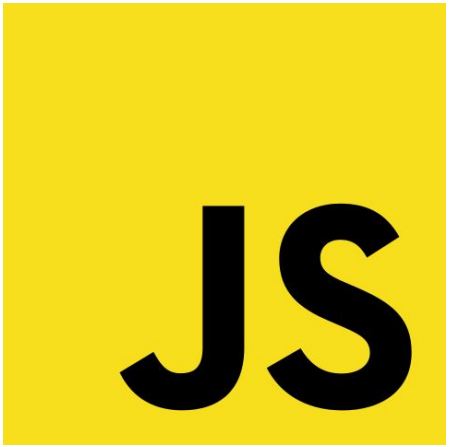
Web Development using Javascript

About me

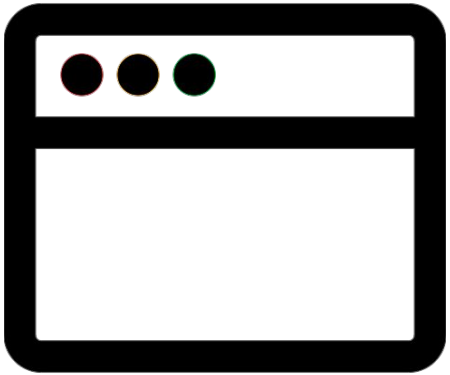
- Currently Staff Engineer at Xendit 👨‍💻, building payment infrastructure 💸 for SouthEast Asia
- Experience with companies at multiple stages with different engineering needs and wants
- Worked with Front-end, Back-end (even Weekend, Dead-end if you want me to 😜)

It takes two types of knowledges:

JavaScript language: syntaxes and features of the language itself



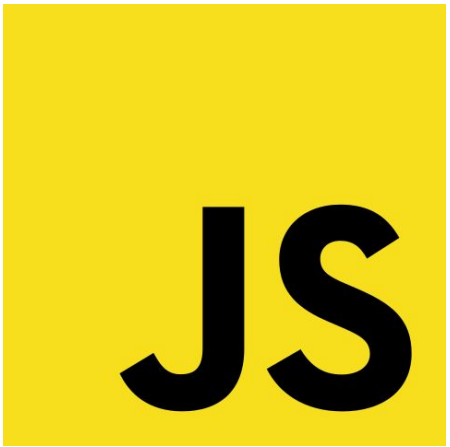
Browser API: “built-in” functionalities provided by browsers



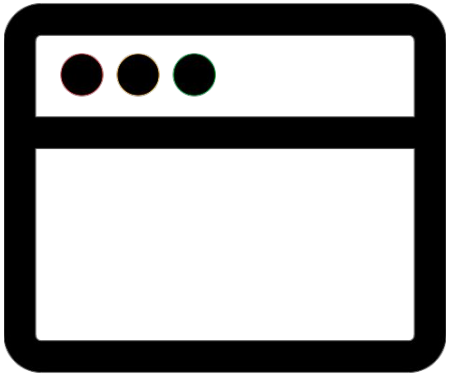
It takes two types of knowledges:

JavaScript language: syntaxes and features of the language itself

Browser API: “built-in” functionalities provided by browsers



JavaScript features works in any JS environment, e.g. NodeJS on server

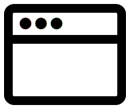


How to run JavaScript in a web page

1. within HTML: **`<script>{....}</script>`**
2. external script: **`<script src=”./script.js” />`**
3. “**`javascript:<code>`**” in your browser URL
4. Run in your console

Copy the text below and paste in URL bar of your browser:

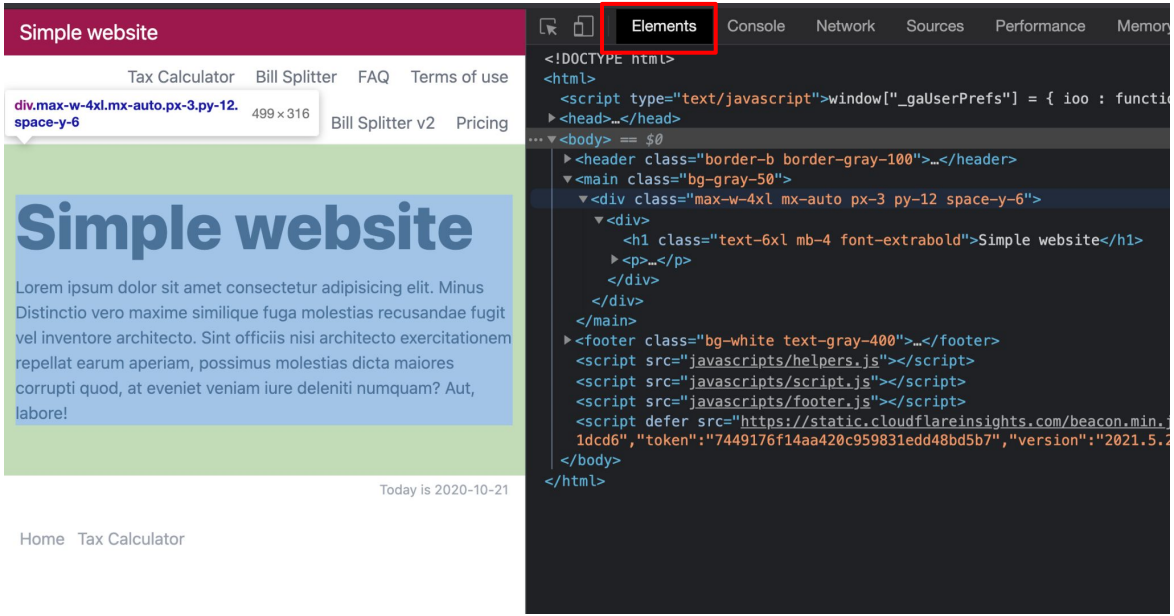
```
javascript:alert('hello world');
```

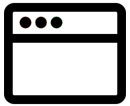


Browser DevTools

Open by typing “F12” or Right Click + Inspect

1. **Elements**: You can inspect/edit HTML & CSS

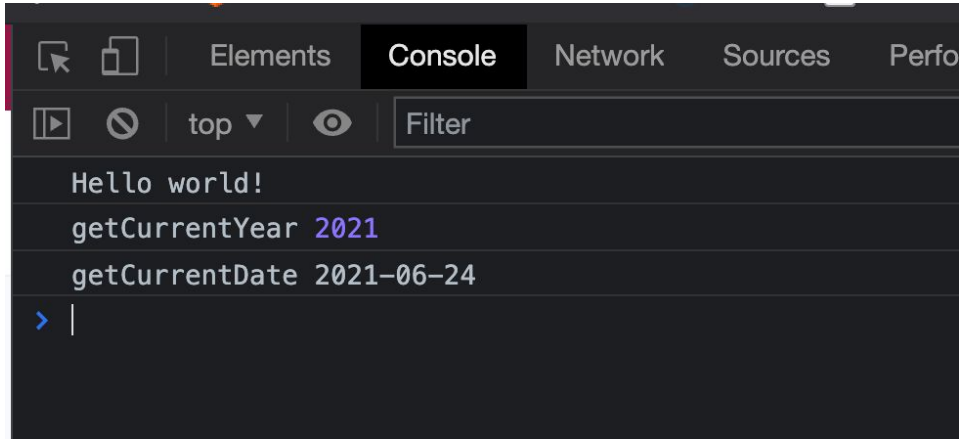


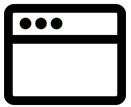


Browser DevTools

Open by typing “F12” or Right Click + Inspect

1. **Elements**: You can inspect/edit HTML & CSS
2. **Console**: Run javascript and logging

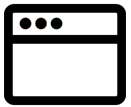




Browser DevTools

Open by typing “F12” or Right Click + Inspect

1. **Elements**: You can inspect/edit HTML & CSS
2. **Console**: Run javascript and logging
3. **Network** (will be covered in Day 2)



Useful Links

- Code Sandbox:
  - https://codesandbox.io/s/intro-to-js-playground-0czx0
  - Sign up for an account and fork the sandbox project
  - Directly edit the code in your own fork for in-class exercises and homeworks
  - Submit the link to your forks for assessments by emailing to hung.ngn.the@gmail.com
- Office Hour (Sat 3-6PM): https://calendly.com/stanley_nguyen/office_hour
- Capstone project:
  - Rubrics https://drive.google.com/file/d/1LY_aCthhTG9pJCs1blYoouojCJJGKP-8/view?usp=sharing
  - Optional (but preferred): Use github.com to host your code

Public Service Announcements

- Deadline for homework from Day 1 and Day 2 is this Friday 1159PM
- Cyrille will be releasing solutions for homework from Day 1 and Day 2 on Saturday

Agenda for today

- Chapter 1: Update HTML
- Chapter 2: Variables and Data Types
- Chapter 3: The DOM
- =🌭=🍔= Lunch BREAK =🥗=🍜=
- Chapter 4: Functions
- Chapter 5: Equality and Flow Control
- Homework / Project Help

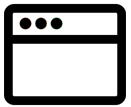
Chapter 1: Update HTML

Demo

Visit https://codesandbox.io/s/intro-to-js-playground-0czx0

and click the Fork button at top right corner or in the bottom right prompt.

file: `public/javascripts/footer.js`



Declaring variable in JavaScript

```
const footerYearEl = document.querySelector('#footer-year');
```

keyword

variable name

expression

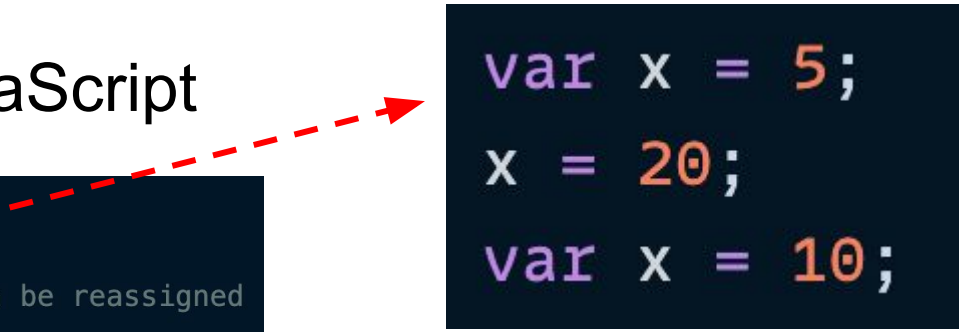
JS

Declaring variable in JavaScript

```
var x = 5; // old way, can be redeclared

let y = 10; // cannot be redeclared but can be reassigned

const z = 15; // cannot be redeclared or reassigned
```

Declaring variable in JavaScript

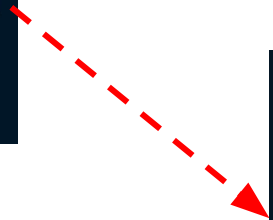
```
var x = 5; // old way, can be redeclared

let y = 10; // cannot be redeclared but can be reassigned

const z = 15; // cannot be redeclared or reassigned
```



JS

Declaring variable in JavaScript

```
var x = 5; // old way, can be redeclared

let y = 10; // cannot be redeclared but can be reassigned

const z = 15; // cannot be redeclared or reassigned
```



```
let y = 10;
y = 12;
let y = 21;
```

JS

Declaring variable in JavaScript

```
var x = 5; // old way, can be redeclared

let y = 10; // cannot be redeclared but can be reassigned

const z = 15; // cannot be redeclared or reassigned
```

```
const z = 15;
z = 16; // not allowed
const z = 17; // not allowed
```

JS

Declaring variable in JavaScript

```
var x = 5; // old way, can be redeclared

let y = 10; // cannot be redeclared but can be reassigned

const z = 15; // cannot be redeclared or reassigned
```

Use const by default, switch to let when u want to make a variable reassignable.

JS

Comments

```
let y = 5; // everything after // will be ignored by JS

/*
 This is multiline comment.
 Everything between slash star and star slash
 will be ignored by JS.
 You can draw ART here if you want:
    |   |   .:'
    |  __ :'__
   .'`  `-'  ``.
  :____________.-'
  :____________:
   :____________`-;
    `.__.-.__.'
*/
```

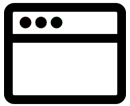
Selecting element

`document.querySelector('<css selector>')`

```
const footerYearEl = document.querySelector('#footer-year');
```

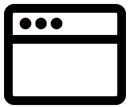
function name

arguments (data provided to function)



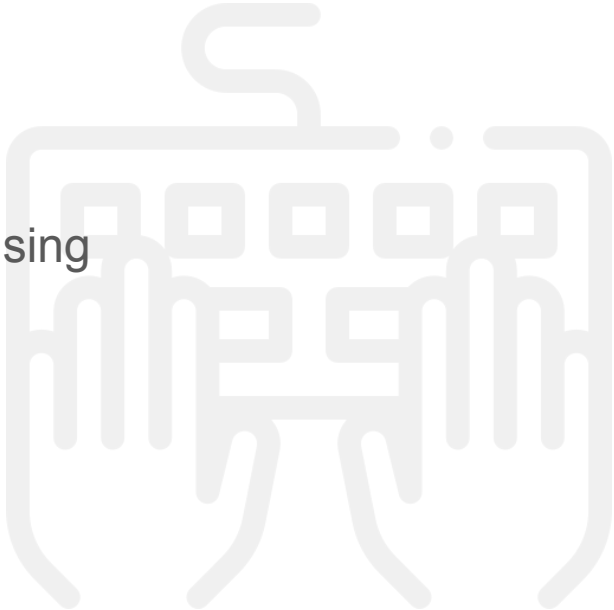
Updating element HTML

```
item3.innerHTML = 'hello <i>You<i>';
```



Do It

Update the date in footer so it reflects the date today using `helpers.getCurrentDate` function.



Recap

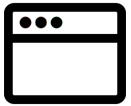
1. Use **`const`** and **`let`** to declare variables.
2. **`document.querySelector`** can be used to get HTML element.
3. Assigning value to **`element.innerHTML`** will update the content of the HTML element.

Chapter 2: Variables and Data Types

Demo

Visit Tax Calculator page (`/tax-calculator`) in preview.

file: **`public/javascripts/calculator.js`**

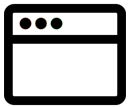


Getting input value

```
const salary = document.querySelector('#salary').value;
```

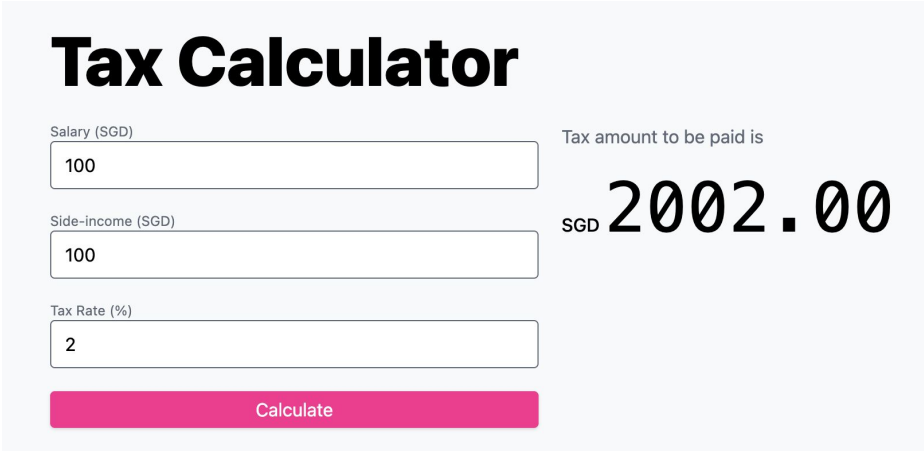
this get us the input element

this get the value of the input



We need to understand more about data types to understand why the calculator is wrong

```
function calculateTax() {
  // alert('calculateTax is called');
  // DEMO: perform the tax calculation here

  const salary = document.querySelector('#salary').value;
  const sideIncome = document.querySelector('#sideIncome').value;
  const taxRate = document.querySelector('#taxRate').value;

  const total = ((salary + sideIncome) * taxRate) / 100;

  document.querySelector('#result').innerHTML = total.toFixed(2);
}
```



Value Type in JavaScript

Primitive

- `undefined`: unintentionally missing value
- `null`: intentionally missing value
- `boolean` (true or false): used for logical operations
- `number` (-1.0003, 23412): used for mathematical calculation
- `string` (“Love”, “Richard”): used for text
- `symbol` (uncommon): used to hide implementation details
- `bigint` (uncommon): used for math of big numbers

Non-primitive

- `function`: used to refer to code
- `object`: used to group related data and code

JS

undefined

All variable that are declared but not assigned will be undefined by default.

```
let x;

console.log(x); // undefined

x = 5;

console.log(x); // 5
```

JS

`null`

`null` is usually used to indicate an intentional empty value. For example, `document.querySelector` returns `null` if there is no element match the provided selector.

Functionally there is no difference with undefined. Both of them means empty.

JS

`boolean`

There are only two possible value for boolean, true and false

Similar to other programming language, we can combine multiple boolean value into one using logic operator

- `&&`: AND
- `||` : OR

- `&&` only results in true if both boolean are true
- `||` results in true if either boolean is true

JS

`number`

We can do maths with number, e.g.

```
let total = 1000 * (10 + 90) / 20;
```

Other than simple arithmetic operation (+,-,*,/), there are other common mathematical operation available in the **`Math`** object, e.g. **`Math.random`**, **`Math.min`**, **`Math.pow`**.

JS

string

```
let s1 = 'simple';

let s2 = "simple too";

let s3 = 'No difference unless "this"!';

let s4 = `Multi
        line
       string`;
```

JS

string

```
let s1 = 'simple';

let s2 = "simple too";

let s3 = 'No difference unless "this"!';

let s4 = `Multi
        line
       string`;
```

Single quote or double quote works entirely the same except when you want to use one of them in the string itself

JS

`string` operations

```
let str1 = 'me';
str1.toUpperCase(); // 'ME'

let str2 = 'handsome';
let joined = str1 + ' ' + str2; // 'me handsome';
let joined2 = `${str1} ${str2}`; // 'me handsome';

let str3 = `
   lonely
`;

str3.toUpperCase(); // 'lonely'
```

JS

object

We can group related data using object.

```
let person = {
    name: 'Taylor Swift',
    age: 31,
    married: undefined,
    retired: false
};
```

name, age, married, and retired are the **properties** of the person object, while ‘Taylor Swift’, 31, undefined and false are the **values** of those properties.

In other words:

- the person object has 4 properties.
- the name property of the person object has the value of ‘Taylor Swift’

JS

object

To get/set the value of properties, we can use **dot notation**.

```
let person = {
    // ...
}

let taylorAge = person.age; // getting the value

person.married = true; // setting the value
```

There are other ways to get the value of object properties, which we will introduce later on.

JS

`object`

Other than declaring the object ourself, we may also get object by browser API, e.g. the result of **`document.querySelector`**, a HTML element, is also an object.

```
item3.innerHTML = 'hello <i>You<i>';
```

This is setting **`innerHTML`** property value for **`item3`** HTML element (an object).

Checking type of value

We can use `typeof` operator to check the type of a value

```
console.log(typeof 'Hello'); // 'string'
console.log(typeof -1293.23); // 'number'
console.log(typeof console.log); // 'function'
```

JS

Do It

Create a file with name **`exercise.js`** in your editor and write code to:

1. declare a variable x without value and then assign it with a value 100.
2. declare a variable y with value true then replace the value with false.
3. declare a variable that store your first name, last name, and age.

Question

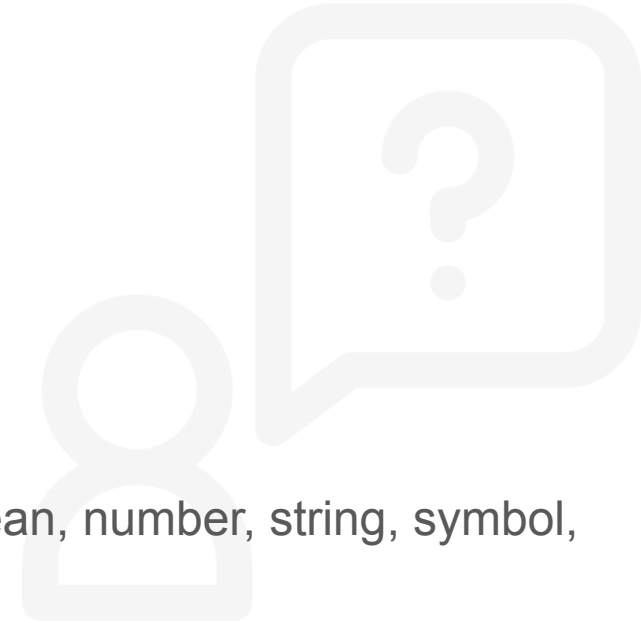
There is a Javascript bug in implementing

typeof that returns wrong value.

Do you know which is that value type?

To recap, the value types are undefined, null, boolean, number, string, symbol, bigint, object, function.

(Hint: You can try to run typeof in your console to find out.)



JS

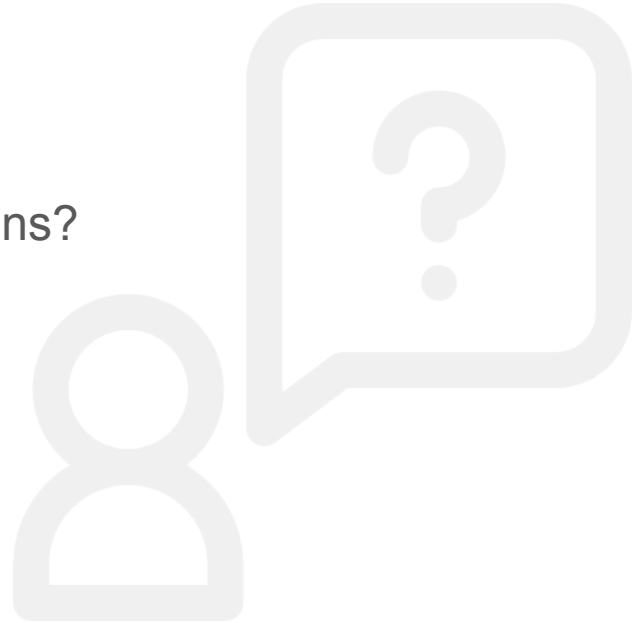
Question

What is the result of each of the following expressions?

```
12 / '6'
```

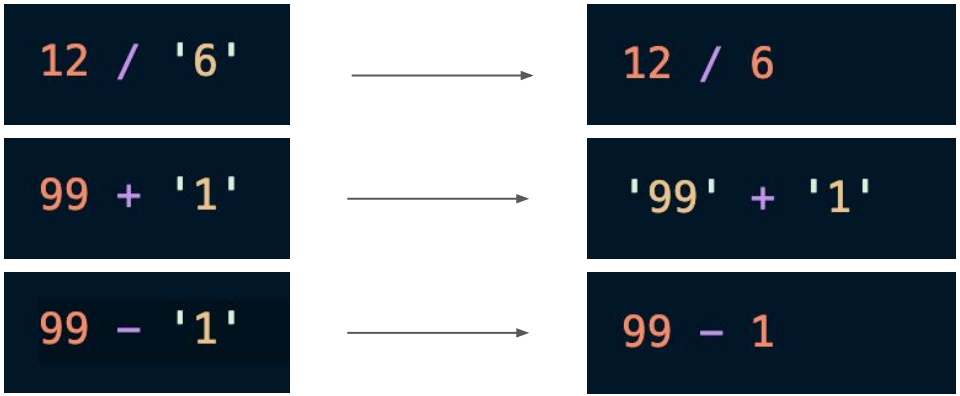
```
99 + '1'
```

```
99 - '1'
```



JS

Type coercion is the process of converting value from one type to another.



JS

We can avoid the confusion with type coercion by converting type explicitly.

```
12 / Number('6')

99 + Number('1')

99 - Number('1')
```

JS

! and !!

You can use ‘!’ to coerce a value to its opposite boolean value and ‘!!’ to get its equivalent boolean value.

Very common to check for **falsy** value, i.e. `null`, `undefined`, `''`, `0`

```
let value = '';
if (!value) {
    console.log('value is empty!');
}
if (!!value) {
    console.log('value is not empty');
}
```

JS

Question

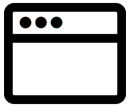
What will be logged in the console?

```
let designation = '';
let age = 24;
if (!age) {
  designation = 'baby';
}
if (!!designation) {
  console.log(`Hello ${designation} boy!`);
} else {
  console.log('Hello man');
}
```


JS

Demo

file: `public/javascripts/calculator.js`



Recap

1. Javascript has 9 data types: undefined, null, boolean, number, string, symbol, bigint, function, and object.
2. We can check the data type of a variable by using `typeof` operator.
3. Each data types has operations that it can perform, e.g. number can multiply and string can be concatenated.
4. When performing operation on different data types, coercion will be performed by JavaScript to convert the data types.
5. Coercion may lead to confusing behavior, therefore it is recommended to convert the data types explicitly when performing operation that involves two data types.

Do It

Go to Bill Splitter (`/bill-splitter`) page, and you should see a simple application to calculate how much each person should pay when sharing a bill.

Update the code in `splitBill` (in `calculator.js`) so that you can calculate the amount using the calculation

payment by each person = total bill / pax

Chapter 3: DOM

DOM (Document Object Model)

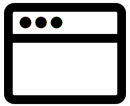
- When you write HTML, browser will convert the HTML markup to **Document Object Model (DOM)**.
- DOM is the data structure that allows JavaScript to manipulate the HTML markup

❌ Not this DOM

`window` object

- When javascript run in browser, you have access to window object which contains useful information about the browser window as well as all global objects.
- `window` object contains everything of the tab/window you open, including URL bar, the page content, etc.

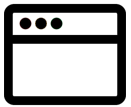
```
console.log(window.location);

console.log(location);

console.log(window.innerWidth);

console.log(window.innerHeight);
```



`document` object

- `document` object is the object contains everything in your HTML, from your `<!DOCTYPE html>` to the closing html tag.
- `document` will be the primary way of us working with DOM.

```
console.log(document);
```



`navigator` object

- There is also something called the **`navigator`** that is higher level than **`window`** object
- **`navigator`** object gives you information about the browser and the device itself. Things like web cam and geo-location access will be available on navigator.
- We will not cover navigator much, just be aware that **`window`**, **`document`**, and **`navigator`** are the 3 most commonly used browser API. Most browsers API are available on one of them.
- Check out https://developer.mozilla.org/en-US/docs/Web/API whenever you need to use a Web API

Example: Using **`navigator`** to get battery status

```
navigator.getBattery()
▶ Promise {<pending>}
navigator.getBattery().then(console.log)
▼ BatteryManager {charging: true, chargingTime: 5700, dischargingTime: Infinity, level: 0.75, onchargingchange: null, …}
    charging: true
    chargingTime: 5700
    dischargingTime: Infinity
    level: 0.75
    onchargingchange: null
    onchargingtimechange: null
    ondischargingtimechange: null
    onlevelchange: null
  ▶ __proto__: BatteryManager
```

Recap

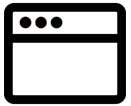
1. DOM is the data structure that allows JavaScript to manipulate the HTML markup
2. `window` object contains everything of the tab/window you open
3. `document` object is the object contains everything in your HTML
4. `navigator` object gives you information about the browser and the device

Chapter 4: function

Demo

Visit FAQ page (`/faq`) in preview.

file: **`public/javascripts/faq.js`**



function

function is the way to group together sets of statements.

For example

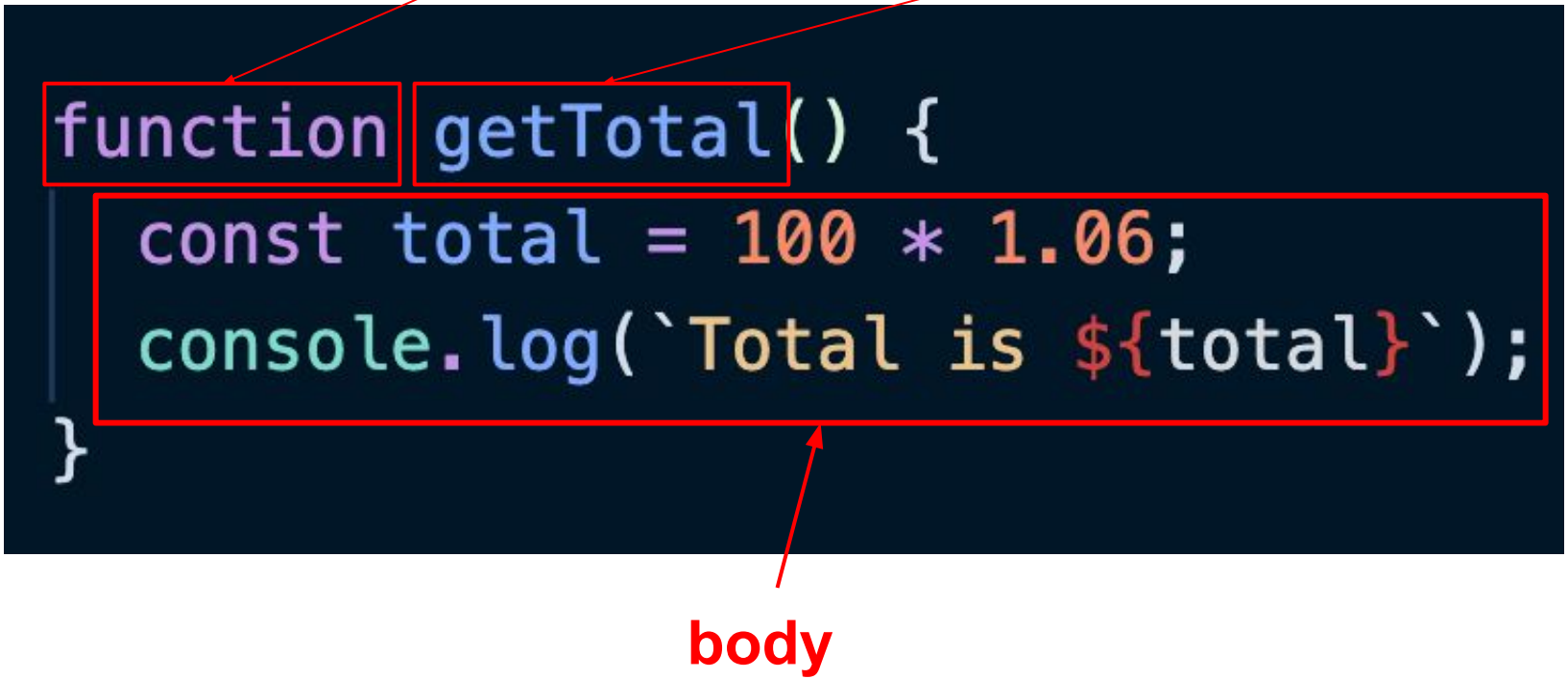
```
function logTotal(x, y) {
    const total = x + y;
    console.log(total);
}

logTotal(1, 3);
logTotal(1, 1);
```

Declaring function

Calling function (twice)

JS

Declaring function



function

function can perform some work (like `console.log`), and sometimes they return a data.

function can take in data (like 1, 3) above. Those data are known as **arguments**.

JS

function

There are many built in function provided by JavaScript. E.g.

```
parseInt('100.23');
parseFloat('100.23');
Math.max(100, 10, 1000);
Date.now();
```

Technically speaking, Math.max and Date.now are **methods** (which we will discuss shortly), there is little difference between method and function.
For now, just treat them as similar.

JS

function to update element `class`

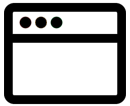
`classList` is a property on html elements that allows us to manipulate class. It provide 3 methods that we can call: `add`, `remove`, and `toggle`.

```
panel.classList.add('hidden');
```

Add class to the element

```
panel.classList.toggle('hidden');
```

Add class to the element if the element does not has the class currently.
Remove the class if the element is having the class.



Function can return value

```
function getTotal() {
  const total = 100 * 1.06;
  return total;
}

// usage
const calculatedTotal = getTotal();
console.log(calculatedTotal);
```

JS

Add parameters to function to make it more reusable

```
function calculateTotal(listPrice) {
    const total = listPrice * 1.06;
    return total;
}

const billTotal = calculateTotal(100);
console.log(`Bill total is ${billTotal}`);
```

listPrice is the **parameter** of the function

100 is the **argument** provided to the getTotal function

JS

Add parameters to function to make it more reusable

```
function calculateTotal(listPrice, taxRate) {
    const total = listPrice * taxRate;
    return total;
}

const billTotal = calculateTotal(100, 1.06);
console.log(`Bill total is ${billTotal}`);
```

We can declare multiple **parameters** for the function.

100 and 1.06 are the **arguments** provided to the getTotal function

JS

You can provide default value for parameter

```
function calculateTotal(listPrice, taxRate = 1.06) {
    const total = listPrice * taxRate;
    return total;
}

const billTotal = calculateTotal(100);
console.log(`Bill total is ${billTotal}`);
```

JS

Question

```
function getTotal(listPrice, taxRate) {
  const total = listPrice * taxRate;
  return total;
}

const price = 1000;
const countryTaxRate = 1.1;

const billTotal = getTotal(price, countryTaxRate);
console.log(`Bill total is ${billTotal}`); // what will be logged?
```

JS

Question

```
const listPrice = 100;

function getTotal(listPrice, taxRate) {
  const total = listPrice * taxRate;
  return total;
}

const price = 1000;
const countryTaxRate = 1.1;

const billTotal = getTotal(price, countryTaxRate);
console.log(`Bill total is ${billTotal}`); // what will be logged?
```

JS

Do It

In your **`exercise.js`** file,

1. Write a function that calculate number of seconds in a week.
2. Write a function that accepts number of days and returns the number of seconds in those days.
3. Write a function that accepts number of period and the period unit and returns the number of seconds in the period. The period unit can be ‘day’ or ‘week’. When period unit is not provided, default it to ‘day’.

function can be passed as parameter

In javascript, function can be passed as parameter, just like other data types.

```
function logHelloAndBye(doInMiddle) {
  console.log('hello');
  doInMiddle();
  console.log('bye');
}

function logName() {
  console.log('I am Winnie the Pooh');
}

logHelloAndBye(logName);
```

This will log
1. ‘hello’
2. ‘I am Winnie the Pooh’
3. ‘bye’

function can be passed as parameter

In javascript, function can be passed as parameter, just like other data types.

```
function logHelloAndBye(doInMiddle) {
  console.log('hello');
  doInMiddle();
  console.log('bye');
}

function logName() {
  console.log('I am Winnie the Pooh');
}

logHelloAndBye(logName);
```

This will log
1. ‘hello’
2. ‘I am Winnie the Pooh’
3. ‘bye’

`logName` is passed to **`logHelloAndBye`** and called by it. That’s why the function that is passed as parameter is usually known as **callback function**.

An example of using callback function is **`element.addEventListener`**

```
element.addEventListener('<event>', callback);
```

When **`<event>`** occurs, the browser will call the **`callback`** function.

An example of using callback function is **`element.addEventListener`**

```
element.addEventListener('<event>', callback);
```

When **`<event>`** occurs, the browser will call the **`callback`** function.

```
function onButtonClick() {
  panel.classList.toggle('hidden');
}

btn.addEventListener('click', onButtonClick);
```

When **`click`** event occurs, the browser will call **`onButtonClick`**.

The function can be rewritten below

```
function onButtonClick() {
  panel.classList.toggle('hidden');
}

btn.addEventListener('click', onButtonClick);
```

Instead of declaring the function and then pass it, we can declare the function where we pass it.

```
btn.addEventListener('click', function onButtonClick() {
  panel.classList.toggle('hidden');
});
```

The function can be rewritten below

```
function onButtonClick() {
  panel.classList.toggle('hidden');
}

btn.addEventListener('click', onButtonClick);
```

```
btn.addEventListener('click', function onButtonClick() {
  panel.classList.toggle('hidden');
});
```

```
btn.addEventListener('click', function () {
  panel.classList.toggle('hidden');
});
```

The function name can be omitted. Function without name like this is known as **anonymous function**.

Anonymous function: function without name

```
btn.addEventListener('click', function () {
  panel.classList.toggle('hidden');
});
```

Anonymous function is usually used as callback function or in an **IIFE** (immediately invoked function expression, will be covered later), where the function is being used where it is declared.

Function Expression

```
const shout = function (word) {
    return word.toUpperCase();
}
```

Other than function declaration and anonymous function, the third way of defining a function is function expression.

You can see function expression as “assigning an anonymous function to a variable”.

JS

Arrow Function

The last way of defining a function is arrow function.

```
const shout = (word) => {
    return word.toUpperCase();
}
```

JS

Arrow Function

The last way of defining a function is arrow function.

```
const shout = (word) => {
    return word.toUpperCase();
}
```

To change a function expression (shown below) to arrow function, remove the **`function`** keyword and add an arrow (**=>**) behind the parameter parenthesis

```
// function expression
const shout = function (word) {
  return word.toUpperCase();
}
```

JS

Arrow Function

Parenthesis in arrow function can be omitted if the function has **only one parameter**.

```
const shout = (word) => {
    return word.toUpperCase();
}
```

```
const shout = word => {
  return word.toUpperCase();
}
```

```
const plus = (a, b) => {
  return a + b;
}
```

```
const plus = a, b => {
  return a + b;
}
```

JS

Arrow Function

The curly braces and return can be omitted if the function body **has single expression**.

```
const shout = (word) => {
    return word.toUpperCase();
}
```

```
const shout = (word) =>
  word.toUpperCase();
```

```
const double = (x) => {
  const result = x * 2;
  return result;
}
```

```
const double = (x) =>
  const result = x * 2;
  return result;
```

JS

Arrow function

For implicit return, there is a gotcha when you return an object.

```
const calculateBill = (unitPrice, quantity, taxRate) => {
    pretaxTotal: unitPrice * quantity,
    total: unitPrice * quantity * (1 + taxRate),
}; // !Error

const calculateBill = (unitPrice, quantity, taxRate) => ({
  pretaxTotal: unitPrice * quantity,
  total: unitPrice * quantity * (1 + taxRate),
});
```

JS

Recap

1. function can be declared using
    a. function declaration,
    b. anonymous function,
    c. function expression, and
    d. arrow function
2. function can accept parameters and return value, but not necessary.
3. function be passed as parameter. The function that is being passed as parameter is known as callback function.
4. `.classList` is useful for updating class on HTML element.
5. `.addEventListener` is used to listen to event on HTML element, such as click.

Do It

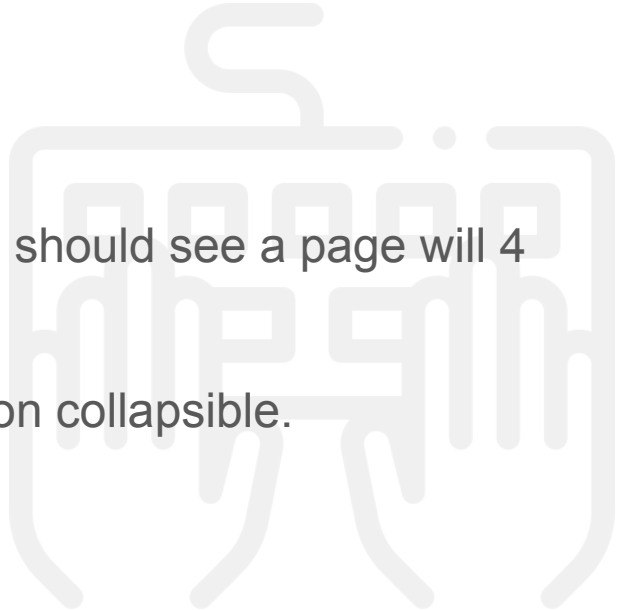
In your **`exercise.js`** file, rewrite the following function declaration as an anonymous function expression and arrow function

```
function getRandomInt(min, max) {
  const minValue = Math.ceil(min);
  const maxValue = Math.floor(max);
  return Math.floor(Math.random() * (maxValue - minValue) + minValue);
}
```

Do It

Go to Terms of Use (`/terms-of-use`) page, and you should see a page will 4 sections.

Write code in **`terms-of-use.js`** to make each section collapsible.

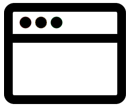


Chapter 5: Equality & Flow Control

Demo

Visit `/tax-calculator-v2` in preview.

file: **`public/javascripts/tax-calculator.js`**



Equal, Double Equal, Triple Equal

Single equal sign (“=”) is value assignment.

Double equal sign (“==”) checks for equality (with possible cohesion).

Triple equal sign (“===”) checks for equality and types.

The opposite of “==” is “!=”.

The opposite of “===” is “!==”.

```
1 == '1' // true
1 === '1' // false
```

JS

Two objects variables only equals when they are the same object.

```
let person1 = {
    name: 'Taylor'
};

let person2 = {
    name: 'Taylor'
};

console.log(person1 === person2); // false

let person3 = person1;
console.log(person1 === person3); // true
```

JS

Question

What is the result for each of the following comparisons?

```
'' == '0'
0 == ''
0 == '0'
false == 'false'
false == '0'
false == undefined
false == null
null == undefined
```

JS

Question

What is the result for each of the following comparisons?

```
'' == '0'           // false
0 == ''             // true
0 == '0'            // true
false == 'false'    // false
false == '0'        // true
false == undefined  // false
false == null       // false
null == undefined   // true
```

Double equal behavior is surprising sometimes.

JS

Equal, Double Equal, Triple Equal

Single equal sign (“=”) is value assignment.

Double equal sign (“==”) checks for equality (with possible cohesion).

Triple equal sign (“===”) checks for equality and types.

Prefer triple equal over double equal.

The opposite of “==” is “!=”.

The opposite of “===” is “!==”.

JS

if: do something conditionally

```
if (condition) {
  // do something if condition is true
}
```

JS

can have else or if else

```
if (condition) {
  // do something if condition is true
} else {
  // do something else if condition is false
}
```

```
if (condition1) {
  // do something if condition1 is true
} else if (condition2) {
  // do something else if condition1 is false but condition2 is true
} else {
  // do something if both condition1 and condition2 are both false
}
```

JS

```
function logIfLargerThan50(num) {
    if (num > 50) {
        console.log(num);
    }
}
```

expression that evaluates to boolean

```
function getAbsoluteValue(val) {
    let result;

    const isValPositive = val > 0;

    if (isValPositive) {
        result = val;
    } else {
        result = -val;
    }

    return result;
}
```

a boolean value

```
function logIfAbsoluteLargerThan50(val) {
  const isAbsoluteLargerThan50 = (num) => getAbsoluteValue(num) > 50;

  if (isAbsoluteLargerThan50(val)) {
    console.log(val);
  }
}
```

JS

Coercion will be applied to make the condition to become boolean.

```
function logString(str) {
  if (str) {
    console.log(`Provided string is ${str}`);
  }
}

logString('Hello');
logString('');
logString(null);
```

JS

function return can be used to skip remaining statements

```
function logString(str) {
  if (!str) {
    return;
  }
  console.log(`Provided string is ${str}`);
}

logString('Hello');
logString('');
logString(null);
```

JS

function return can be used to skip remaining statements

```
function logString(str) {
  if (!str) {
    return;
  }
  console.log(`Provided string is ${str}`);
}

logString('Hello');
logString('');
logString(null);
```

When **`return`** keyword is executed within a function, the following statements inside the function body will not run.

JS

use IIFE so you can use return any place

```
let value = -10;

(function() {
  if (value <= 0) {
    return;
  }

  if (value < 100) {
    console.log('Less than 100');
    return;
  }

  console.log('Perfect');
})();
```

IIFE (immediately-invoked function expression) is a single expression that does two things

- declare the function
- invoke the function

JS

use IIFE so you can use return any place

```
let value = -10;

(function() {
  if (value <= 0) {
    return;
  }

  if (value < 100) {
    console.log('Less than 100');
    return;
  }

  console.log('Perfect');
})();
```

IIFE (immediately-invoked function expression) is a single expression that does two things

- declare the function
- invoke the function

JS

use IIFE so you can use return any place

```
let value = -10;

(function() {
  if (value <= 0) {
    return;
  }

  if (value < 100) {
    console.log('Less than 100');
    return;
  }

  console.log('Perfect');
})();
```

IIFE (immediately-invoked function expression) is a single expression that does two things

- declare the function
- invoke the function

JS

use IIFE so you can use return any place

```
let value = -10;

(function() {
  if (value <= 0) {
    return;
  }

  if (value < 100) {
    console.log('Less than 100');
    return;
  }

  console.log('Perfect');
})();
```

IIFE (immediately-invoked function expression) is a single expression that does two things

- declare the function
- invoke the function

IIFE is one of the two places that you can use anonymous function. (The other place is as callback function.)

JS

Do It

In your **`exercise.js`** file, write a **`compute`** function that perform mathematical operations. It should accepts 3 parameters (**`operation`**, **`a`**, and **`b`**).

- **`operation`** can be ‘add’, ‘minus’, ‘multiple’, or ‘divide’
- **`a`**, and **`b`** are both numbers

You must use **ALL** of the following flow controls:

- if / else
- early return

If you can’t use all of the flow controls in single function, write multiple versions of the function.

ternary can be used to assign value conditionally

```
const getTaxAmount = salary => salary < 3000
  ? 0
  : salary * 0.1;
```

In case you have problem reading the code, this function is equivalent to

```
function getTaxAmount(salary) {
    return salary < 3000 ? 0 : salary * 0.1;
}
```

We can use it to run expression conditionally

```
const logStringTernary = str => str
    ? console.log(`string is ${str}`)
    : undefined;

logStringTernary('Hello');
logStringTernary('');
logStringTernary(undefined);
```

Due to && operator exit early if it gets false, we can use it to run code conditionally too. (1 of 2)

```
function isPositive(num) {
  console.log('running isPositive')
  return num > 0;
}

function isEven(num) {
  console.log('running isLarge');
  return num % 2 === 0;
}

const isPositiveEven = num => isPositive(num) && isEven(num);

console.log(isPositiveEven(1000)); // both isPositive and isEven is run
console.log(isPositiveEven(-2)); // only isPositive is run
```

Due to && operator exit early if it gets false, we can use it to run code conditionally too. (2 of 2)

```
const logString = str => str && console.log(`string is ${str}`);

logString('Hello');
logString('');
logString(null);
```

switch statement for many cases for single variable

```
function determinePrice(oriPrice, change, changeAmount) {
    let result;

    switch (change) {
    case 'increase':
      result = oriPrice + changeAmount;
      break;

    case 'decrease':
      result = oriPrice - changeAmount;
      break;

    default:
      result = oriPrice;
  }

  return result;
}

console.log(determinePrice(100, 'increase', 20));
console.log(determinePrice(100));
console.log(determinePrice(100, 'decrease', 5));
```

Do It

In your **`exercise.js`** file, write a **`compute`** function that perform mathematical operations. It should accepts 3 parameters (**`operation`**, **`a`**, and **`b`**).

- **`operation`** can be ‘add’, ‘minus’, ‘multiple’, or ‘divide’
- **`a`**, and **`b`** are both numbers

You must use **ALL** of the following flow controls:

- ternary
- switch statement

If you can’t use all of the flow controls in single function, write multiple versions of the function.

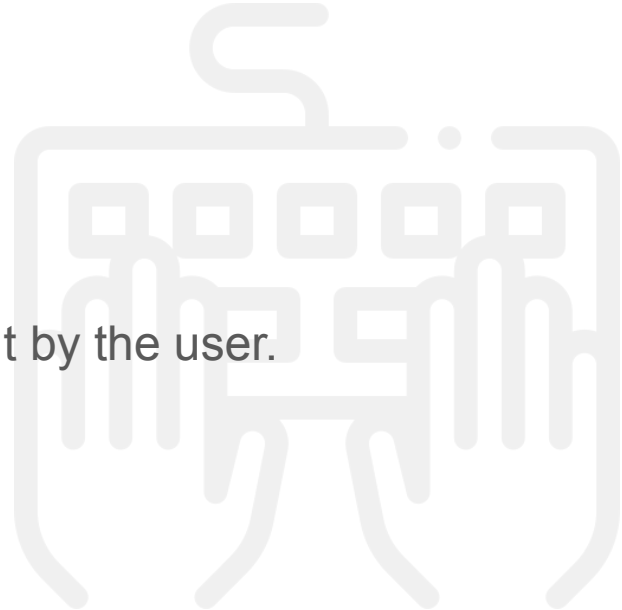
Recap

1. Use triple equal for equality check.
2. We can control the flow of code using:
    a. if/else
    b. return
    c. && operator
    d. ternary
    e. switch

Do It

Go to Bill Splitter v2 (`/bill-splitter-v2`) page.

Write code in **`bill-calculator.js`** to validate input by the user.



Take Home Homework (Day 1)

Visit “About Us” page (`/about-us`) and scroll to “Our vision” section.

Add code in `public/javascripts/video.js` so that:

1. When “Watch video” button when clicked, the video will plays and the button text will be changed to “Pause video”.
2. When the button is clicked again, the video will paused and the button will be changed back to “Watch video”.

Reference:

- Learn about playing/pausing video loaded with `<video>` HTML element in this article.

Agenda for today

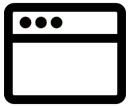
- Chapter 6: Object and Array
- =🌭=🍔= Lunch BREAK =🥗=🍜=
- Chapter 7: Scope
- Chapter 8: Async Programming
- Bonus Chapter: Destructuring
- Homework / Project Help

Chapter 6: Object and Array

Demo

Visit FAQ page (`/faq`) in preview.

file: **`public/javascripts/faq.js`**



We can use square notation to get property value as well.

```
const person = {
  name: 'Peter',
  age: 6,
  'secret-code': 'abcd',
};

console.log(person.name);
console.log(person['name']);
console.log(person['secret-code']);
```

JS

Object property value can be function too. We usually call a property whose value is function as method.

```
const dog = {
  name: 'Rocky',
  bark: function () {
    console.log('woff');
  },
  cry() {
    console.log('wuuu....');
  },
  sayMyName: function () {
    console.log(`My name is ${this.name}`);
  },
};
```

bark, **cry**, **sayMyName** are all methods of dog object.

The difference of method compared to typical function is in method, **this** refers to the object.

JS

We can use **`class`** to create similar object.

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

  sayMyName() {
    console.log(`I am ${this.name}`);
  }
}

const cat = new Animal('kitty');
const mouse = new Animal('jerry');

cat.sayMyName(); // I am kitty
mouse.sayMyName(); // I am jerry
```

We can use **`class`** to create similar object.

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

  sayMyName() {
    console.log(`I am ${this.name}`);
  }
}

const cat = new Animal('kitty');
const mouse = new Animal('jerry');

cat.sayMyName(); // I am kitty
mouse.sayMyName(); // I am jerry
```

`constructor` is a special method that will be called when the class is called with **`new`** keyword.

We can use **`class`** to create similar object.

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

  sayMyName() {
    console.log(`I am ${this.name}`);
  }
}

const cat = new Animal('kitty');
const mouse = new Animal('jerry');

cat.sayMyName(); // I am kitty
mouse.sayMyName(); // I am jerry
```

`constructor` is a special method that will be called when the class is called with **`new`** keyword.

Each **`animal`** has its own name.

There are a lot of nuance of using class due to **`this`** value depends on how you call it.

```
const sayMouseName = mouse.sayMyName;

sayMouseName(); // Error
```

It’s common to use **`class`** provided by JavaScript

```
const now = new Date();
console.log(now.getFullYear());
```

Arrays are list-like objects.

We can declare array by using bracket (`[]`) or **`Array`** constructor.

```
const days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'];
const numbers = [1,2,3];
const emptyList = [];
const anotherEmptyList = new Array();
```

JS

Getting array item

```
const firstDay = days[0]; // 'Mon'

const lastDay = days[days.length - 1]; // 'Sun'

const eighthDay = days[7]; // undefined
```

Use bracket notation to get the item. (Item index starts from 0)

array.length returns us the total number of items in the array, thus we can use it to get last item.

When trying to get item of index larger than the length of the array, you will get **`undefined`**.

JS

How to know if a variable is an Array

```
const anyObject = {};

console.log(typeof days); // 'object'
console.log(typeof anyObject); // 'object'
console.log(Array.isArray(days)); // true
console.log(Array.isArray(anyObject)); // false
```

Array is an object too.

`Array.isArray` can used to check if an item is an Array.

[].push, [].pop

```
const items = [1,2,3,4];
items.push(5);
console.log(items); // [1,2,3,4,5]

const removed = items.pop();
console.log(items); // [1,2,3,4]
console.log(removed); // 5
```

JS

[].join, ''.split()

```
const persons = ['Eric', 'Emily', 'Muthu'];

const allNames = persons.join(', '); // 'Eric, Emily, Muthu'

const splittedNames = allNames.split(', ');
console.log(splittedNames); // ['Eric', 'Emily', 'Muthu']
```

JS

Use for loop to iterate through all items

```
for (let i = 0; i < persons.length; i++) {
  const item = persons[i];
  console.log(item);
}

for (const item of persons) {
  console.log(item);
}
```

This is usually known as **`for...of`** loop to differentiate with the first type.

JS

Do It

In your **`array-exercise.js`** file,

1. create a **`doubleItem`** function that accepts an array of number and returns an array of equal length but with all of its item double, e.g. **`doubleItem([2,1,3])`** returns **`[4,2,6]`**.
2. create an **`invertSign`** function that accepts an array of number and returns an array of equal length but with all of its item sign inverted, e.g. **`invertSign([2,1,3])`** returns **`[-2,-1,-3]`**.

Possible Answer for **`doubleItem`**

```
const doubleItem = (numbers) => {
  const result = [];
  for (let index = 0; index < numbers.length; index++) {
    const element = numbers[index];
    result.push(element * 2);
  }
  return result;
};
```


JS

Possible Answer for **`doubleItem`**

```
const doubleItem = (numbers) => {
  const result = [];
  for (let index = 0; index < numbers.length; index++) {
    const element = numbers[index];
    result.push(element * 2);
  }
  return result;
};
```


This part is the most important part of this function.x

JS

A more generic **`mapItem`** function using callback function

```
const mapItem = (array, mapFn) => {
  const result = [];
  for (let index = 0; index < array.length; index++) {
    const element = array[index];
    result.push(mapFn(element));
  }
  return result;
}

// how to use
mapItem([2, 1, 3], item => item * 2); // [4, 2, 6]
mapItem([2, 1, 3], item => -item); // [-2, -1, -3]
```

We use the callback function to decide how to transform the item

Array has methods for common array operations.

Operation	Example	Array methods
Convert all item using similar transformation	Get the names of all customers in data	`.map`
Filter items based on certain condition	Get all the customers that are having income larger than 10K	`.filter`
Do something for each element	For each customer in the data, append an item in HTML	`.forEach`
Summarize all items by combining them	Get the sum of all customers	`.reduce`

`[].map(callback)`

```
const mapResult = [1, 2, 3, 4].map((x) => x * 2);
console.log(mapResult); // 2,4,6,8

const students = [
  { firstName: 'Bill', lastName: 'Gates', age: 55 },
  { firstName: 'Michael', lastName: 'Jackson', age: 80 },
  { firstName: 'Tiger', lastName: 'Woods', age: 52 },
  { firstName: 'Steve', lastName: 'Jobs', age: 55 },
];
const names = students.map((student) => `${student.firstName} ${student.lastName}`);
console.log(names); // ['Bill Gates', 'Michael Jackson', 'Tiger Woods', 'Steve Jobs']
```

Use `.map` method to transform the item. The callback function will receives the item as its parameter and should returns the transformed item.

JS

the callback passed to `.map` receives index as second parameter, which can be used or ignored

```
const numberList = [1, 2, 3];
const doubleAfterSecond = numberList.map((num, index) =>
  index >= 2 ? num * 2 : num
);
console.log(doubleAfterSecond); // [1,2,6]
```

Do It

In **`array-exercise.js`** file, write the code to get the list of id numbers.

```
const participants = [
  { name: 'Abu', idNumber: 'S7282395H', gender: 'male' },
  { name: 'Mary', idNumber: 'T4689018Z', gender: 'female' },
  { name: 'Suzi', idNumber: 'G5512873T', gender: 'female' },
  { name: 'T Chakra', idNumber: 'T8198514B', gender: 'male' }
];

// TODO: assign idNumber of all participants to idNumbers
const idNumbers = [];
console.log(idNumbers); // ['S7282395H', 'T4689018Z', 'G5512873T', 'T8198514B']
```

JS

`[].filter(callback)`

`.filter` allow us to remove item based on some condition.

```
const positiveNumbers = [1, -1000, 20, -0.3].filter((n) => n > 0);
console.log(positiveNumbers);

const celebrities = [
  { name: 'Jack Neo', isSingaporean: true },
  { name: 'Susanto', isSingaporean: false },
  { name: 'John Smith', isSingaporean: false },
  { name: 'Lee Hsien Loong', isSingaporean: true },
];
const singaporean = celebrities.filter((person) => person.isSingaporean);
console.log(singaporean);
```

The callback function receives the element as parameter and should return **`true`** to keep the item or **`false`** to exclude the item.

JS

callback passed to `.filter` also received index and the array as parameter

```
const prefilterNumbers = [1, 2, 3, 4, 5, 6];
const afterFourth = prefilterNumbers.filter((num, index) => index >= 4);
// this will keep fifth items onwards
console.log(afterFourth); // [5, 6];

const half = numbers.filter(
  (num, index, allNumbers) => index >= Math.round(allNumbers.length / 2)
); // this will keep second half of array
console.log(half); // [4, 5, 6]
```

We can chain `.filter` and `.map`

```
const numbers = [1, 2, 3, 4, 5];

numbers
.filter((n) => n % 2 === 0)
.map((n) => n * 2);
```

`[].forEach(callback)`

```
persons.forEach((item) => {
  console.log('in forEach', item);
});
```

`.forEach` is used to run arbitrary operation for each item.

JS

`[].reduce(callback)`

```
const total = [1, 3, 2, 2].reduce((prevResult, item) =>  prevResult + item, 0);
console.log(total); // 8
```

`.reduce` allows us to combine all values in an array into single value

The callback function receives `prevResult` and `item` as parameters and should returns a result. If there are more item, the result will be passed as `prevResult` parameter, else the returned result will become the final result of the `.reduce` method.

For first item, prevResult will be the second parameter provided to `.reduce` method.

JS

[].reduce(callback)

```
const total = [1, 3, 2, 2].reduce((prevResult, item) =>  prevResult + item, 0);
console.log(total); // 8
```

prevResult	item	result (prevResult + item)
0	1	
	3	
	2	
	2	

JS

[].reduce(callback)

```
const total = [1, 3, 2, 2].reduce((prevResult, item) =>  prevResult + item, 0);
console.log(total); // 8
```

prevResult	item	result (prevResult + item)
0	1	1
1	3	
	2	
	2	

Result returned with previous item will become **prevResult** for next item

JS

[].reduce(callback)

```
const total = [1, 3, 2, 2].reduce((prevResult, item) =>  prevResult + item, 0);
console.log(total); // 8
```

prevResult	item	result (prevResult + item)
0	1	1
1	3	**4**
4	2	
	2	

JS

[].reduce(callback)

```
const total = [1, 3, 2, 2].reduce((prevResult, item) =>  prevResult + item, 0);
console.log(total); // 8
```

prevResult	item	result (prevResult + item)
0	1	1
1	3	4
4	2	**6**
6	2	

JS

[].reduce(callback)

```
const total = [1, 3, 2, 2].reduce((prevResult, item) =>  prevResult + item, 0);
console.log(total); // 8
```

prevResult	item	result (prevResult + item)
0	1	1
1	3	4
4	2	6
6	2	**8**

Because there are no more item, so this will be the final result.

JS

Do It

In **`array-exercise.js`**, given the array below

```
const numbers = [3, -2, 1000, 201, 50, 100, 33, 50, -21];
```

- Get the result of the multiplications of all numbers using Array.reduce.
- Get the result of the multiplications of all numbers using **`for...of`** loop.

JS

We can use **`Array.from`** to convert Array-like object to Array

```
const accordionBtns = Array.from(document.querySelectorAll('.js-accordion'));
```

Array of HTMLElement

This returns **`NodeList`**, which is an object that is Array-like (it has **`length`** property and we can access HTMLElement using index), but it's not an **`Array`** (no array methods).

Do It

In **`array-exercise.js`**, given the array below

```
const numbers = [3, -2, 1000, 201, 50, 100, 33, 50, -21];
```

- Get all the positive numbers (above or equal to 0)
- Get all the odd numbers
- Get all the number that is larger than the number before it (exclude the first number)

Do It

In **`array-exercise.js`**, given the array below

```
const participants = [
  { name: 'Abu', idNumber: 'S7282395H', gender: 'male' },
  { name: 'Mary', idNumber: 'T4689018Z', gender: 'female' },
  { name: 'Suzi', idNumber: 'G5512873T', gender: 'female' },
  { name: 'T Chakra', idNumber: 'T8198514B', gender: 'male' }
];
```

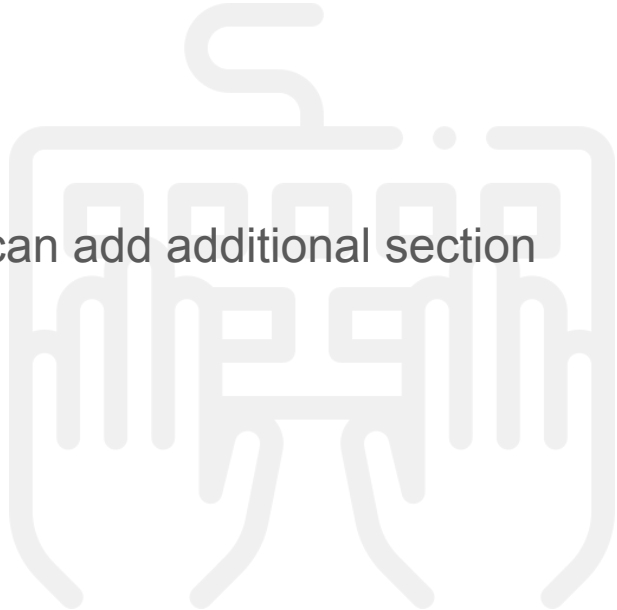
- Get the names for all the participants
- Get the id number for female participants only

Recap

1. Bracket notation can be used to get/set object property value.
2. Function assigned to object property is known as **`method`**.
3. We can create similar object by using **`class`** (invoked with **`new`** keyword).
4. Array is list-like object that provide many useful methods when working with list of data.
5. **`[].map`** is used to transform element.
6. **`[].filter`** is used to filter element
7. **`[].forEach`** is to perform action for each element
8. **`[].reduce`** is to combine all element into single value
9. **`Array.from`** is used to convert Array-like object (e.g. **`NodeList`**) to **`Array`**.

Do It

Refactor `terms-of-use.js` so that content creator can add additional section without changing your script.



Chapter 7: Scope

Demo

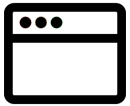
Visit Broken Tax Calculator (`/tax-calculator-broken`).

The html (**`public/tax-calculator-broken.html`**) is almost the same with Tax Calculator (**`public/tax-calculator.html`**), except the following part

tax-calculator-broken.html

```
</footer>
<script src="javascripts/helpers.js"></script>
<script src="javascripts/footer.js"></script>
<script src="javascripts/calculator.js"></script>
<script>
  var calculateTax = 5;
</script>
<script>
  const calculateBtn = document.querySelector('#calculateBtn');
  calculateBtn.addEventListener('click', calculateTax);
</script>
```

With this additional block, the page is broken now. **Why?**



Scope is where a variable can be accessed.

There are 3 types of scopes that we will discuss:

1. global scope
2. function scope
3. block scope

Global scope

By default, when you declare a variable or function in your `.js` file, those will be declared at global scope.

In global scope means that the variable/function can be access **everywhere**, even other files.

Global scope allows us use code from other .js file in a html file.

tax-calculator.html

```
<script src="javascripts/helpers.js"></script>
<script src="javascripts/footer.js"></script>
<script src="javascripts/calculator.js"></script>
<script>
  const calculateBtn = document.querySelector('#calculateBtn');
  calculateBtn.addEventListener('click', calculateTax);
</script>
```

We can use **`calculateTax`** function in **`calculator.js`**, thanks to global scope.

Global scope is convenient, but it makes our code fragile.

tax-calculator-broken.html

```
</footer>
<script src="javascripts/helpers.js"></script>
<script src="javascripts/footer.js"></script>
<script src="javascripts/calculator.js"></script>
<script>
  var calculateTax = 5;
</script>
<script>
  const calculateBtn = document.querySelector('#calculateBtn');
  calculateBtn.addEventListener('click', calculateTax);
</script>
</body>
```

Because the code relies on **`calculateTax`** on global scope, this means someone else may accidentally overwrite **`calculateTax`** and breaks the code.

Function scope

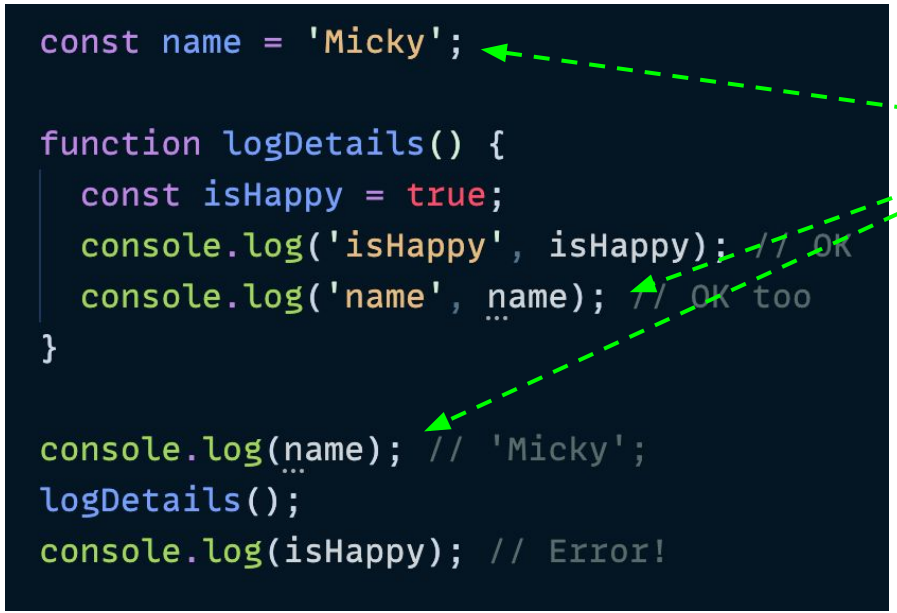
Variable declared within a function will be scoped within the function.

```
const name = 'Micky';

function logDetails() {
  const isHappy = true;
  console.log('isHappy', isHappy); // OK
  console.log('name', name); // OK too
}

console.log(name); // 'Micky';
logDetails();
console.log(isHappy); // Error!
```

JS

Function scope

Variable declared within a function will be scoped within the function.



Name is declared with global scope, therefore it is accessible everywhere

JS

Function scope

Variable declared within a function will be scoped within the function.

```
const name = 'Micky';

function logDetails() {
  const isHappy = true;
  console.log('isHappy', isHappy); // OK
  console.log('name', name); // OK too
}

console.log(name); // 'Micky';
logDetails();
console.log(isHappy); // Error!
```

```
Uncaught ReferenceError: isHappy is not defined
    at <anonymous>:1:13
```

`isHappy` is declared with function scope, therefore it is only accessible inside the **`logDetails`** function.

JS

We can use IIFE to “protect” our variable from read/edit.

file: **`public/javascripts/audio.js`**

```
const happyAudio = document.querySelector('#happy-audio');
const happyBtn = document.querySelector('#audio-btn');
const audioLabel = document.querySelector('#audio-label');

let isPlaying = false;

happyBtn.addEventListener('click', () => {
  if (isPlaying) {
    happyAudio.pause();
    happyBtn.innerHTML = 'Play';
    audioLabel.classList.remove('animate-pulse');
    isPlaying = false;
  } else {
    happyAudio.play();
    happyBtn.innerHTML = 'Pause';
    audioLabel.classList.add('animate-pulse');
    isPlaying = true;
  }
});
```

All js in the same page has access to **`isPlaying`** and may mess with our logic.

We can use IIFE to “protect” our variable from read/edit.

javascripts/audio.js

```
const happyAudio = document.querySelector('#happy-audio');
const happyBtn = document.querySelector('#audio-btn');
const audioLabel = document.querySelector('#audio-label');

let isPlaying = false;

happyBtn.addEventListener('click', () => {
  if (isPlaying) {
    happyAudio.pause();
    happyBtn.innerHTML = 'Play';
    audioLabel.classList.remove('animate-pulse');
    isPlaying = false;
  } else {
    happyAudio.play();
    happyBtn.innerHTML = 'Pause';
    audioLabel.classList.add('animate-pulse');
    isPlaying = true;
  }
});
```

All js in the same page has access to **`isPlaying`** and may mess with our logic.

```
(function() {

  const happyAudio = document.querySelector('#happy-audio');
  const happyBtn = document.querySelector('#audio-btn');
  const audioLabel = document.querySelector('#audio-label');

  let isPlaying = false;

  happyBtn.addEventListener('click', () => {
    if (isPlaying) {
      happyAudio.pause();
      happyBtn.innerHTML = 'Play';
      audioLabel.classList.remove('animate-pulse');
      isPlaying = false;
    } else {
      happyAudio.play();
      happyBtn.innerHTML = 'Pause';
      audioLabel.classList.add('animate-pulse');
      isPlaying = true;
    }
  });

})();
```

Wrapping all code in IIFE to protect our variables from read and update.

JS

Block scope

Variable declared within the curly braces block (`{}`) will be scoped within the block.

```
function getGreeting(name, gender) {
  if (gender === 'male') {
    let designation = 'Mr.';
  } else if (gender === 'female') {
    let designation = 'Ms.';
  }

  return `${designation} ${name}`;
}
```

```
▸Uncaught ReferenceError: designation is not
defined
    at getGreeting (<anonymous>:8:3)
    at <anonymous>:1:1
```

Recap

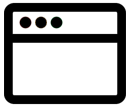
1. When declaring variable in .js file, variable is accessible globally (everywhere). The variable is said to be **declared at global scope**.
2. When variable is declared in function, the variable is accessible within the function.The variable is said to be **declared at function scope**.
3. When variable is declare within curly braces (`{ }`), the variable is only accessible within the curly braces. The variable is said to be **declared at block scope**.

Chapter 8: Async Programming

Demo

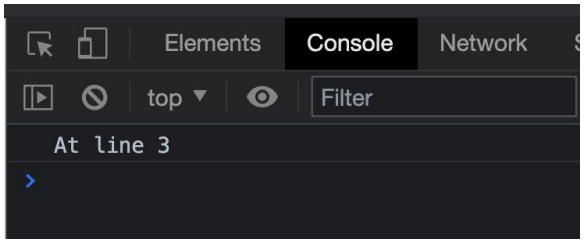
Visit home page (`/`).

file: **`public/javascripts/signup.js`**



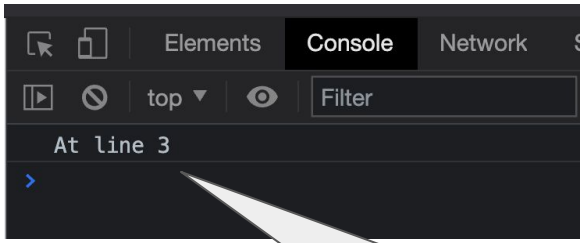
`setTimeout` is a browser API to allows you to perform some action after some delay.

```
1
2  setTimeout(() => {
3    console.log('At line 3');
4  }, 1000);
5
```



`setTimeout` is a browser API to allows you to perform some action after some delay.

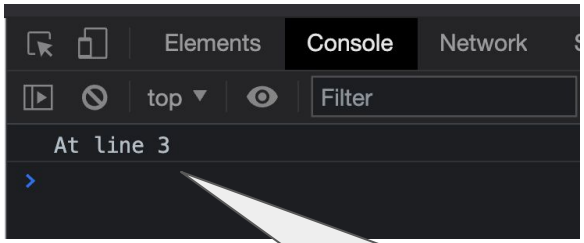
```
1
2  setTimeout(() => {
3    console.log('At line 3');
4  }, 1000);
5
```



If you refresh the browser, you will see this log appears after 1 second.

`setTimeout` is a browser API to allows you to perform some action after some delay.

```
1
2  setTimeout(() => {
3    console.log('At line 3');
4  }, 1000);
5
```



If you refresh the browser, you will see this log appears after 1 second.

```
setTimeout(callback, numOfMilliSeconds);
```

`setTimeout` will call the callback function when **`numOfMilliSeconds`** has passed after **`setTimeout`** is called.

`setTimeout` returns timer id (a number), which can be used to cancel the action if it has not been called.

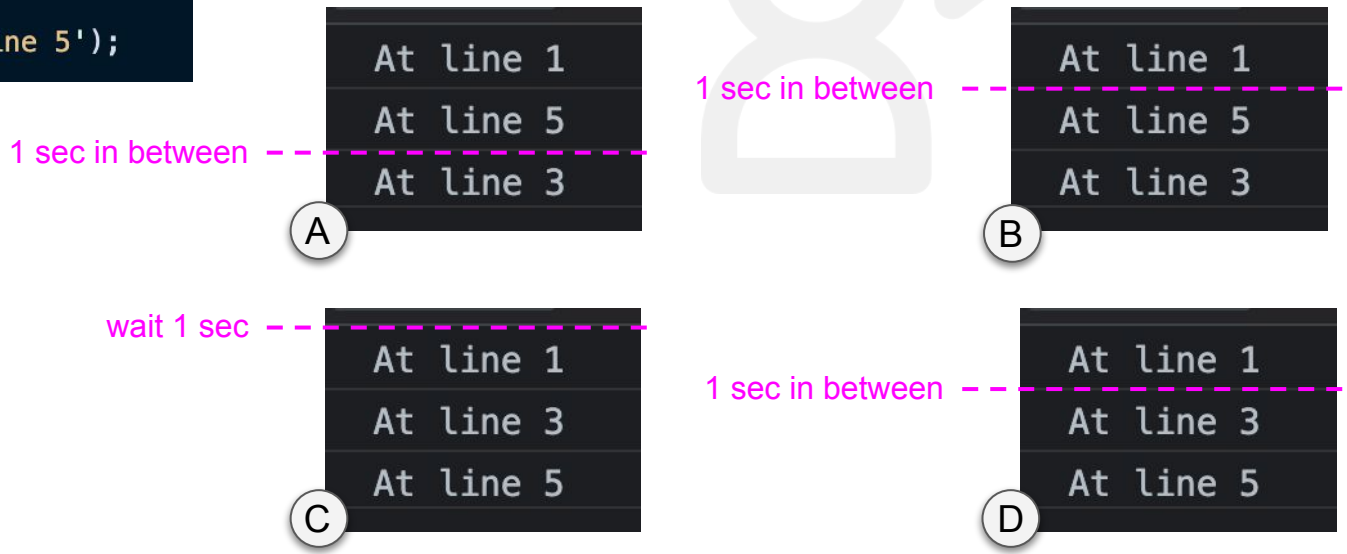
You can pass the timer id to **`clearTimeout`** (another browser API) to cancel the action if it has not been called.

```
const timerId = setTimeout(() => {/* do thing */}, 1000);

clearTimeout(timerId); // cancel the timeout if the callback has not invoked
```

Question

What will be logged in console when the following code is run?

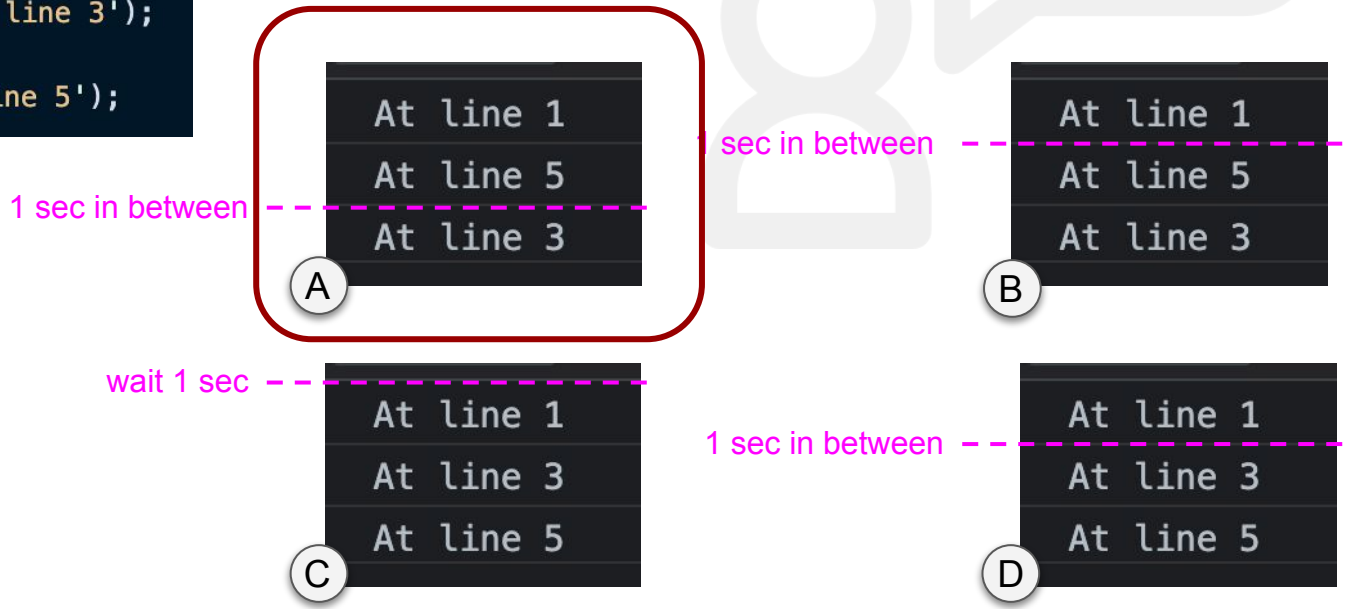
```
1  console.log('At line 1');
2  setTimeout(() => {
3    console.log('At line 3');
4  }, 1000);
5  console.log('At line 5');
```



Question

What will be logged in console when the following code is run?

```
1  console.log('At line 1');
2  setTimeout(() => {
3    console.log('At line 3');
4  }, 1000);
5  console.log('At line 5');
```

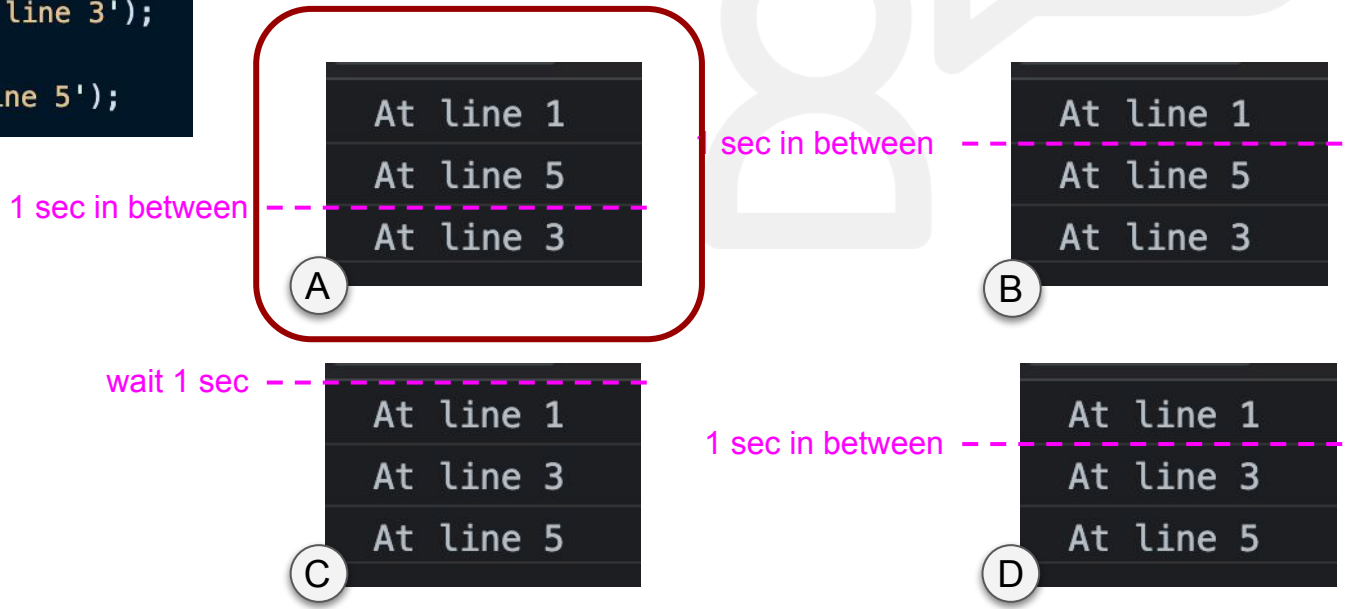


Question

What will be logged in console when

```
1  console.log('At line 1');
2  setTimeout(() => {
3    console.log('At line 3');
4  }, 1000);
5  console.log('At line 5');
```

`setTimeout` does not stop JS from continue running, it just "schedule" the callback to be run later and then continue running the next line.



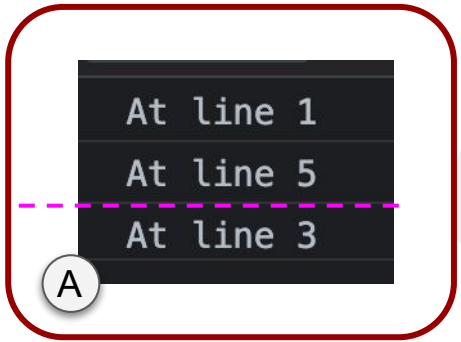
Question

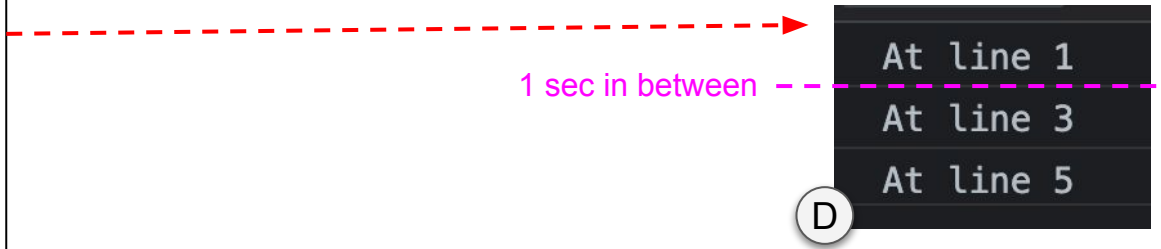
What will be logged in console when

```
1  console.log('At line 1');
2  setTimeout(() => {
3    console.log('At line 3');
4  }, 1000);
5  console.log('At line 5');
```

`setTimeout` does not stop JS from continue running, it just "schedule" the callback to be run later and then continue running the next line.

How do we achieve this (running line 5 after line 3 is run)?





Question

What will be logged in console when

```
1  console.log('At line 1');
2  setTimeout(() => {
3    console.log('At line 3');
4  }, 1000);
5  console.log('At line 5');
```

`setTimeout` does not stop JS from continue running, it just "schedule" the callback to be run later and then continue running the next line.

```
At line 1
At line 5
At line 3
```

A

How do we achieve this (running line 5 after line 3 is run)?

```
1  console.log('At line 1');
2  setTimeout(() => {
3    console.log('At line 3');
4    console.log('At line 5');
5  }, 1000);
```

Putting the code inside the callback is the way to have the code run later.

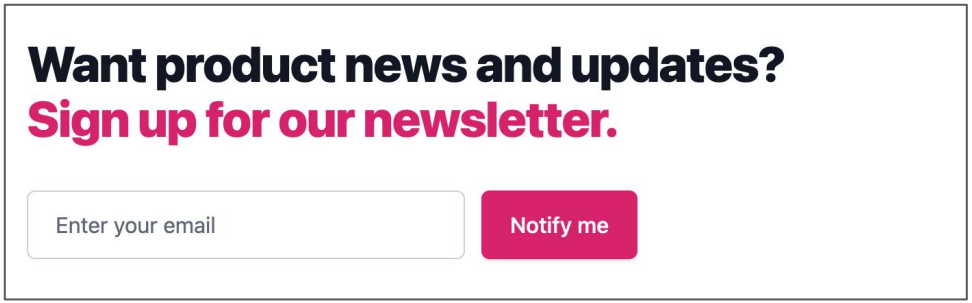
1 sec in between

```
At line 1
At line 3
At line 5
```

D

Do It

Visit https://codesandbox.io/s/intro-to-js-playground-0czx0, in the preview, click “About Us”, then scroll down to the signup newsletter section.



Try to fill up your email and click Notify me, and you will see a notification is shown and a message is logged when signup succeeds (after few seconds).

Fix the code in `public/javascripts/subscribe.js` so that notification is only shown **after** the signup succeeds.

Demo available at: https://intro-to-js-playground.vercel.app/about-us

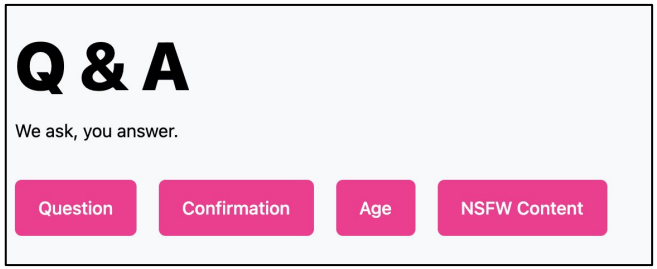
Do It

Write the code that performs the following steps:

1. log message “Start”
2. after 1 second, log message “1 sec passed”
3. 2 seconds after step 2, log message “another 2 sec passed”
4. 2 seconds after step 3, log message “End”

Do It

Visit https://codesandbox.io/s/intro-to-js-playground-0czx0, in the preview, click “Q & A” at the footer.



Click Question and Confirmation buttons and observe the behaviors. The code that make it works is in `public/javascripts/q-and-a.js`. (The code use `helpers.prompt` function defined in `helpers.js`, but you do not need to understand the code in `helpers.js` for this exercise, you just need to know **how to use it** by referring the code in `q-and-a.js`).

Add code in `public/javascripts/q-and.js` so to add functionalities to “Age” and “NSFW Content” buttons as described in the “TODO” comments in the file by using `helpers.prompt` function.

Demo available at: https://intro-to-js-playground.vercel.app/q-and-a

Async programming is the programming that involves operations that does not run instantly

```
(function setupSignup() {
  const signupBtn = document.querySelector('#signup-button');
  let timerId;
  signupBtn.addEventListener('click', function invokeSignup() {
    clearTimeout(timerId);
    const notification = document.querySelector('#signup-notification');
    notification.classList.remove('hidden');
    timerId = setTimeout(() => {
      notification.classList.add('hidden');
    }, 2000);
  });
})();
```

This run instantly when **`setupSignup`** is called.

Code inside **`invokeSignup`** will **not** run instantly when **`setupSignup`** is called.

JS

Async programming is the programming that involves operations that does not run instantly

```
(function setupSignup() {
  const signupBtn = document.querySelector('#signup-button');
  let timerId;
  signupBtn.addEventListener('click', function invokeSignup() {
    clearTimeout(timerId);
    const notification = document.querySelector('#signup-notification');
    notification.classList.remove('hidden');
    timerId = setTimeout(() => {
      notification.classList.add('hidden');
    }, 2000);
  });
})();
```

This run instantly when **`invokeSignup`** is called.

This code will **not** run instantly when **`invokeSignup`** is called. It will run after 2 seconds (2000ms)

JS

Talk: What the heck is the event loop anyway?

Link to video

Using callback for async programming is fine, but the code become hard to read when we need to run those functions sequentially

```
console.log('Start');

setTimeout(() => {
  console.log('1 sec passed');

  setTimeout(() => {
    console.log('another 2 sec passed');

    setTimeout(() => {
      console.log('End');
    }, 2000);
  }, 2000);
}, 1000);
```

This is famously known as callback hell.

Using callback for async programming is fine, but the code become hard to read when we need to run those functions sequentially

```
console.log('Start');

setTimeout(() => {
  console.log('1 sec passed');

  setTimeout(() => {
    console.log('another 2 sec passed');

    setTimeout(() => {
      console.log('End');
    }, 2000);
  }, 2000);
}, 1000);
```

This is famously known as callback hell.

There is another way of writing asynchronous code: **`Promise`**.

`Promise`: A Ticket to Future Value

You

Ask for burger

Crew

`Promise`: A Ticket to Future Value

You

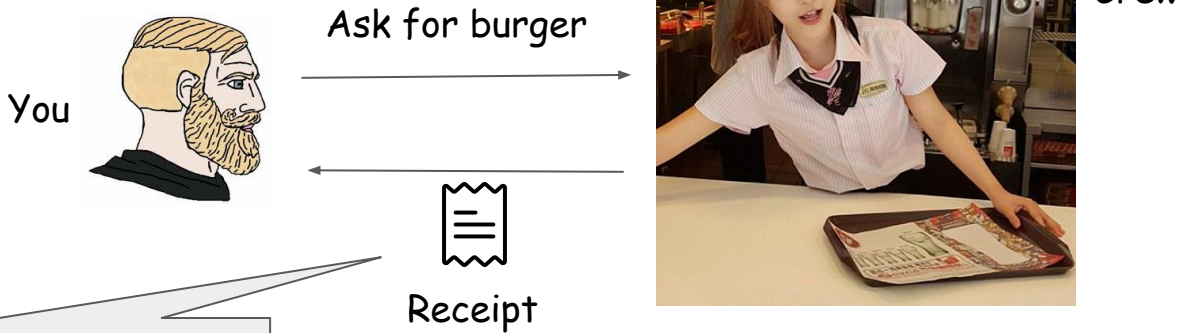
Ask for burger

Receipt

Crew

JS

`Promise`: A Ticket to Future Value



Receipt is not a burger, but it allows you to claim the burger when burger is ready in future. In other words, **receipt is a ticket that you get instantly to access a future value** (burger).

While holding the receipt, you can do something else (like continue scroll your phone).

JS

`Promise`: A Ticket to Future Value

Here is your burger...

Sorry sir, our machine is on fire...

Some time later...

JS

`Promise`: A Ticket to Future Value

Here is your burger...

Sorry sir, our machine is on fire...

Some time later...

Having the ticket does not ensure you get the value. It is possible that the future value cannot be fulfilled due to some issue.

JS

`Promise`: how to

```
const promise = new Promise((resolve, reject) => {
    // do some async operation,
    // when success, call resolve with the value
    // when error, call reject with the error
})

promise
.then((value) => {
    // do something with the value provided when success
})
.catch((err) => {
    // do something for the error
});
```

`Promise`: how to

```
const promise = new Promise((resolve, reject) => {
    // do some async operation,
    // when success, call resolve with the value
    // when error, call reject with the error
})

promise
.then((value) => {
    // do something with the value provided when success
})
.catch((err) => {
    // do something for the error
});
```

`Promise` is a **`class`** that you can used to create promise using the **`new`** keyword (like **`Date`**).

`Promise`: how to

```
const promise = new Promise((resolve, reject) => {
    // do some async operation,
    // when success, call resolve with the value
    // when error, call reject with the error
})

promise
.then((value) => {
    // do something with the value provided when success
})
.catch((err) => {
    // do something for the error
});
```

`Promise` is a **`class`** that you can used to create promise using the **`new`** keyword (like **`Date`**).

Pass a callback to **`.then`** method to handle success scenario.
Pass a callback to **`.catch`** method to handle error scenario.

`Promise`: how to

```
const promise = new Promise((resolve, reject) => {
    // do some async operation,
    // when success, call resolve with the value
    // when error, call reject with the error
})

promise
.then((value) => {
    // do something with the value provided when success
})
.catch((err) => {
    // do something for the error
});
```

`Promise` is a **`class`** that you can used to create promise using the **`new`** keyword (like **`Date`**).

When **`resolve`** in the Promise constructor is called, the callback passed to **`.then`** will be called.

`Promise`: how to

```
const promise = new Promise((resolve, reject) => {
    // do some async operation,
    // when success, call resolve with the value
    // when error, call reject with the error
})

promise
.then((value) => {
    // do something with the value provided when succes
})
.catch((err) => {
    // do something for the error
});
```

`Promise` is a **`class`** that you can used to create promise using the **`new`** keyword (like **`Date`**).

When **`resolve`** in the Promise constructor is called, the callback passed to **`.then`** will be called.

When **`reject`** in the Promise constructor is called, the callback passed to **`.catch`** will be called.

Example

```
function wait(ms) {
    return new Promise((resolve) => {
        setTimeout(() => resolve(ms / 1000), ms);
    })
}

wait(1000)
.then((value) => {
    console.log(`Called after ${value}s.`)
})
```

Example

`wait` will return a `Promise` instantly when called.

```
function wait(ms) {
    return new Promise((resolve) => {
        setTimeout(() => resolve(ms / 1000), ms);
    })
}

wait(1000)
.then((value) => {
    console.log(`Called after ${value}s.`)
})
```

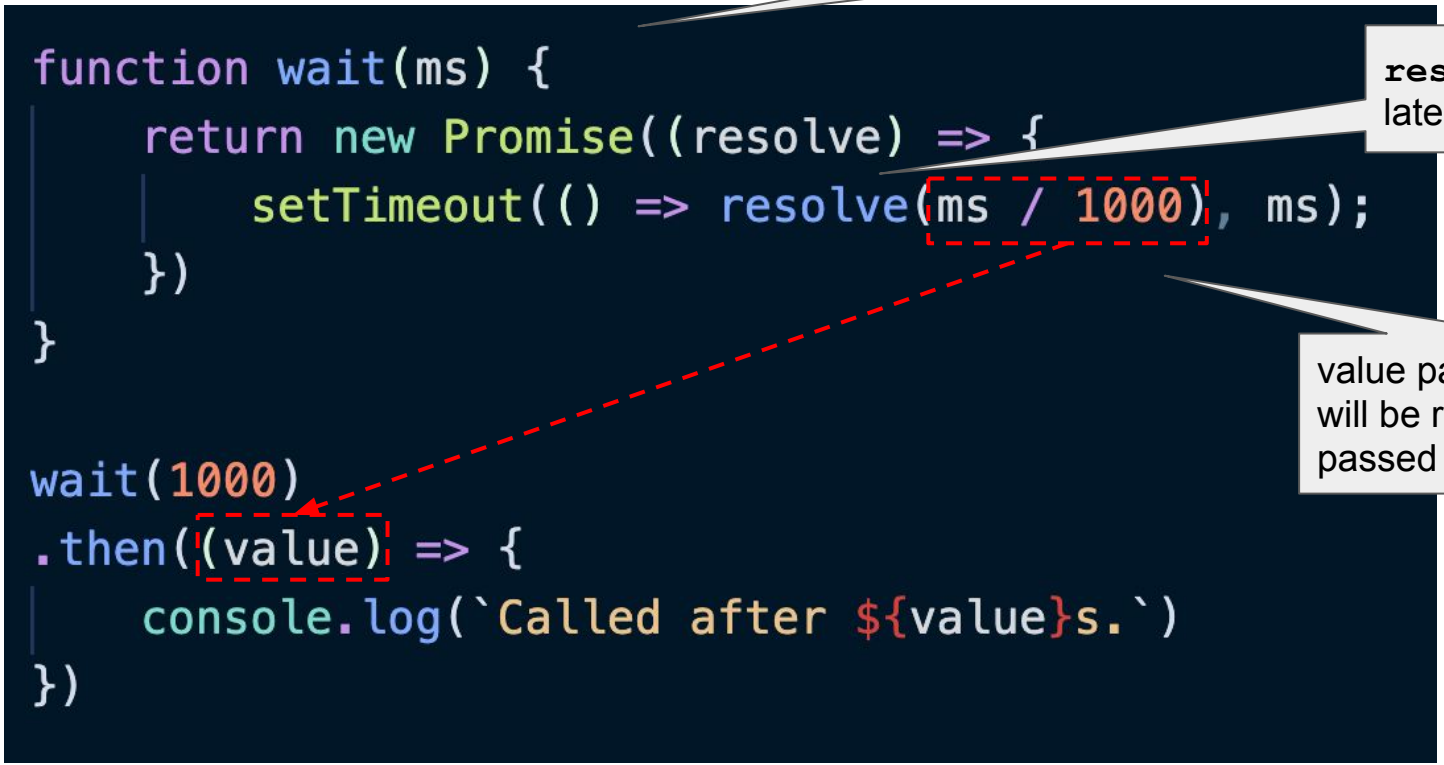
Example

`wait` will return a `Promise` instantly when called.

`resolve` will be called later.

```
function wait(ms) {
    return new Promise((resolve) => {
        setTimeout(() => resolve(ms / 1000), ms);
    })
}

wait(1000)
.then((value) => {
    console.log(`Called after ${value}s.`)
})
```

Example



`wait` will return a **`Promise`** instantly when called.

`resolve` will be called later.

value passed to **`resolve`** will be received by callback passed to **`then`**.

Do It

In `q-and-a.js`, create a `prompt` function that wraps `helpers.prompt` and returns a `Promise` when called. Update all the code from using `helpers.prompt` to use this `prompt` function.

Do It

In **`async-exercise.js`**, rewrite the following code using Promise:

```
function getServerTime(callback) {
  const xmlHttp = new XMLHttpRequest();
  xmlHttp.open('HEAD', window.location.href);
  xmlHttp.setRequestHeader('Content-Type', 'text/html');
  xmlHttp.addEventListener('load', () => {
    callback(xmlHttp.getResponseHeader('Date'));
  });
  xmlHttp.send('');
}

getServerTime((serverDate) => console.log(`Server date is ${serverDate}`));
```

. **`then`** method will returns a Promise to allow chaining.

```
wait(1000)
.then(() => {
  console.log('1 sec passed.');
})
.then(() => {
  console.log('Sometimes after 1 sec.');
});
```

Async operations can be run sequentially by chaining **`Promises`**

```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));

wait(1000)
.then(() => {
  console.log('1 sec passed.');
  return wait(2000);
})
.then(() => {
  console.log('3 sec passed.');
});
```

This function is rewritten with arrow function to be more concise.

Async operations can be run sequentially by chaining **`Promises`**

```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));

wait(1000)
.then(() => {
  console.log('1 sec passed.');
  return wait(2000);
})
.then(() => {
  console.log('3 sec passed.');
});
```

This function is rewritten with arrow function to be more concise.

This callback is run when the first Promise resolved (which is after 1 sec).

Async operations can be run sequentially by chaining **`Promise`**s

```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));

wait(1000)
.then(() => {
  console.log('1 sec passed.');
  return wait(2000);
})
.then(() => {
  console.log('3 sec passed.');
});
```

This function is rewritten with arrow function to be more concise.

This callback is run when the first Promise resolved (which is after 1 sec).

Because the previous callback return a promise, this callback will be invoked until that promise is resolved (which is another 2 sec later).

Do It

Write the code that performs the following steps using **`Promise`**:

1. log message “Start”
2. after 1 second, log message “1 sec passed”
3. 2 seconds after step 2, log message “another 2 sec passed”
4. 2 seconds after step 3, log message “End”

You may use the following **`wait`** function that wraps **`setTimeout`**:

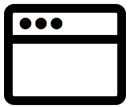
```
function wait(ms) {
  return new Promise((resolve) => {
    setTimeout(() => resolve(), ms);
  });
}
```

Demo

Visit pokemons page (`/pokemons`).

Visit `/api/pokemons` to see the response of pokemon api. (You can use site like http://jsonviewer.stack.hu/ to format the JSON result in a more readable way.)

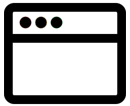
file: `public/javascripts/pokemons.js`



`fetch` is a browser API for us to get data from a URL

```
fetch('<url>')
.then(response => {
  const resultPromise = response.json();
  return resultPromise;
})
.then(result => {
  // do something with the result
});
```

Pass the URL to fetch the data. Usually you get this URL from your backend engineer or API documentation (example).

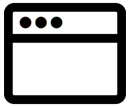


`fetch` is a browser API for us to get data from a URL

```
fetch('<url>')
.then(response => {
  const resultPromise = response.json();
  return resultPromise;
})
.then(result => {
  // do something with the result
});
```

Pass the URL to fetch the data. Usually you get this URL from your backend engineer or API documentation (example).

`fetch` returns a Promise which will resolve to a `Response` object, which has a `.json` method that will returns a Promise



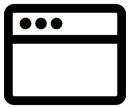
`fetch` is a browser API for us to get data from a URL

```
fetch('<url>')
.then(response => {
  const resultPromise = response.json();
  return resultPromise;
})
.then(result => {
  // do something with the result
});
```

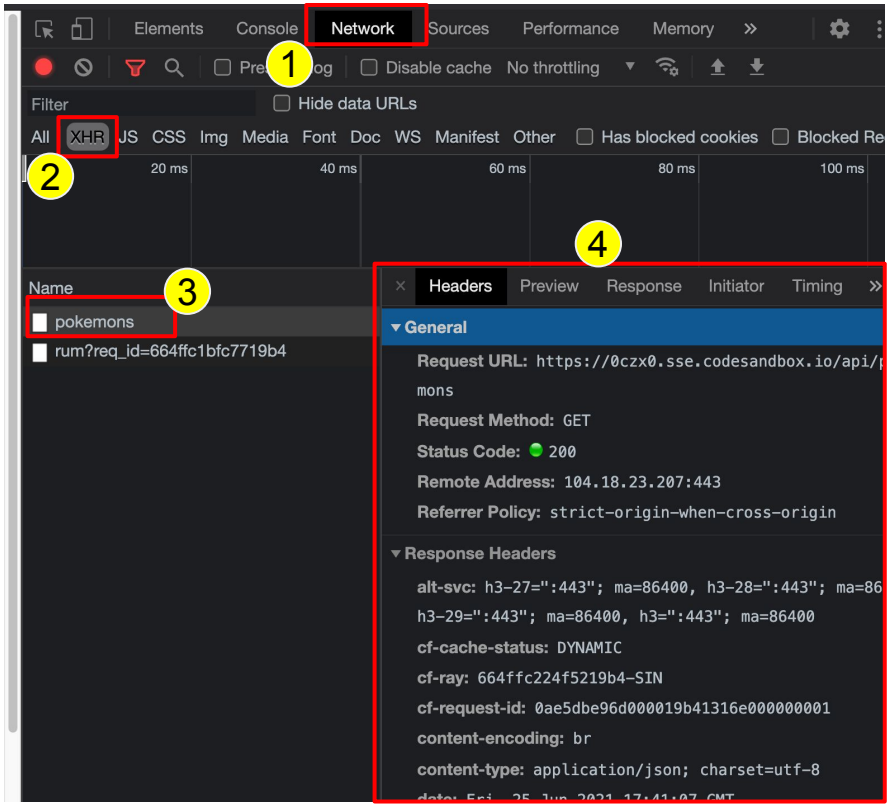
Pass the URL to fetch the data. Usually you get this URL from your backend engineer or API documentation (example).

`fetch` returns a Promise which will resolve to a `Response` object, which has a `.json` method that will returns a Promise

The Promise returned by the `.json` method will be resolved with the returned data in JSON format.



Inspecting network request



1. Click the Network Tab in your browser DevTools (open with F12)
2. Click XHR to filter the requests to data fetch (else it will show all the network requests, including getting the css files and js files)
3. Click on the requests that we made.
4. The details of the requests will be shown.

async function allow to make the code more readable by stopping the code execution in the function for async operation.

add **async** keyword before the function.

```
async function run() {
  await wait(1000);
  console.log('1 sec passed.');
  await wait(2000);
  console.log('3 sec passed.');
}
```

async function allow to make the code more readable by stopping the code execution in the function for async operation.

add **`async`** keyword before the function.

`await` keyword will stop the function until promise on its right is resolved.

```
async function run() {
  await wait(1000);
  console.log('1 sec passed.');
  await wait(2000);
  console.log('3 sec passed.');
}
```

async function allow to make the code more readable by stopping the code execution in the function for async operation.

add **async** keyword before the function.

await keyword will stop the function until promise on its right is resolved.

```
async function run() {
  await wait(1000);
  console.log('1 sec passed.');
  await wait(2000);
  console.log('3 sec passed.');
}
```

This is equivalent to the Promise version below:

```
function run() {
  wait(1000)
  .then(() => {
    console.log('1 sec passed.');
    return wait(2000);
  })
  .then(() => {
    console.log('3 sec passed.');
  });
}
```

Do It

In **`q-and-a.js`**, use **`async/await`** to remove the **`.then`** callback.

Do It

Write the code that performs the following steps using **`Promise`** and **`async/await`**:

1. log message “Start”
2. after 1 second, log message “1 sec passed”
3. 2 seconds after step 2, log message “another 2 sec passed”
4. 2 seconds after step 3, log message “End”

You may use the following **`wait`** function that wraps **`setTimeout`**:

```
function wait(ms) {
  return new Promise((resolve) => {
    setTimeout(() => resolve(), ms);
  });
}
```

Do It

In **`async-exercise.js`**, rewrite the following code using async function:

```
function wait(second) {
  return new Promise((resolve) => setTimeout(resolve), second * 1000);
}

function fetchPricingData() {
  return fetch('/api/pricing').then((res) => res.json());
}

function fetchDataThenWait() {
  return fetchPricingData().then((pricingData) => {
    return wait(1).then(() => pricingData);
  });
}
```

Recap

There are 3 ways to write asynchronous code in JS:

- callback function,
- `Promise`, and
- `async` function.

Take Home Homework (Day 2)

Visit “Pricing” page (`/pricing`).

Add code in `public/javascripts/pricing.js` so that:

1. The price displayed in the page will be according to the data retrieved from `/api/pricing`.
2. When the “Monthly billing” and “Yearly billing” button are clicked:
   a. the price unit will be updated (“/mo” for monthly and “/yr” for yearly).
   b. the price for each tier should be updated as well.

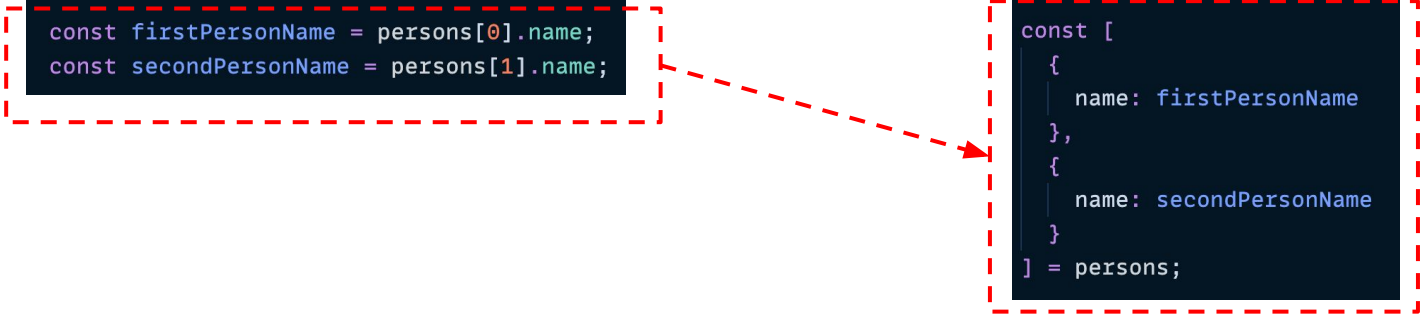
Reference:

- Learn about Array `.find` method in this article.

Bonus Chapter: Destructuring

Destructuring is a syntactical sugar that makes code more declarative.

```
const persons = [
  {
    name: 'Luke Skywalker',
    role: 'son'
  },
  {
    name: 'Darth Vader',
    role: 'father'
  }
];

const firstPersonName = persons[0].name;
const secondPersonName = persons[1].name;
```



```
const persons = [
  {
    name: 'Luke Skywalker',
    role: 'son'
  },
  {
    name: 'Darth Vader',
    role: 'father'
  }
];

const [
  {
    name: firstPersonName
  },
  {
    name: secondPersonName
  }
] = persons;
```

JS

Array destructuring allows us to get item from array

```
function getPersons() {
  return ['Ali', 'Ahkau', 'Muthu'];
}

const persons = getPersons();
const person1 = persons[0]; // 'Ali'
const person2 = persons[1]; // 'Ahkau'
const person3 = persons[2]; // 'Muthu'
```

```
function getPersons() {
  return ['Ali', 'Ahkau', 'Muthu'];
}

const persons = getPersons();
const [person1, person2, person3] = persons;
```

JS

Array destructuring allows us to get item from array

```
function getPersons() {
  return ['Ali', 'Ahkau', 'Muthu'];
}

const persons = getPersons();
const person1 = persons[0]; // 'Ali'
const person2 = persons[1]; // 'Ahkau'
const person3 = persons[2]; // 'Muthu'
```

```
function getPersons() {
  return ['Ali', 'Ahkau', 'Muthu'];
}

const persons = getPersons();
const [person1, person2, person3] = persons;
```

This may be confusing at first, as it looks like declaring an array.
To differentiate them, remember that

- when declaring array the square bracket (`[]`) is on the right of the equal (=) sign,
- when destructuring, the square bracket is on the left.

JS

Array destructuring allows us to get item from array

```
function getPersons() {
  return ['Ali', 'Ahkau', 'Muthu'];
}

const persons = getPersons();
const person1 = persons[0]; // 'Ali'
const person2 = persons[1]; // 'Ahkau'
const person3 = persons[2]; // 'Muthu'
```

```
function getPersons() {
  return ['Ali', 'Ahkau', 'Muthu'];
}

const persons = getPersons();
const [person1, person2, person3] = persons;
```

This may be confusing at first, as it looks like declaring an array.
To differentiate them, remember that

- when declaring array the square bracket (`[]`) is on the right of the = sign,
- when destructuring, the square bracket is on the left.

The great thing about destructuring is not that it's shorter, it is that the **code syntax shows the shape of the data visually**.

JS

We can ignore certain item by just not giving it a name.

```
function getPersons() {
  return ['Ali', 'Ahkau', 'Muthu'];
}

const persons = getPersons();
const person1 = persons[0]; // 'Ali'
const person3 = persons[2]; // 'Muthu'
```

```
function getPersons() {
  return ['Ali', 'Ahkau', 'Muthu'];
}

const persons = getPersons();
const [person1, , person3] = persons;
```

JS

Use the `...rest` syntax to group the remaining items.

```
function getPersons() {
  return ['Ali', 'Ahkau', 'Muthu', 'Siti', 'Lily'];
}

const persons = getPersons();
const person1 = persons[0]; // 'Ali'
const person2 = persons[1]; // 'Ahkau'
const others = persons.slice(2); // ['Muthu', 'Siti', 'Lily']
```

```
function getPersons() {
  return ['Ali', 'Ahkau', 'Muthu', 'Siti', 'Lily'];
}

const persons = getPersons();
const [person1, person2, ...others] = persons;
```

JS

Use the `...rest` syntax to group the remaining items.

```
function getPersons() {
  return ['Ali', 'Ahkau', 'Muthu', 'Siti', 'Lily'];
}

const persons = getPersons();
const person1 = persons[0]; // 'Ali'
const person2 = persons[1]; // 'Ahkau'
const others = persons.slice(2); // ['Muthu', 'Siti', 'Lily']
```

```
function getPersons() {
  return ['Ali', 'Ahkau', 'Muthu', 'Siti', 'Lily'];
}

const persons = getPersons();
const [person1, person2, ...others] = persons;
```

`.slice` is an Array method that can be used to clone array. The first parameter is what is the `startIndex` for items to clone, i.e. `.slice(2)` will clone the array from item at index 2.. Read more about it in this article.

JS

Object destructuring allows us to get property values.

```
function getPerson() {
  return {
    firstName: 'Malcolm',
    lastName: 'Kee',
    age: 29,
  };
}

const me = getPerson();
const myFirstName = me.firstName;
const myLastName = me.lastName;
const myAge = me.age;
```



```
function getPerson() {
  return {
    firstName: 'Malcolm',
    lastName: 'Kee',
    age: 29,
  };
}

const {
  firstName: myFirstName,
  lastName: myLastName,
  age: myAge
} = getPerson();
```

JS

Object destructuring allows us to get property values.

```
function getPerson() {
  return {
    firstName: 'Malcolm',
    lastName: 'Kee',
    age: 29,
  };
}

const me = getPerson();
const myFirstName = me.firstName;
const myLastName = me.lastName;
const myAge = me.age;
```

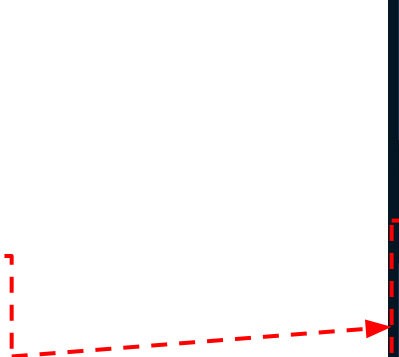
```
function getPerson() {
  return {
    firstName: 'Malcolm',
    lastName: 'Kee',
    age: 29,
  };
}

const {
  firstName: myFirstName,
  lastName: myLastName,
  age: myAge
} = getPerson();
```

This looks like creating object but it's not (just like array destructuring). On left hand side of equal sign is destructuring.

JS

Object destructuring will be shorter if the variable name is same as the property name.

```
function getPerson() {
  return {
    firstName: 'Malcolm',
    lastName: 'Kee',
    age: 29,
  };
}

const me = getPerson();
const firstName = me.firstName;
const lastName = me.lastName;
const age = me.age;
```



```
function getPerson() {
  return {
    firstName: 'Malcolm',
    lastName: 'Kee',
    age: 29,
  };
}

const {
  firstName,
  lastName,
  age
} = getPerson();
```

JS