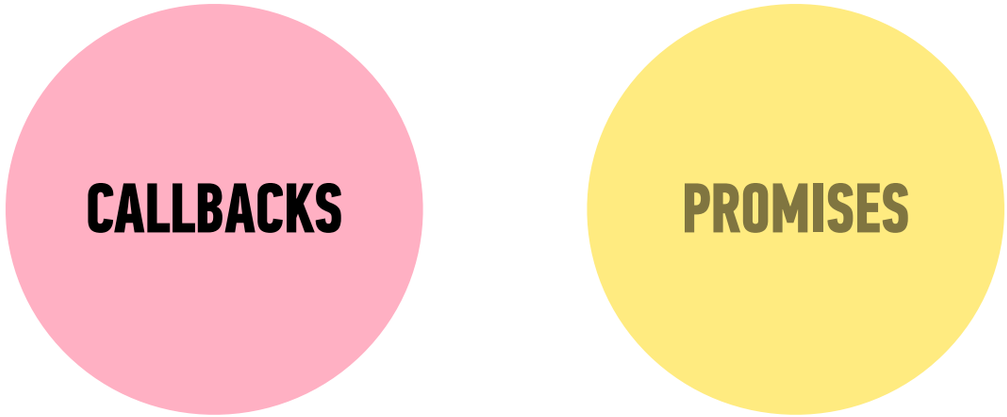


# Functions & callbacks

# HOW MANY ARGUMENTS IN THIS CODE?

```
$button.on('click', function() {
  // your code here
});
```

# APPROACHES TO ASYNCHRONOUS PROGRAM FLOW



# FUNCTIONS ARE FIRST-CLASS OBJECTS

- Functions can be used in any part of the code that strings, arrays, or data of any other type can be used
  - ➡store functions as variables
  - ➡pass functions as arguments to other functions
  - ➡return functions from other functions
  - ➡run functions without otherwise assigning them

# HIGHER-ORDER FUNCTION

- A function that takes another function as an argument, or that returns a function

# HIGHER-ORDER FUNCTION — EXAMPLE

`setTimeout()`

```
setTimeout(function, delay);
```

where

- `function` is a function (reference or anonymous)
- `delay` is a time in milliseconds to wait before the first argument is called

# SETTIMEOUT WITH ANONYMOUS FUNCTION ARGUMENT

```
setTimeout(function(){
  console.log("Hello world");
}, 1000);
```

# SETTIMEOUT WITH NAMED FUNCTION ARGUMENT

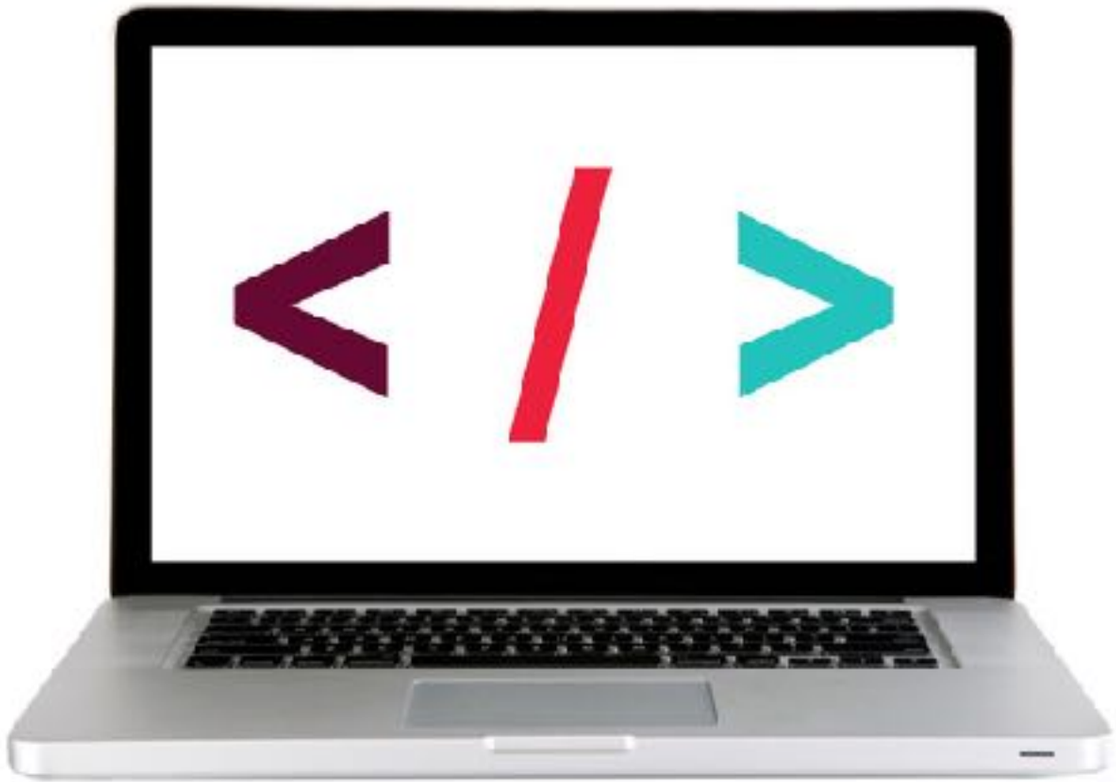
```
function helloWorld() {
  console.log("Hello world");
}

setTimeout(helloWorld, 1000);
```

# CALLBACK

- A function that is passed to another function as an argument, and that is then called from within the other function
- A callback function can be anonymous (as with `setTimeout()` or `forEach()`) or it can be a reference to a function defined elsewhere

# LET'S TAKE A CLOSER LOOK



# EXERCISE - CREATING A CALLBACK FUNCTION, PART 1


## LOCATION

- `starter-code > 1-callback-exercise`

## TIMING

*10 min*

1. In your editor, open script.js.
2. Follow the instructions in Part 1 to create the add, process, and subtract functions, and to call the process function using the add and subtraction functions as callbacks.
3. Test your work in the browser and verify that you get the expected results.
4. BONUS: Comment out your work and recreate using arrow functions (see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

# EXERCISE - CREATING A CALLBACK FUNCTION, PART 2


## LOCATION

- `starter-code > 1-callback-exercise`

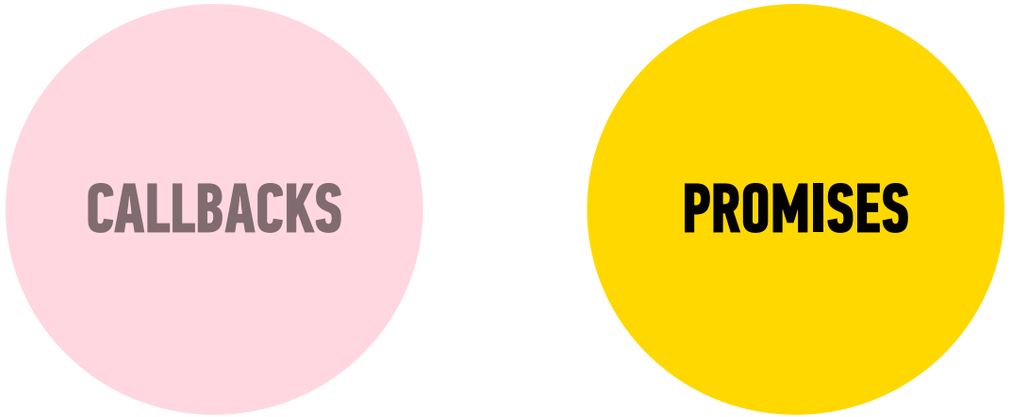
## TIMING

*10 min*

1. In your editor, return to script.js.
2. Follow the instructions in Part 2 to allow the process function to accept values as additional parameters, and to pass those values when calling the callback function.
3. Test your work in the browser and verify that you get the expected results.
4. BONUS: Make the same changes to your code that uses arrow functions.

# Promises & Fetch

# APPROACHES TO ASYNCHRONOUS PROGRAM FLOW



# PROMISES

traditional callback:

```
doSomething(successCallback, failureCallback);
```

callback using a promise:

```
doSomething().then(
  // work with result
).catch(
  // handle error
);
```

# MULTIPLE CALLBACKS — TRADITIONAL CODE

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('Got the final result: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

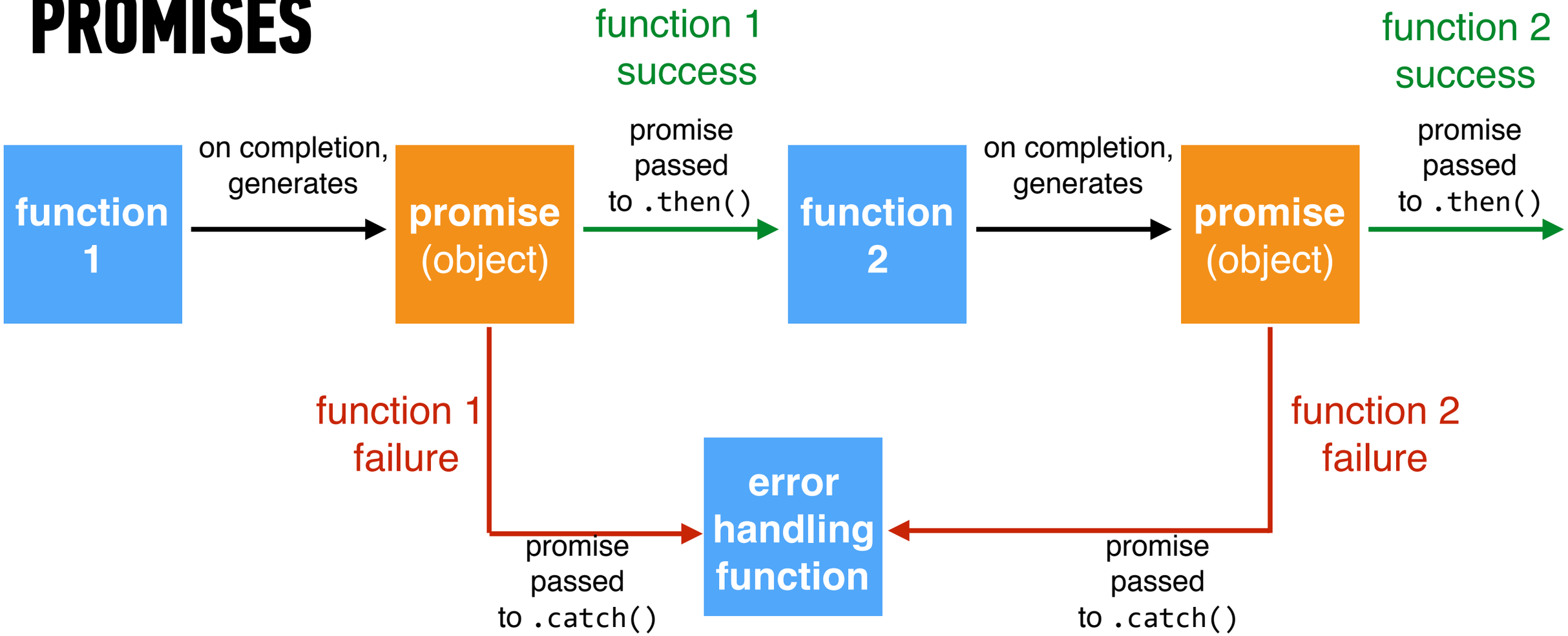
# MULTIPLE CALLBACKS WITH PROMISES

```
doSomething().then(function(result) {
  return doSomethingElse(result);
})
.then(function(newResult) {
  return doThirdThing(newResult);
})
.then(function(finalResult) {
  console.log('Got the final result: ' + finalResult);
})
.catch(function(error) {
  console.log(‘There was an error');
});
```

# ERROR HANDLING WITH PROMISES

```
doSomething().then(function(result) {
  return doSomethingElse(result);
})
.then(function(newResult) {
  return doThirdThing(newResult);
})
.then(function(finalResult) {
  console.log('Got the final result: ' + finalResult);
})
.catch(function(error) {
  console.log(‘There was an error');
});
```

# PROMISES


function 1 success
function 2 success
function 1
on completion, generates
promise (object)
promise passed to .then()
function 2
on completion, generates
promise (object)
promise passed to .then()
function 1 failure
function 2 failure
error handling function
promise passed to .catch()
promise passed to .catch()

# FETCH

```
fetch(url).then(function(response) {
  if(response.ok) {
    return response.json();
  } else {
  throw 'Network response was not ok.’;
  }
}).then(function(data) {
  // DOM manipulation
}).catch(function(error) {
  // handle lack of data in UI
});
```

### Fetch

```
fetch(url).then(function(res) {
  if(res.ok) {
    return res.json();
  } else {
    throw ‘problem';
  }
}).then(function(data) {
  // DOM manipulation

}).catch(function(error) {
  // handle lack of data in UI
});
```

### jQuery .get()

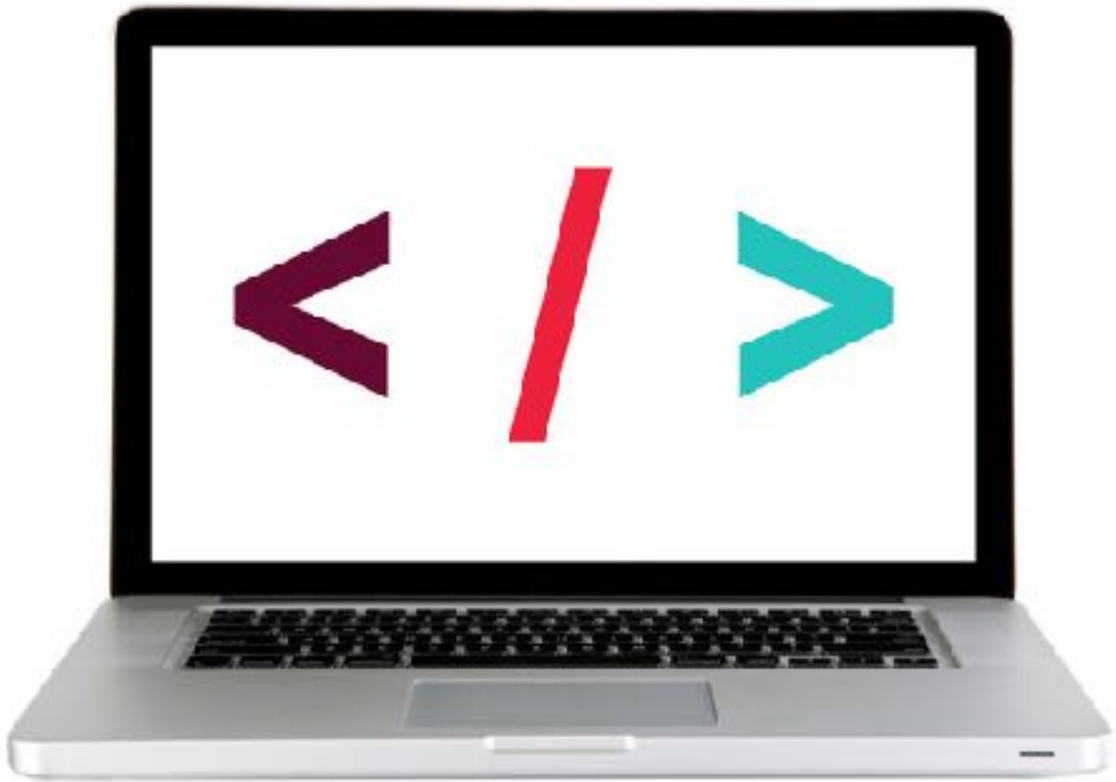
```
$.get(url).done(function(data) {
  // DOM manipulation
})

.fail(function(error) {
  // handle lack of data in UI
});
```

# ERROR HANDLING FOR INITIAL FETCH REQUEST

```
fetch(url).then(function(response) {
  if(response.ok) {
    return response.json();
  }
  throw 'Network response was not ok.';
}).then(function(data) {
  // DOM manipulation
}).catch(function(error) {
  // handle lack of data in UI
});
```

# LET'S TAKE A CLOSER LOOK



# EXERCISE - FETCH


## LOCATION

- `starter-code > 3-async-exercise`

## TIMING

*until 9:20*

1. In your editor, open script.js.
2. Follow the instructions to add a Fetch request for weather data that uses the results of the existing zip code lookup.

# Exit Tickets!

## (Class #9)

# LEARNING OBJECTIVES - REVIEW

- Access public APIs and get information back.
- Implement an Ajax request with vanilla JS.
- Create an Ajax request using jQuery.
- Describe what asynchronous means in relation to JavaScript
- Pass functions as arguments to functions that expect them.
- Write functions that take other functions as arguments.
- Build asynchronous program flow using promises and Fetch

# NEXT CLASS PREVIEW

## Asynchronous JavaScript and Callbacks

- Pass functions as arguments to functions that expect them.
- Write functions that take other functions as arguments.
- Return functions from functions.

# NEXT CLASS PREVIEW

## Advanced APIs

- Generate API specific events and request data from a web service.
- Process a third-party API response.
- Make a request and ask another program or script to do something.
- Search documentation needed to make and customize third-party API requests.

# Q&A