Console.log(…) : Write … onto the log

String Methods & String Stuff

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

```
console.log('hello'.toUpperCase()); // Prints 'HELLO'
console.log('Hey'.startsWith('H')) // Prints true
console.log('      Remove whitespace     '.trim()); // Prints 'Remove whitespace' (removed whitespace in front and back)
```

```
## String Interpolation
const myPet = 'armadillo';
console.log(`I own a pet ${myPet}.`); ##Use "backticks" instead of quotes
// Output: I own a pet armadillo.
```

Built in Objects

Math Object

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

console.log(Math.random()); // Prints a random number between 0 and 1
Math.floor() : takes a decimal number, and rounds down to the nearest whole number.
Math.ceil(x) : Returns the smallest integer greater than or equal to x.

Number Object

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

Number.isInteger() : Determine whether the passed value is an integer.

Variables

Declaring Variable (var, let, const)

var

```
var myName = 'Arya';
console.log(myName);
// Output: Arya
```

let

let keyword signals that the variable can be reassigned a different value.
let (and even var) is that we can declare a variable without assigning the variable a value. In such a case, the variable will be automatically initialized with a value of undefined:

const

A const variable cannot be reassigned because it is constant. If you try to reassign a const variable, you'll get a TypeError.

Operators

Pretty much same with Python

Comparison & Logical Operators

Only diff with python is:
equal to: ===
Is not equal to: !==

the and operator (&&)
the or operator (||)
the not operator, otherwise known as the bang operator (!)

Increment and Decrement

```
let a = 10;
a++;
console.log(a); // Output: 11

let b = 20;
b--;
console.log(b); // Output: 19
```

Typeof

```
const unknown1 = 'foo';
console.log(typeof unknown1); // Output: string
```

If / Else Statements

```
if (true) {
  console.log('This message will print!');
}
// Prints: This message will print!

if (false) {
  console.log('The code in this block will not run.');
} else {
  console.log('But the code in this block will!');
}
// Prints: But the code in this block will!
```

Truthy and Falsy values

Falsy values

0
Empty strings like "" or ''
null which represent when there is no value at all
undefined which represent when a declared variable lacks a value
NaN, or Not a Number

Ternary Operator

```
let isNightTime = true;

if (isNightTime) {
  console.log('Turn on the lights!');
} else {
  console.log('Turn off the lights!');
}
```

SAME AS

```
isNightTime ? console.log('Turn on the lights!') : console.log('Turn off the lights!');
```

Else – If Statement

The else if statement allows for more than two possible outcomes

```
let stopLight = 'yellow';

if (stopLight === 'red') {
  console.log('Stop!');
} else if (stopLight === 'yellow') {
  console.log('Slow down.');
} else if (stopLight === 'green') {
  console.log('Go!');
} else {
    console.log('Caution, unknown!');
}
```

Switch Statement

```
let groceryItem = 'papaya';

switch (groceryItem) {
    case 'tomato':
        console.log('Tomatoes are $0.49');
        break;
    case 'lime':
        console.log('Limes are $1.49');
        break;
    case 'papaya':
        console.log('Papayas are $1.29');
        break;
    default:
        console.log('Invalid item');
        break;
}
```

Functions

```
console.log(greetWorld()); // Output: Hello, World!

function greetWorld() {
    console.log('Hello, World!');
}
```

++++++++++++++++++

## parameters and default parameters

```
function greeting (name = 'stranger') {
    console.log(`Hello, ${name}!`)
}

greeting('Nick') // Output: Hello, Nick!
greeting() // Output: Hello, stranger!
```

## Assigning a constant var to a function

```
const plantNeedsWater = function(day) {
    if(day === 'Wednesday'){
        return true;
    } else {
        return false;
    }
};

plantNeedsWater('Tuesday');

console.log(plantNeedsWater('Tuesday'));
```

## arrow function syntax : shorter way to write functions and assign it to a const var

```
const rectangleArea = (width, height) => {
    let area = width * height;
    return area;
};
```

## Refactoring Functions to a more concise format

```
[Before Refactor]
const squareNum = (num) => {
    return num * num;
};
```

```
[After Refactor]
const squareNum = num => num * num;
```

## Functions as Data

```
const checkThatTwoPlusTwoEqualsFourAMillionTimes = () => {
  for(let i = 1; i <= 1000000; i++) {
    if ( (2 + 2) != 4) {
      console.log('Something has gone very wrong :( ');
    }
  }
};

// Write your code below
const is2p2 = checkThatTwoPlusTwoEqualsFourAMillionTimes;
is2p2();
console.log(is2p2.name);
//output
checkThatTwoPlusTwoEqualsFourAMillionTimes
```

Scope

Scope is the context in which our variables are declared. We think about scope in relation to blocks because variables can exist either outside of or within these blocks.

## Blocks
Block is the code found inside a set of curly braces {}. Blocks help us group one or more statements together and serve as an important structural marker for our code.

## Global Scope
In global scope, variables are declared outside of blocks. These variables are called global variables. Because global variables are not bound inside a block, they can be accessed by any code in the program, including code in blocks.

## Block Scope
When a variable is defined inside a block, it is only accessible to the code within the curly braces {}.

Array

Pretty much same properties as Python list

Various Array Methods
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

push, pop, length, join(), .slice(), .splice(), .shift(), .unshift(), and .concat() etc.

```
++++++++++++++++++
const newYearsResolutions = ['Keep a journal', 'Take a falconry class'];

console.log(newYearsResolutions.length);
// Output: 2

++++++++++++++++++++++++++++
const itemTracker = ['item 0', 'item 1', 'item 2'];

itemTracker.push('item 3', 'item 4');

console.log(itemTracker);
// Output: ['item 0', 'item 1', 'item 2', 'item 3', 'item 4'];
+++++++++++++++++++++++++++

const newItemTracker = ['item 0', 'item 1', 'item 2'];

const removed = newItemTracker.pop();

console.log(newItemTracker);
// Output: [ 'item 0', 'item 1' ]
console.log(removed);
// Output: item 2
+++++++++++++++++++++++++
```

Shift() : removes first item from array
Unshift(…): adds "…" as first time of array
Slice(a,b) : select / filter array[a] to array[b-1]
indexOf('….') : returns index number of '…' in the array

loop

## initialization, stopping condition, iteration statement

```
for (let counter = 0; counter < 4; counter++) {
   console.log(counter);
}

// output
0
1
2
3
```

## Nested Loops

```
const myArray = [6, 19, 20];
const yourArray = [19, 81, 2];
for (let i = 0; i < myArray.length; i++) {
   for (let j = 0; j < yourArray.length; j++) {
      if (myArray[i] === yourArray[j]) {
         console.log('Both loops have the number: ' + yourArray[j])
      }
   }
};
```

## while / do…while statement / Break

While, break work pretty much the same way as python

```
let countString = ' '; #empty string
let i = 0;

do {
   countString = countString + i;
   i++;
} while (i < 5);

console.log(countString);
```

Note that the while and do...while loop are different! Unlike the while loop, do...while will run at least once whether or not the condition evaluates to true.

Iterator

Array Iterator Documentation
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#Iteration_methods

## forEach

```
const fruits = ['mango', 'papaya', 'pineapple', 'apple'];

// Iterate over fruits below
fruits.forEach(fruit => console.log("I want to eat a " + fruit));
```

```
// I want to eat a mango
I want to eat a papaya
I want to eat a pineapple
I want to eat a apple
```

OR

```
function printFruits(fruit){
   console.log("I want to eat a " + fruit);
}

groceries.forEach(printFruits);
```

## map
takes an argument of a callback function and returns a new array!

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

const bigNumbers = numbers.map(number => {
   return number * 10;
});

console.log(numbers); // Output: [1, 2, 3, 4, 5]
console.log(bigNumbers); // Output: [10, 20, 30, 40, 50]
```

## Filter

Returns a new array. However, .filter() returns an array of elements after filtering out certain elements from the original array. The callback function for the .filter() method should return true or false depending on the element that is passed to it. The elements that cause the callback function to return true are added to the new array.

```
const words = ['chair', 'music', 'pillow', 'brick', 'pen', 'door'];

const shortWords = words.filter(word => {
   return word.length < 6;
});

console.log(words); // Output: ['chair', 'music', 'pillow', 'brick', 'pen', 'door'];
console.log(shortWords); // Output: ['chair', 'music', 'brick', 'pen', 'door']
```

## findIndex( )

.findIndex() on an array will return the index of the first element that evaluates to true in the callback function.

```
const jumbledNums = [123, 25, 78, 5, 9];
const lessThanTen = jumbledNums.findIndex(num => {
   return num < 10;
});

console.log(lessThanTen); // Output: 3
```

## Reduce

.reduce() method returns a single value after iterating through the elements of an array, thereby reducing the array.

```
const numbers = [1, 2, 4, 10];

const summedNums = numbers.reduce((accumulator, currentValue) => {
   return accumulator + currentValue
})

console.log(summedNums) // Output: 17
```

The .reduce() method can also take an optional second parameter to set an initial value for accumulator

```
const numbers = [1, 2, 4, 10];

const summedNums = numbers.reduce((accumulator, currentValue) => {
   return accumulator + currentValue
}, 100)    // <- Second argument for .reduce()

console.log(summedNums); // Output: 117
```

## Some

The some() method tests whether at least one element in the array passes the test implemented by the provided function. It returns a Boolean value.

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

// checks whether an element is even
const even = (element) => element % 2 === 0;

console.log(array.some(even));
// expected output: true
```

## Every

The every() method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.

```
const isBelowThreshold = (currentValue) => currentValue < 40;

const array1 = [1, 30, 39, 29, 10, 13];

console.log(array1.every(isBelowThreshold));
// expected output: true
```

Objects

We use curly braces, {}, to designate an object literal (similar to dictionary in Python)
Here the keys are called the "properties"

```
let spaceship = {
   homePlanet: 'Earth',
   color: 'silver'
};
```

## access values with dot notation

```
spaceship.homePlanet; // Returns 'Earth',
spaceship.color; // Returns 'silver'
```

## access values with bracket

```
let spaceship = {
  'Fuel Type': 'Turbo Fuel',
  'Active Duty': true,
  homePlanet: 'Earth',
  numCrew: 5
};
spaceship['Active Duty'];     // Returns true
```

delete spaceship[‘Active Duty’]

## method

.xxxx( ) of an object

```
const alienShip = {
  invade: function () {
    console.log('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.')
  }
};
```

OR

```
const alienShip = {
  invade () {
    console.log('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.')
  }
};
```

If you wanna have multiple methods in one object…

```
let alienShip = {
  retreat() {
    console.log(retreatMessage)
  },
  takeOff() {
    console.log('Spim... Borp... Glix... Blastoff!')
  }
};
```

## Nested Objects

# Pass by Reference

```
let spaceship = {
  'Fuel Type' : 'Turbo Fuel',
  homePlanet : 'Earth'
};

let greenEnergy = obj => {
  obj['Fuel Type'] = 'avocado oil';
}

let remotelyDisable = obj => {
  obj.disabled = true;
}

greenEnergy(spaceship);
```

```
remotelyDisable(spaceship);

console.log(spaceship)
```

## Looping through nested objects

```
let spaceship = {
        crew: {
        captain: {
                name: 'Lily',
                degree: 'Computer Engineering',
                cheerTeam() { console.log('You got this!') }
                },
        'chief officer': {
                name: 'Dan',
                degree: 'Aerospace Engineering',
                agree() { console.log('I agree, captain!') }
                },
        medic: {
                name: 'Clementine',
                degree: 'Physics',
                announce() { console.log(`Jets on!`) } },
        translator: {
                name: 'Shauna',
                degree: 'Conservation Science',
                powerFuel() { console.log('The tank is full!') }
                }
        }
};

for (let crewMember in spaceship.crew) { ##for … in… syntax
    console.log(`${crewMember}: ${spaceship.crew[crewMember].name}`)
};

// captain: Lily
chief officer: Dan
medic: Clementine
translator: Shauna
```

Advanced Objects

```
const goat = {
    dietType: 'herbivore',
    makeSound() {
        console.log('baaa');
    },
    diet() {
        console.log(this.dietType);
    }
};

goat.diet();
// Output: herbivore
```

## If you use “arrow function”, the “this” method may not work as intended

```
const goat = {
    dietType: 'herbivore',
    makeSound() {
        console.log('baaa');
```

```
  },
  diet: () => {
    console.log(this.dietType);
  }
};

goat.diet(); // Prints undefined
```

## Privacy

place an underscore _ before the name of a property to mean that the property should not be altered

# Getter

Getters are methods that get and return the internal properties of an object.
Getters can perform an action on the data when getting a property.
Getters can return different values using conditionals.
In a getter, we can access the properties of the calling object using this.
The functionality of our code is easier for other developers to understand.

```
const person = {
  _firstName: 'John',
  _lastName: 'Doe',
  get fullName() {
    if (this._firstName && this._lastName){
      return `${this._firstName} ${this._lastName}`;
    } else {
      return 'Missing a first name or a last name.';
    }
  }
}

// To call the getter method:
person.fullName; // 'John Doe'
```

## Setter
reassign values of existing properties within an object
Like getter methods, there are similar advantages to using setter methods that include checking input, performing actions on properties, and displaying a clear intention for how the object is supposed to be used. Nonetheless, even with a setter method, it is still possible to directly reassign properties.

```
const person = {
  _age: 37,
  set age(newAge){
    if (typeof newAge === 'number'){
      this._age = newAge;
    } else {
      console.log('You must assign a number to age');
    }
  }
};

person.age = 40;
console.log(person._age); // Logs: 40
```

## Factory Functions

there are times where we want to create many instances of an object quickly.
A factory function is a function that returns an object and can be reused to make multiple object instances. Factory

functions can also have parameters allowing us to customize the object that gets returned.

```
const monsterFactory = (name, age, energySource, catchPhrase) => {
   return {
      name: name, # you can actually remove the keys (truncated version) and it will still work fine
      age: age,
      energySource: energySource,
      scare() {
         console.log(catchPhrase);
      }
   }
};
```

## Destructured Assignment

```
const vampire = {
   name: 'Dracula',
   residence: 'Transylvania',
   preferences: {
      day: 'stay inside',
      night: 'satisfy appetite'
   }
};
```

INSTEAD OF

```
const residence = vampire.residence;
console.log(residence); // Prints 'Transylvania'
```

WE CAN DO

```
const { residence } = vampire;
console.log(residence); // Prints 'Transylvania'
```

WE CAN EVEN USE destructured assignment TO GRAB NESTED PROPERTIES OF AN OBJECT

```
const { day } = vampire.preferences;
console.log(day); // Prints 'stay inside'
```

## Built in Object Methods

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#Methods

.hasOwnProperty(), .valueOf() etc.

```
const robot = {
            model: 'SAL-1000',
   mobile: true,
   sentient: false,
   armor: 'Steel-plated',
   energyLevel: 75
};

const robotKeys = Object.keys(robot);

console.log(robotKeys);
// [ 'model', 'mobile', 'sentient', 'armor', 'energyLevel' ]

const robotEntries = Object.entries(robot)
console.log(robotEntries);
[ [ 'model', 'SAL-1000' ],
   [ 'mobile', true ],
   [ 'sentient', false ],
   [ 'armor', 'Steel-plated' ],
   [ 'energyLevel', 75 ] ]

const newRobot = Object.assign({laserBlaster: true, voiceRecognition: true}, robot);
console.log(newRobot);
{ laserBlaster: true,
   voiceRecognition: true,
   model: 'SAL-1000',
   mobile: true,
   sentient: false,
   armor: 'Steel-plated',
   energyLevel: 75 }
```

Classes

## Constructor
Class and object seem similar but most importance difference is the “constructor” method.
JavaScript calls the **constructor**() method every time it creates a new instance of a class.

```
class Dog {
  constructor(name) {
    this.name = name;
    this.behavior = 0;
  }
}
```

## Instance
instance is an object that contains the property names and methods of a class, but with unique property values.

```
const halley = new Dog('Halley'); // Create new Dog instance
```

## Methods

Class method and getter syntax is the same as it is for objects except you can’t include commas between methods.

```
class Dog {
  constructor(name) {
    this._name = name;
    this._behavior = 0;
  }

  get name() {
    return this._name;
  }

  get behavior() {
    return this._behavior;
  }

  incrementBehavior() {
    this._behavior++;
  }
}
```

## Inheritance

With inheritance, you can create a **parent** class (also known as a superclass) with properties and methods that multiple child classes (also known as subclasses) share. The **child** classes inherit the properties and methods from their parent class.

```
class Animal {
  constructor(name) {
    this._name = name;
    this._behavior = 0;
  }

  get name() {
    return this._name;
  }

  get behavior() {
    return this._behavior;
  }

  incrementBehavior() {
    this._behavior++;
  }
}
```

Cat class will inherit information from the Animal Class

```
class Cat {
  constructor(name, usesLitter) {
    this._name = name;
    this._usesLitter = usesLitter;
    this._behavior = 0;
  }

  get name() {
    return this._name;
  }

  get behavior() {
    return this._behavior;
  }

  get usesLitter() {
    return this._usesLitter;
  }

  incrementBehavior() {
    this._behavior++;
  }
}

class Cat extends Animal {
  constructor(name, usesLitter) {
    super(name);
    this._usesLitter = usesLitter;
  }
}
```

- *The extends keyword makes the methods of the animal class available inside the cat class.*
- *The super keyword calls the constructor of the parent class. In this case, super(name) passes the name argument of the Cat class to the constructor of the Animal class. When the Animal constructor runs, it sets this._name = name; for new Cat instances.*

```
const bryceCat = new Cat('Bryce', false);
```

## Static Methods

Sometimes you will want a class to have methods that aren't available in individual instances, but that you can call directly from the class.

```
class Animal {
  constructor(name) {
    this._name = name;
    this._behavior = 0;
  }

  static generateName() {
    const names = ['Angel', 'Spike', 'Buffy', 'Willow', 'Tara'];
    const randomNumber = Math.floor(Math.random()*5);
    return names[randomNumber];
  }
}

console.log(Animal.generateName()); // returns a name

const tyson = new Animal('Tyson');
tyson.generateName(); // TypeError
```

The example above will result in an error, because you cannot call static methods (.generateName()) on an instance (tyson).

Intermediate Modules

## Defining a module

```
let Menu = {};
Menu.specialty = "Roasted Beet Burger with Mint Sauce";

module.exports = Menu;
```

## Require( ) function

➔ require() function to import modules

```
const Menu = require('./menu.js');

function placeOrder() {
  console.log('My order is: ' + Menu.specialty);
}

placeOrder();
```

OR

You can just directly save functions and properties to **module.exports** :

```
module.exports = {
  specialty: "Roasted Beet Burger with Mint Sauce",
  getSpecialty: function() {
    return this.specialty;
  }
};

const Menu = require('./menu.js');

console.log(Menu.getSpecialty());
```

++++++++++++++++++++++++++

2-airplane.js

```
module.exports = {
  myAirplane: "CloudJet",
  displayAirplane: function() {
    return this.myAirplane;
  }
};
```

2-missionControl.js

```
const Airplane = require('./2-airplane.js');

console.log(Airplane.displayAirplane());
```

## export default

export default in place of module.exports. Node.js doesn't support export default by default. module.exports is usually used for Node.js development and ES6 syntax is used for front-end development

```
let Menu = {};
export default Menu;
```

## Import

Airplane.js

```
let Airplane = {};

Airplane.availableAirplanes = [
{
  name: 'AeroJet',
  fuelCapacity: 800
 },
 {name: 'SkyJet',
  fuelCapacity: 500
 }
];

export default Airplane;
```

missionControl.js

```
import Airplane from './airplane';

function displayFuelCapacity() {
  Airplane.availableAirplanes.forEach(function(element){
  console.log('Fuel Capacity of ' + element.name + ': ' + element.fuelCapacity);
  });
}

displayFuelCapacity();
```

## Named Exports and Imports

Airplane.js

```
let availableAirplanes = [
    {
  name: 'AeroJet',
  fuelCapacity: 800,
  availableStaff: ['pilots', 'flightAttendants', 'engineers', 'medicalAssistance', 'sensorOperators'],
},
{name: 'SkyJet',
  fuelCapacity: 500,
  availableStaff: ['pilots', 'flightAttendants']
}
];

let flightRequirements = {
    requiredStaff: 4,
};

function meetsStaffRequirements(availableStaff, requiredStaff) {
    if (availableStaff.length >= requiredStaff) {
        return true;
    } else {
        return false;
    }
};

export { availableAirplanes, flightRequirements, meetsStaffRequirements};
```

missionControl.js

```
import {availableAirplanes, flightRequirements, meetsStaffRequirements} from './airplane';

function displayFuelCapacity() {
    availableAirplanes.forEach(function(element) {
        console.log('Fuel Capacity of ' + element.name + ': ' + element.fuelCapacity);
    });
}

displayFuelCapacity();

function displayStaffStatus() {
    availableAirplanes.forEach(function(element) {
      console.log(element.name + ' meets staff requirements: ' + meetsStaffRequirements(element.availableStaff, flightRequirements.requiredStaff) );
    });
}

displayStaffStatus();
```

## Export as

```
let specialty = '';
let isVegetarian = function() {
};
let isLowSodium = function() {
};

export { specialty as chefsSpecial, isVegetarian as isVeg, isLowSodium };
```

## Import as

Can use the same shortened module names in export as to import as

```
import { chefsSpecial, isVeg } from './menu';
```

OR if you want to import all function and objects from a module using import as…

```
import * as Carte from './menu';
```

```
Carte.chefsSpecial;
Carte.isVeg();
Carte.isLowSodium();
```

Promises

The Promise constructor method takes a function parameter called the executor function which runs automatically when the constructor is called. The executor function generally starts an asynchronous operation and dictates how the promise should be settled.

resolve is a function with one argument. Under the hood, if invoked, resolve() will change the promise's status from pending to fulfilled, and the promise's resolved value will be set to the argument passed into resolve().

reject is a function that takes a reason or error as an argument. Under the hood, if invoked, reject() will change the promise's status from pending to rejected, and the promise's rejection reason will be set to the argument passed into reject().

```
const executorFunction = (resolve, reject) => { };
const myFirstPromise = new Promise(executorFunction);
```

## Example Executor function

```
const executorFunction = (resolve, reject) => {
   if (someCondition) {
         resolve('I resolved!');
   } else {
         reject('I rejected!');
   }
}
const myFirstPromise = new Promise(executorFunction);
```

## Node setTimeout( ) function

setTimeout() is a Node API (a comparable API is provided by web browsers) that uses callback functions to schedule tasks to be performed after a delay. setTimeout() has two parameters: a callback function and a delay in milliseconds.

```
~~Call back Function~~
const delayedHello = () => {
   console.log('Hi! This is an asynchronous greeting!');
};

setTimeout(delayedHello, 2000);
```

## Consuming Promises

The initial state of an asynchronous promise is pending, but we have a guarantee that it will settle. How do we tell the computer what should happen then? Promise objects come with an aptly named **.then( )** method. It allows us to say, "I have a promise, when it settles, then here's what I want to happen…"

```
let prom = new Promise((resolve, reject) => {
   let num = Math.random();
   if (num < .5 ){
      resolve('Yay!');
   } else {
      reject('Ohhh noooo!');
   }
});

const handleSuccess = (resolvedValue) => {
   console.log(resolvedValue);
};
```

```
const handleFailure = (rejectionReason) => {
    console.log(rejectionReason);
};

prom.then(handleSuccess, handleFailure);
```

## Catch

.catch() accomplishes the same thing as using a .then() with only a failure handler.

```
prom
    .then((resolvedValue) => {
        console.log(resolvedValue);
    })
    .catch((rejectionReason) => {
        console.log(rejectionReason);
    });
```

## Chaining multiple promises

This process of chaining promises together is called **composition**.

```
const {checkInventory, processPayment, shipOrder} = require('./library.js');

const order = {
    items: [['sunglasses', 1], ['bags', 2]],
    giftcardBalance: 79.82
};

checkInventory(order)
.then((resolvedValueArray) => {
    // Write the correct return statement here:
  return processPayment(resolvedValueArray);
})
.then((resolvedValueArray) => {
    // Write the correct return statement here:
    return shipOrder(resolvedValueArray);
})
.then((successMessage) => {
    console.log(successMessage);
})
.catch((errorMessage) => {
    console.log(errorMessage);
});
```

## Promise all

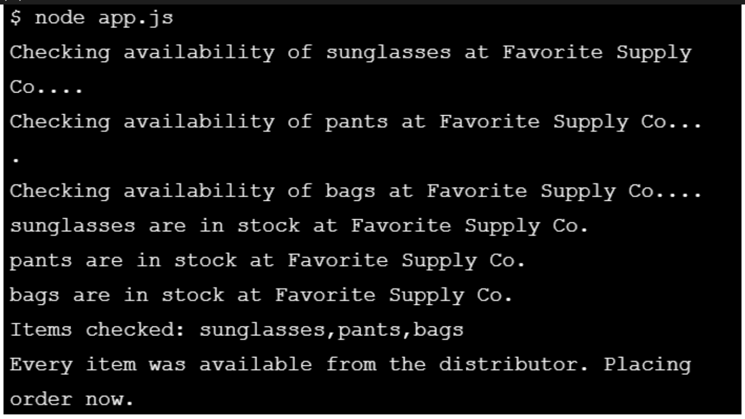
What if we're dealing with multiple promises, but we don't care about the order? To maximize efficiency we should use concurrency, multiple asynchronous operations happening together. With promises, we can do this with the function Promise.all().

EXAMPLE

Library.js

```
const checkAvailability = (itemName, distributorName) => {
    console.log(`Checking availability of ${itemName} at ${distributorName}...`);
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (restockSuccess()) {
                console.log(`${itemName} are in stock at ${distributorName}`)
                resolve(itemName);
            } else {
                reject(`Error: ${itemName} is unavailable from ${distributorName} at this time.`);
            }
        }, 1000);
    });
};

module.exports = { checkAvailability };

// This is a function that returns true 80% of the time
// We're using it to simulate a request to the distributor being successful this often
function restockSuccess() {
    return (Math.random() > .2);
}
```

```
const {checkAvailability} = require('./library.js');

const onFulfill = (itemsArray) => {
  console.log(`Items checked: ${itemsArray}`);
  console.log(`Every item was available from the distributor. Placing order now.`);
};

const onReject = (rejectionReason) => {
  console.log(rejectionReason);
};

// Write your code below:

const checkSunglasses = checkAvailability('sunglasses', 'Favorite Supply Co.');
const checkPants = checkAvailability('pants', 'Favorite Supply Co.');
const  checkBags = checkAvailability('bags', 'Favorite Supply Co.');

Promise.all([checkSunglasses, checkPants, checkBags])
  .then(onFulfill)
  .catch(onReject);

// OUTPUT ON BASH
```



```
$ node app.js
Checking availability of sunglasses at Favorite Supply
Co....
Checking availability of pants at Favorite Supply Co...
.
Checking availability of bags at Favorite Supply Co....
sunglasses are in stock at Favorite Supply Co.
pants are in stock at Favorite Supply Co.
bags are in stock at Favorite Supply Co.
Items checked: sunglasses,pants,bags
Every item was available from the distributor. Placing
order now.
```

Async Wait

Introduces a new syntax for using promises and generators. Improves readability and scalability of our code.

## async

async keyword is used to write functions that handle asynchronous actions
async functions always return a promise. This means we can use traditional promise syntax, like .then() and .catch with our async functions.
- If there's nothing returned from the function, it will return a promise with a resolved value of undefined.
- If there's a non-promise value returned from the function, it will return a promise resolved to that value.

- If a promise is returned from the function, it will simply return that promise

```
async function fivePromise() {
   return 5;
}

fivePromise()
.then(resolvedValue => {
       console.log(resolvedValue);
   })   // Prints 5
```

## await operator

The await keyword can only be used inside an async function. await is an operator: it returns the resolved value of a promise. Since promises resolve in an indeterminate amount of time, await halts, or pauses, the execution of our async function until a given promise is resolved.

```
async function asyncFuncExample(){
   let resolvedValue = await myPromise();
   //myPromise() is a function that returns a promise which will resolve to the string "I am resolved now!".
   console.log(resolvedValue);
}

asyncFuncExample(); // Prints: I am resolved now!
```

## Wring async Functions

```
async function noAwait() {
 let value = myPromise();
 console.log(value);
}

async function yesAwait() {
 let value = await myPromise();
 console.log(value);
}

noAwait(); // Prints: Promise { <pending> }
yesAwait(); // Prints: Yay, I resolved!
```

// When used properly in yesAwait(), the variable value was assigned the resolved value of the myPromise() promise, whereas in noAwait(), value was assigned the promise object itself.

## Handling Dependent Promises

The true beauty of async...await is when we have a series of asynchronous actions which depend on one another.

This function…

```
function nativePromiseVersion() {
       returnsFirstPromise()
       .then((firstValue) => {
              console.log(firstValue);
              return returnsSecondPromise(firstValue);
       })
     .then((secondValue) => {
              console.log(secondValue);
       });
}
```

Can be rewritten in a simpler form using async … wait

```
async function asyncAwaitVersion() {
  let firstValue = await returnsFirstPromise();
  console.log(firstValue);
  let secondValue = await returnsSecondPromise(firstValue);
  console.log(secondValue);
}
```

FULL EXAMPLE

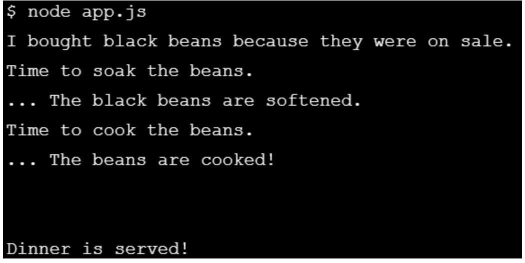
Library.js

```
/*
This is the shopForBeans function from the last exercise
*/

const shopForBeans = () => {
  return new Promise((resolve, reject) => {
  const beanTypes = ['kidney', 'fava', 'pinto', 'black', 'garbanzo'];
  setTimeout(()=>{
    let randomIndex = Math.floor(Math.random() * 5)
    let beanType = beanTypes[randomIndex];
    console.log(`I bought ${beanType} beans because they were on sale.`)
   resolve(beanType);
  }, 1000)
})
}

let soakTheBeans = (beanType) => {
   return new Promise((resolve, reject) => {
     console.log('Time to soak the beans.')
    setTimeout(()=>{
      console.log(`... The ${beanType} beans are softened.`)
      resolve(true)
      }, 1000)
  })
}

let cookTheBeans = (isSoftened) => {
  return new Promise((resolve, reject) => {
    console.log('Time to cook the beans.')
    setTimeout(()=>{
      if (isSoftened) {
        console.log('... The beans are cooked!')
        resolve('\n\nDinner is served!')
      }
    }, 1000)
  })
}


module.exports = {shopForBeans, soakTheBeans, cookTheBeans}
```

app.js

```
const {shopForBeans, soakTheBeans, cookTheBeans} = require('./library.js');

// Write your code below:
async function makeBeans() {
  let type = await shopForBeans();
  let isSoft = await soakTheBeans(type);
  let dinner = await cookTheBeans(isSoft);
  console.log(dinner);
}

makeBeans();
```

//Bash Output



```
$ node app.js
I bought black beans because they were on sale.
Time to soak the beans.
... The black beans are softened.
Time to cook the beans.
... The beans are cooked!


Dinner is served!
```

## Handling Errors : Try … Catch

FULL EXAMPLE

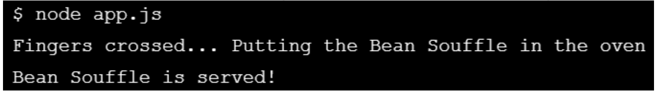
Library.js

```
//This function returns true 50% of the time.
let randomSuccess = () => {
 let num = Math.random();
 if (num < .5 ){
   return true;
 } else {
   return false;
 }
};

//This function returns a promise that resolves half of the time and rejects half of the time
let cookBeanSouffle = () => {
 return new Promise((resolve, reject) => {
   console.log('Fingers crossed... Putting the Bean Souffle in the oven');
   setTimeout(()=>{
     let success = randomSuccess();
     if(success){
       resolve('Bean Souffle');
     } else {
       reject('Dinner is ruined!');
     }
   }, 1000);
 })
};

module.exports = cookBeanSouffle;
```

app.js

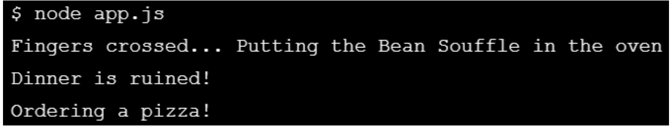
```
const cookBeanSouffle = require('./library.js');

// Write your code below:

async function hostDinnerParty() {
 try {
   let dinner = await cookBeanSouffle();
   console.log(`${dinner} is served!`);
 }
 catch(error){
   console.log(error);
   console.log('Ordering a pizza!');
 }
}

hostDinnerParty();
```

// Output when everything went okay and promise got resolved



```
$ node app.js
Fingers crossed... Putting the Bean Souffle in the oven
Bean Souffle is served!
```

// Output when there was an error (when sth went wrong and promise got rejected)



```
$ node app.js
Fingers crossed... Putting the Bean Souffle in the oven
Dinner is ruined!
Ordering a pizza!
```

## Handling Independent Promises

But what if our async function contains multiple promises which are not dependent on the results of one another to execute?

```
async function concurrent() {
  const firstPromise = firstAsyncThing();
  const secondPromise = secondAsyncThing();
console.log(await firstPromise, await secondPromise);
}
```

With our concurrent() function both promises' asynchronous operations can be run simultaneously. If possible, we want to get started on each asynchronous operation as soon as possible!

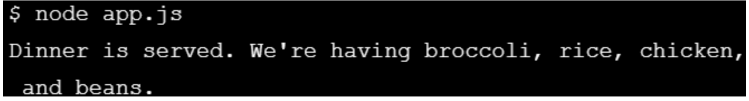
FULL EXAMPLE

```
let cookBeans = () => {
  return new Promise ((resolve, reject) => {
   setTimeout(()=>{
     resolve('beans')
   }, 1000)
 })
}

let steamBroccoli = () => {
 return new Promise ((resolve, reject) => {
   setTimeout(()=>{
     resolve('broccoli')
   }, 1000)
 })
}

let cookRice = () => {
 return new Promise ((resolve, reject) => {
   setTimeout(()=>{
     resolve('rice')
   }, 1000)
 })
}

let bakeChicken = () => {
 return new Promise ((resolve, reject) => {
   setTimeout(()=>{
     resolve('chicken')
   }, 1000)
 })
}

module.exports = {cookBeans, steamBroccoli, cookRice, bakeChicken}
```

app.js

```
let {cookBeans, steamBroccoli, cookRice, bakeChicken} = require('./library.js')

// Write your code below:

async function serveDinner() {
 let vegetablePromise = steamBroccoli();
 let starchPromise = cookRice();
 let proteinPromise = bakeChicken();
 let sidePromise = cookBeans();
 console.log(`Dinner is served. We're having ${await vegetablePromise}, ${await starchPromise}, ${await proteinPromise}, and ${await sidePromise}.`)
}

serveDinner()
```

// Bash Output



```
$ node app.js
Dinner is served. We're having broccoli, rice, chicken,
 and beans.
```

## Promise.all( )

We can pass an array of promises as the argument to Promise.all(), and it will return a single promise. This promise will resolve when all of the promises in the argument array have resolved.

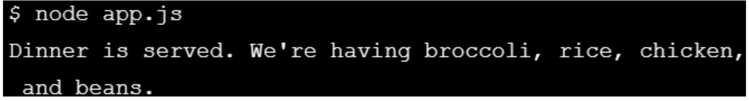
```
async function asyncPromAll() {
    const resultArray = await Promise.all([asyncTask1(), asyncTask2(), asyncTask3(), asyncTask4()]);
    for (let i = 0; i<resultArray.length; i++){
        console.log(resultArray[i]);
    }
}
```

(library.js is the same as the previous section on broccoli, rice etc.)

App.js

```
let {cookBeans, steamBroccoli, cookRice, bakeChicken} = require('./library.js')

// Write your code below:
async function serveDinnerAgain(){
  let foodArray = await Promise.all([steamBroccoli(), cookRice(), bakeChicken(), cookBeans()]);

  console.log(`Dinner is served. We're having ${foodArray[0]}, ${foodArray[1]}, ${foodArray[2]}, and ${foodArray[3]}.`)
}

serveDinnerAgain()
```

// Bash Output (same output as the previous section FULL EXAMPLE but this time using promise.all( ))



```
$ node app.js
Dinner is served. We're having broccoli, rice, chicken,
 and beans.
```

Requests

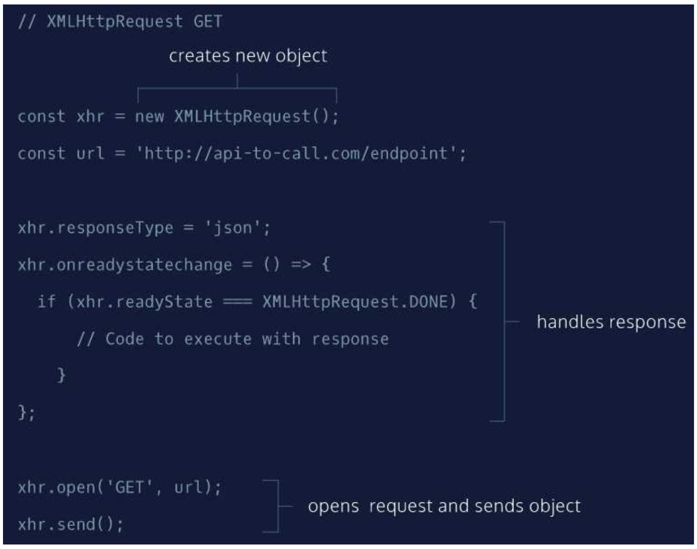
## HTTP Requests

One of JavaScript's greatest assets is its non-blocking properties, or that it is an asynchronous language.

Websites, like newspaper websites, take advantage of these non-blocking properties to provide a better user experience. Generally, a site's code is written so that users don't have to wait for a giant image to load before being allowed to read the actual article—rather, that text is rendered first and then the image can load in the background.

## XHR GET Requests

Asynchronous JavaScript and XML (AJAX), enables requests to be made after the initial page load.
[XML Documentation](#)
the XMLHttpRequest (XHR) API, named for XML, can be used to make many kinds of requests and supports other forms of data.



```
// XMLHttpRequest GET

const xhr = new XMLHttpRequest();
const url = 'http://api-to-call.com/endpoint';

xhr.responseType = 'json';
xhr.onreadystatechange = () => {
  if (xhr.readyState === XMLHttpRequest.DONE) {
      // Code to execute with response
    }
};

xhr.open('GET', url);
xhr.send();
```

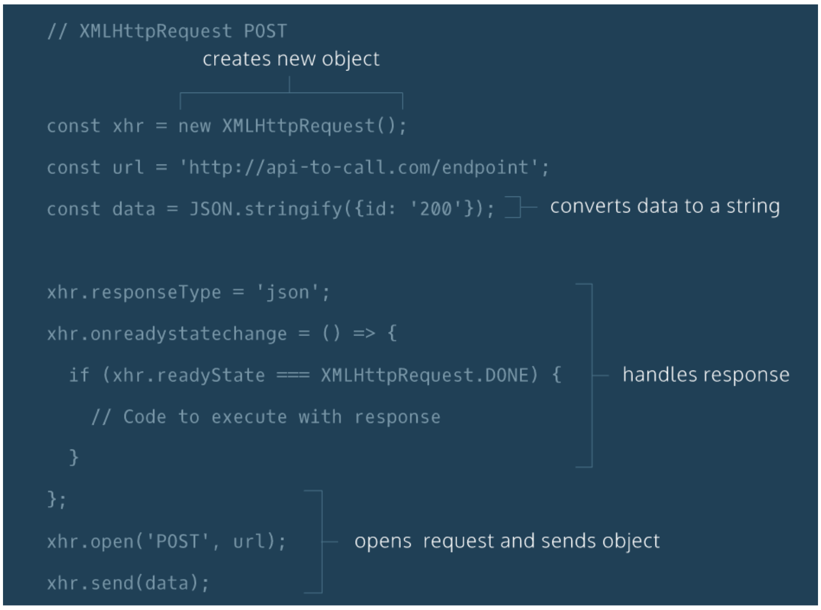
FULL EXAMPLE

```
const xhr = new XMLHttpRequest();
const url = 'https://api-to-call.com/endpoint';
xhr.responseType = 'json';
xhr.onreadystatechange = () => {
   if (xhr.readyState === XMLHttpRequest.DONE){
      return xhr.response;
   }
}

xhr.open('GET', url);
xhr.send();
```

FULL EXAMPLE: GET Request to Datamuse API to search for words that rhyme!

```
// Information to reach API
const url = 'https://api.datamuse.com/words?';
const queryParams = 'rel_rhy=';

// Selecting page elements
const inputField = document.querySelector('#input');
const submit = document.querySelector('#submit');
const responseField = document.querySelector('#responseField');

// AJAX function
const getSuggestions = () => {
  const wordQuery = inputField.value;
  const endpoint = `${url}${queryParams}${wordQuery}`;

  const xhr = new XMLHttpRequest();
  xhr.responseType = 'json';

  xhr.onreadystatechange = () => {
    if (xhr.readyState === XMLHttpRequest.DONE) {
      renderResponse(xhr.response);
    }
  }

  xhr.open('GET', endpoint);
  xhr.send();
}

// Clear previous results and display results to webpage
const displaySuggestions = (event) => {
  event.preventDefault();
  while(responseField.firstChild){
    responseField.removeChild(responseField.firstChild);
  }
  getSuggestions();
}

submit.addEventListener('click', displaySuggestions);
```

## XHR Post Requests



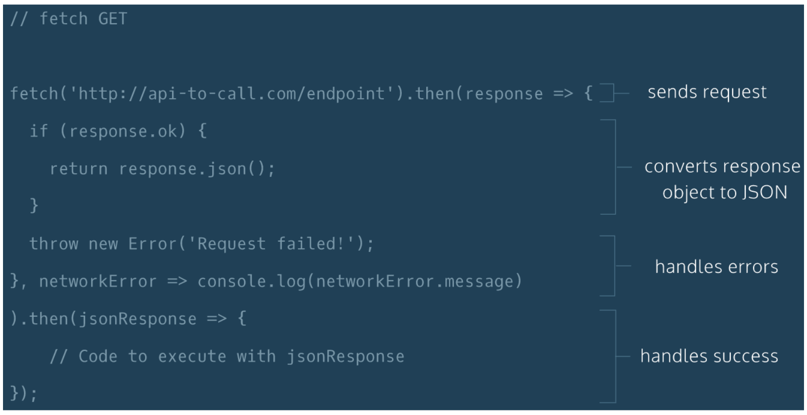
```
// XMLHttpRequest POST

const xhr = new XMLHttpRequest();
const url = 'http://api-to-call.com/endpoint';
const data = JSON.stringify({id: '200'});

xhr.responseType = 'json';
xhr.onreadystatechange = () => {
  if (xhr.readyState === XMLHttpRequest.DONE) {
    // Code to execute with response
  }
};
xhr.open('POST', url);
xhr.send(data);
```

FULL EXAMPLE: POST request to Rebrandly API to shorten URL

// Information to reach API

```
const apiKey = '<Your API Key>';
const url = 'https://api.rebrandly.com/v1/links';

// Some page elements
const inputField = document.querySelector('#input');
const shortenButton = document.querySelector('#shorten');
const responseField = document.querySelector('#responseField');

// AJAX functions
const shortenUrl = () => {
  const urlToShorten = inputField.value;
  const data = JSON.stringify({destination: urlToShorten});

  const xhr = new XMLHttpRequest();
  xhr.responseType = 'json';

  xhr.onreadystatechange = () => {
    if (xhr.readyState === XMLHttpRequest.DONE) {
                renderResponse(xhr.response);
                }
  }
  xhr.open('POST', url);
  xhr.setRequestHeader('Content-type', 'application/json');
        xhr.setRequestHeader('apikey', apiKey);
  xhr.send(data);
}


// Clear page and call AJAX functions
const displayShortUrl = (event) => {
  event.preventDefault();
  while(responseField.firstChild){
    responseField.removeChild(responseField.firstChild);
  }
  shortenUrl();
}

shortenButton.addEventListener('click', displayShortUrl);
```

## Fetch( ) get request

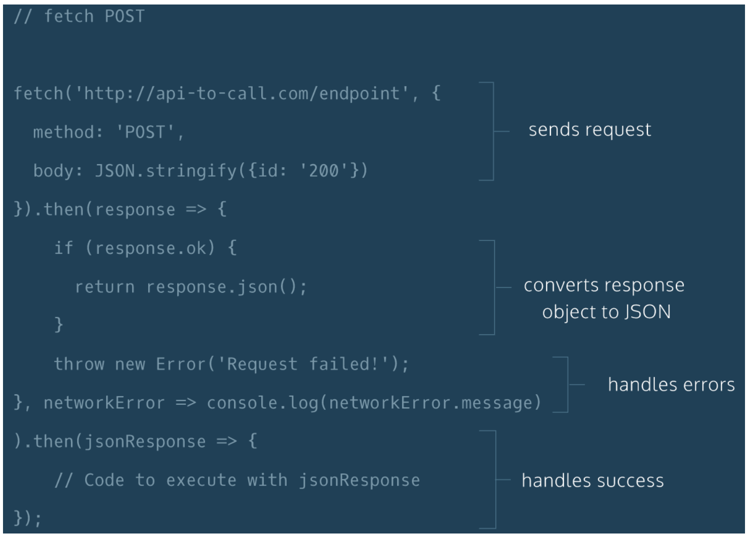


```
// fetch GET


fetch('http://api-to-call.com/endpoint').then(response => {
  if (response.ok) {
    return response.json();
  }
  throw new Error('Request failed!');
}, networkError => console.log(networkError.message)
).then(jsonResponse => {
    // Code to execute with jsonResponse
});
```

FULL EXAMPLE: use fetch(), then() for GET Request and manipulate it to access the Datamuse API and render information in the browser

```
// Information to reach API
const url = 'https://api.datamuse.com/words';
const queryParams = '?sl=';

// Selects page elements
const inputField = document.querySelector('#input');
const submit = document.querySelector('#submit');
const responseField = document.querySelector('#responseField');

// AJAX function
const getSuggestions = () => {
  const wordQuery = inputField.value;
  const endpoint = `${url}${queryParams}${wordQuery}`;

  fetch(endpoint, {cache: 'no-cache'}).then(response => {
    if (response.ok) {
      return response.json();
    }
    throw new Error('Request failed!');
  }, networkError => {
    console.log(networkError.message)
  })
}

// Clears previous results and display results to webpage
const displaySuggestions = (event) => {
  event.preventDefault();
  while(responseField.firstChild){
    responseField.removeChild(responseField.firstChild);
  }
  getSuggestions();
};

submit.addEventListener('click', displaySuggestions);
```

## Fetch( ) Post Request



```
// fetch POST

fetch('http://api-to-call.com/endpoint', {
  method: 'POST',
  body: JSON.stringify({id: '200'})
}).then(response => {
    if (response.ok) {
      return response.json();
    }
    throw new Error('Request failed!');
}, networkError => console.log(networkError.message)
).then(jsonResponse => {
    // Code to execute with jsonResponse
});
```

FULL EXAMPLE: Rebrandly API

```
// Information to reach API
const apiKey = '<Your API Key>';
const url = 'https://api.rebrandly.com/v1/links';

// Some page elements
```

```
const inputField = document.querySelector('#input');
const shortenButton = document.querySelector('#shorten');
const responseField = document.querySelector('#responseField');

// AJAX functions
const shortenUrl = () => {
  const urlToShorten = inputField.value;
  const data = JSON.stringify({destination: urlToShorten})

        fetch(url, {
    method: 'POST',
    headers: {
      'Content-type': 'application/json',
      'apikey': apiKey
    },
    body: data
  })
}

// Clear page and call AJAX functions
const displayShortUrl = (event) => {
  event.preventDefault();
  while(responseField.firstChild){
    responseField.removeChild(responseField.firstChild)
  }
  shortenUrl();
}

shortenButton.addEventListener('click', displayShortUrl);
```

## async GET Requests

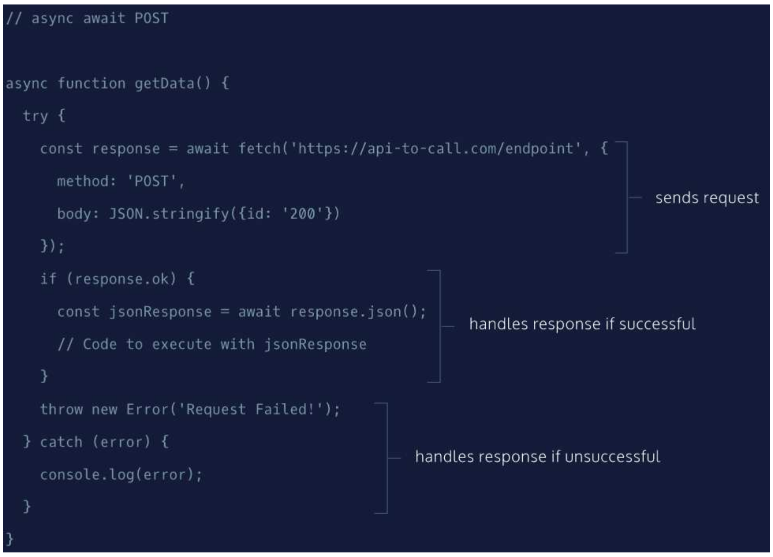
```
const getData = async () => {
  try {
    const response = await fetch('https://api-to-call.com/endpoint');
    if (response.ok) {
      const jsonResponse = await response.json();
      return jsonResponse;
    }
    throw new Error('Request failed!');
  } catch (error) {
    console.log(error);
  }
}
```

+++++++++++++++++

```
// Information to reach API
const url = 'https://api.datamuse.com/words?';
const queryParams = 'rel_jja=';

// Selecting page elements
const inputField = document.querySelector('#input');
const submit = document.querySelector('#submit');
const responseField = document.querySelector('#responseField');

// AJAX function
// Code goes here
const getSuggestions = async () => {
  const wordQuery = inputField.value;
  const endpoint = `${url}${queryParams}${wordQuery}`;
```

```
    try {
        const response = await fetch(endpoint, {cache: 'no-cache'});
        if(response.ok){
            const jsonResponse = await response.json();
            renderResponse(jsonResponse);
        }
    } catch (error) {
        console.log(error);
    }
}

// Clear previous results and display results to webpage
const displaySuggestions = (event) => {
    event.preventDefault();
    while(responseField.firstChild){
        responseField.removeChild(responseField.firstChild);
    }
    getSuggestions();
}

submit.addEventListener('click', displaySuggestions);
```

## aynsc post requests



```
// async await POST


async function getData() {
  try {
    const response = await fetch('https://api-to-call.com/endpoint', {
      method: 'POST',
      body: JSON.stringify({id: '200'})
    });
    if (response.ok) {
      const jsonResponse = await response.json();
      // Code to execute with jsonResponse
    }
    throw new Error('Request Failed!');
  } catch (error) {
    console.log(error);
  }
}
```

sends request
handles response if successful
handles response if unsuccessful

```
const getData = async () => {
    try {
        const response = await fetch('https://api-to-call.com/endpoint', {
            method: 'POST',
            body: JSON.stringify({id: 200})
        })
        if(response.ok){
            const jsonResponse = await response.json();
            return jsonResponse;
        }
        throw new Error('Request failed!');
    } catch(error) {
        console.log(error);
    }
}
```

+++++++++++++++++++

FULL EXAMPLE: Rebrandly API

```
// information to reach API
const apiKey = '<Your API Key>';
const url = 'https://api.rebrandly.com/v1/links';

// Some page elements
const inputField = document.querySelector('#input');
const shortenButton = document.querySelector('#shorten');
const responseField = document.querySelector('#responseField');

// AJAX functions
// Code goes here
const shortenUrl = async () => {
        const urlToShorten = inputField.value;
  const data = JSON.stringify({destination: urlToShorten});
  try {
    const response = await fetch(url, {
                    method: 'POST',
      body: data,
      headers: {
        'Content-type': 'application/json',
                          'apikey': apiKey
      }
    });
              if(response.ok){
      const jsonResponse = await response.json();
      renderResponse(jsonResponse);
    }
  } catch (error) {
    console.log(error);
  }
}

// Clear page and call AJAX functions
const displayShortUrl = (event) => {
  event.preventDefault();
  while(responseField.firstChild){
    responseField.removeChild(responseField.firstChild);
  }
  shortenUrl();
}

shortenButton.addEventListener('click', displayShortUrl);
```