**INTERMEDIATE JAVASCRIPT Classes**

**Introduction to Classes**

JavaScript is an *object-oriented programming* (OOP) language we can use to model real-world items. Classes are a tool that developers use to quickly produce similar objects.

Take, for example, an object representing a dog named halley. This dog's name (a key) is "Halley" (a value) and has an age (another key) of 3 (another value). We create the halley object below:

let halley = {

\_name: 'Halley',

\_behavior: 0,

get name() {

return this.\_name;

},

get behavior() {

return this.\_behavior;

},

incrementBehavior() {

this.\_behavior++;

}

}

Now, imagine you own a dog daycare and want to create a catalog of all the dogs who belong to the daycare. Instead of using the syntax above for every dog that joins the daycare, we can create a Dog class that serves as a template for creating new Dog objects. For each new dog, you can provide a value for their name.

As you can see, classes are a great way to reduce duplicate code and debugging time.

Spend some time identifying the similarities and differences between the Dog class in **main.js** and the code we used to create our halley object in the instructions above.

class Dog {

constructor(name) {

this.\_name = name;

this.\_behavior = 0;

}

get name() {

return this.\_name;

}

get behavior() {

return this.\_behavior;

}

incrementBehavior() {

this.\_behavior ++;

}

}

const halley = new Dog('Halley');

console.log(halley.name); // Print name value to console

console.log(halley.behavior); // Print behavior value to console

halley.incrementBehavior(); // Add one to behavior

console.log(halley.name); // Print name value to console

console.log(halley.behavior); // Print behavior value to console

# Constructor

In the last we created a class called Dog, and used it to produce a Dog object.

Although you may see similarities between class and object syntax, there is one important method that sets them apart. It's called 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;

}

}

* Dog is the name of our class. By convention, we capitalize and CamelCase class names.
* JavaScript will invoke the constructor() method every time we create a new instance of our Dogclass.
* This constructor() method accepts one argument, name.
* Inside of the constructor() method, we use the this keyword. In the context of a class, this refers to an instance of that class. In the Dog class, we use this to set the value of the Dog instance's name property to the name argument.
* Under this.name, we create a property called behavior, which will keep track of the number of times a dog misbehaves. The behavior property is always initialized to zero.

**Instance**

An class *instance* is an object that contains the property names and methods of a class, but with unique property values. Let's look at our Dog class example.

class Dog {

constructor(name) {

this.name = name;

this.behavior = 0;

}

}

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

console.log(halley.name); // Log the name value saved to halley

// Output: 'Halley'

Below our Dog class, we use the new keyword to create an instance of our Dog class.

* We create a new variable named halley that will store an instance of our Dog class.
* We use the new keyword to generate a new instance of the Dog class. The new keyword calls the constructor(), runs the code inside of it, and then returns the new instance.
* We pass the 'Halley' string to the Dogconstructor, which sets the name property to 'Halley'.
* Finally, we log the value saved to the name key in our halley object, which logs 'Halley' to the console.

# Methods

At this point, we have a Dog class that spins up objects with name and behavior properties. Below, we will add getters and a method to bring our class to life.

Class method and getter syntax is the same as it is for objects **except you can not 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++;

}

}

In the example above, we add getter methods for nameand behavior. Notice, we also prepended our property names with underscores (\_name and \_behavior), which indicate these properties should not be accessed directly. Under the getters, we add a method named .incrementBehavior(). When you call .incrementBehavior() on a Dog instance, it adds 1 to the \_behavior property. Between each of our methods, we did not include commas.

# Method Calls

Finally, let's use our new methods to access and manipulate data from Dog instances.

class Dog {

constructor(name) {

this.\_name = name;

this.\_behavior = 0;

}

get name() {

return this.\_name;

}

get behavior() {

return this.\_behavior;

}

incrementBehavior() {

this.\_behavior++;

}

}

const halley = new Dog('Halley');

In the example above, we create the Dog class, then create an instance, and save it to a variable named halley.

The syntax for calling methods and getters on an instance is the same as calling them on an object — append the instance with a period, then the property or method name. For methods, you must also include opening and closing parentheses.

Let's take a moment to create two Dog instances and call our .incrementBehavior() method on one of them.

let nikko = new Dog('Nikko'); // Create dog named Nikko

nikko.incrementBehavior(); // Add 1 to nikko instance's behavior

let bradford = new Dog('Bradford'); // Create dog name Bradford

console.log(nikko.behavior); // Logs 1 to the console

console.log(bradford.behavior); // Logs 0 to the console

In the example above, we create two new Doginstances, nikko and bradford. Because we increment the behavior of our nikko instance, but not bradford, accessing nikko.behavior returns 1 and accessing bradford.behavior returns 0.

**Final Example:**

class Surgeon {

constructor(name, department) {

this.\_name = name;

this.\_department = department;

this.\_remainingVacationDays = 20;

}

get name() {

return this.\_name;

}

get department() {

return this.\_department;

}

get remainingVacationDays() {

return this.\_remainingVacationDays;

}

takeVacationDays(daysOff) {

this.\_remainingVacationDays -= daysOff;

}

}

const surgeonCurry = new Surgeon('Curry', 'Cardiovascular');

const surgeonDurant = new Surgeon('Durant', 'Orthopedics');

console.log(surgeonCurry.name);

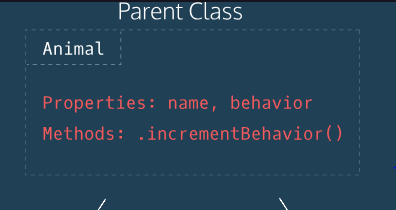
surgeonCurry.takeVacationDays(3);

console.log(surgeonCurry.remainingVacationDays);

# Inheritance I

When multiple classes share properties or methods, they become candidates for inheritance — a tool developers use to decrease the amount of code they need to write.

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.



# Inheritance II

Now, we created a parent class named Animal for two child classes named Cat and Dog.

class Animal {

constructor(name) {

this.\_name = name;

this.\_behavior = 0;

}

get name() {

return this.\_name;

}

get behavior() {

return this.\_behavior;

}

incrementBehavior() {

this.\_behavior++;

}

}

The code below shows the Cat class that 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++;

}

}

**Inheritance III**

We've abstracted the shared properties and methods of our Cat and Dog classes into a parent class called Animal (See below).

class Animal {

constructor(name) {

this.\_name = name;

this.\_behavior = 0;

}

get name() {

return this.\_name;

}

get behavior() {

return this.\_behavior;

}

incrementBehavior() {

this.\_behavior++;

}

}

Now that we have these shared properties and methods in the parent Animal class, we can extend them to the subclass, Cat.

class Cat extends Animal {

constructor(name, usesLitter) {

super(name);

this.\_usesLitter = usesLitter;

}

}

Let's pay special attention to our new keywords: extends and super.

* The extends keyword makes the methods of the animal class available inside the cat class.
* The constructor, called when you create a new Catobject, accepts two arguments, name and usesLitter.
* 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 Catinstances.
* \_usesLitter is a new property that is unique to the Cat class, so we set it in the Cat constructor.

Notice, we call super on the first line of our constructor(), then set the usesLitter property on the second line. In a constructor(), you must always call the super method before you can use the thiskeyword — if you do not, JavaScript will throw a reference error. To avoid reference errors, it is best practice to call super on the first line of subclass constructors.

In the example above, we create a new instance the Cat class, named bryceCat. We pass it 'Bryce' and false for our name and usesLitter arguments. When we call console.log(bryceCat.\_name) our program prints, Bryce.

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

console.log(bryceCat.\_name); // output: Bryce

**Inheritance IV**

When we call extends in a class declaration, all of the parent methods are available to the child class.

class Animal {

constructor(name) {

this.\_name = name;

this.\_behavior = 0;

}

get name() {

return this.\_name;

}

get behavior() {

return this.\_behavior;

}

incrementBehavior() {

this.\_behavior++;

}

}

class Cat extends Animal {

constructor(name, usesLitter) {

super(name);

this.\_usesLitter = usesLitter;

}

}

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

In the example above, our Cat class extends Animal. As a result, the Cat class has access to the Animalgetters and the .incrementBehavior() method.

console.log(bryceCat.name);

Since the extends keyword brings all of the parent's getters and methods into the child class, bryceCat.nameaccesses the name getter and returns the value saved to the name property.

bryceCat.incrementBehavior(); // Call .incrementBehavior() on Cat instance

console.log(bryceCat.behavior); // Log value saved to behavior

The correct answer is 1. But why?

# Inheritance V

In addition to the inherited features, child classes can contain their own properties, getters, setters, and methods.

class Cat extends Animal {

constructor(name, usesLitter) {

super(name);

this.\_usesLitter = usesLitter;

}

get usesLitter() {

return this.\_usesLitter;

}

}

In the example above, we create a usesLitter getter in the Cat class that returns the value saved to \_usesLitter.

One benefit is that when you need to change a method or property that multiple classes share, you can change the parent class, instead of each subclass.

Before we move past inheritance, take a moment to see how we would create an additional subclass, called Dog.

class Dog extends Animal {

constructor(name) {

super(name);

}

}

This Dog class has access to the same properties, getters, setters, and methods as the Dog class we made without inheritance, and is a quarter the size.

# 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.

Take the Date class, for example — you can both create Date instances to represent whatever date you want, and call static methods, like Date.now() which returns the current date, directly from the class. The .now()method is static, so you can call it directly from the class, but not from an instance of the class.

Let's see how to use the static keyword to create a static method called generateName method in our Animal 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];

}

}

In the example above, we create a static method called .generateName() that returns a random name when it's called. Because of the static keyword, we can only access .generateName() by appending it to the Animal class.

We call the .generateName() method with the following syntax:

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

You cannot access the .generateName() method from instances of the Animal class or instances of its subclasses (See below).

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 JAVASCRIPT MODULES**

**Hello Modules**

JavaScript *modules* are reusable pieces of code that can be exported from one program and imported for use in another program.

Modules are particularly useful for separating code with similar logic into files called modules,

* find, fix, and debug code more easily;
* reuse and recycle defined logic in different parts of our application;
* keep information private and protected from other modules;
* and, importantly, prevent pollution of the global namespace and potential naming collisions, by cautiously selecting variables and behavior we load into a program.

**module.exports**

We can get started with modules by defining a module in one file and making the module available for use in another file. Below is an example of how to define a module and how to export it using the statement module.exports.

In **menu.js** we write:

let Menu = {}; // object

Menu.specialty = "Roasted Beet Burger with Mint Sauce"; // obj -> property + value

module.exports = Menu; //export the menu obj as a module.

Let's consider what this code means.

The pattern we use to export modules is thus:

1. Define an object to represent the module.
2. Add data or behavior to the module.
3. Export the module.

**require()**

To make use of the exported module and the behavior we define within it, we import the module. A common way to do this is with the require() function.

We would create a separate file **order.js** and import the Menu module from **menu.js** to **order.js** using require():

In **order.js** we would write:

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

function placeOrder() {

console.log('My order is: ' + Menu.specialty);

}

By calling Menu.specialty we access the property specialty  defined in the Menu module

placeOrder();// function invoke

1. In **order.js** we import the module by creating a variable with const called Menu and setting it equal to the value of the require() function. *We can set this variable to any variable name we like, such as menuItems.*
2. require() is a JavaScript function that enables a module to load by passing in the module's filepath as a parameter.
3. './menu.js' is the argument we pass to the require()function.

Taking a closer look, the pattern to import a module is:

1. Import the module
2. Use the module and its properties within a program.

**module.exports II**

We can also wrap any collection of data and functions in an object, and export the object using module.exports. In **menu.js**, we could write:

let Menu = {};

module.exports = {

specialty: "Roasted Beet Burger with Mint Sauce",

getSpecialty: function() {

return this.specialty;

}

};

In the above code, notice:

1. module.exports exposes the current module as an object.
2. specialty and getSpecialty are properties on the object.

Then in **order.js**, we write:

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

console.log(Menu.getSpecialty());

Here we can still access the behavior in the Menu module.

**Example2:**

**2-airplane.js**

//let myAirplane ={};

// without this also run

module.exports = {

myAirplane: "CloudJet",

displayAirplane: function() {

return this.myAirplane;

}

};

**2-missionControl.js**

const Airplane1 = require('./2-airplane.js');

console.log(Airplane1.displayAirplane());

**export default**

As of ES6, JavaScript has implemented a new more readable and flexible syntax for exporting modules. These are usually broken down into one of two techniques, *default export* and *named exports*.

We'll begin with the first syntax, default export. The default export syntax works similarly to the module.exports syntax, allowing us to export one module per file.

Let's look at an example in **menu.js**.

let Menu = {};

export default Menu;

1. export default uses the JavaScript export statement to export JavaScript objects, functions, and primitive data types.
2. Menu refers to the name of the Menu object, the object that we are setting the properties on within our modules.

When using ES6 syntax, we use export default in place of module.exports.

**import**

ES6 module syntax also introduces the import keyword for importing objects. In our **order.js** example, we import an object like this:

import Menu from './menu';

1. The import keyword begins the statement.
2. The keyword Menu here specifies the name of the variable to store the default export in.
3. from specifies where to load the module from.
4. './menu' is the name of the module to load. When dealing with local files, it specifically refers to the name of the file without the extension of the file.

We can then continue using the Menu module in the **order.js**file.

**Example2:**

Example 2:Continue

**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**

ES6 introduced a second common approach to export modules. In addition to default export, *named exports* allow us to export data through the use of variables.

Let's see how this works. In **menu.js** we would be sure to give each piece of data a distinct variable name:

let specialty = '';

function isVegetarian() {

};

function isLowSodium() {

};

export { specialty, isVegetarian };

1. Notice that, when we use named exports, we are not setting the properties on an object. Each export is stored in its own variable.
2. specialty is a string object, while isVegetarian and isLowSodium are objects in the form of functions. Recall that in JavaScript, every function is in fact a function object.
3. export { specialty, isVegetarian }; exports objects by their variable names. Notice the keyword export is the prefix.
4. specialty and isVegetarian are exported, while isLowSodium is not exported, since it is not specified.

**Named Imports**

To import objects stored in a variable, we use the importkeyword and include the variables in a set of {}.

In the **order.js** file, for example, we would write:

import { specialty, isVegetarian } from './menu';

console.log(specialty);

1. Here specialty and isVegetarian are imported.
2. If we did not want to import either of these variables, we could omit them from the import statement.
3. We can then use these objects as in within our code. For example, we would use specialty instead of Menu.specialty.

**Example:** **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};

**missonControl.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 Named Exports**

Named exports are also distinct in that they can be exported as soon as they are declared, by placing the keyword export in front of variable declarations.

In **menu.js**

export let specialty = '';

export function isVegetarian() {

};

function isLowSodium() {

};

1. The export keyword allows us to export objects upon declaration, as shown in export let specialty and export function isVegetarian() {}.
2. We no longer need an export statement at the bottom of our file, since this behavior is handled above.

# Import Named Imports

To import variables that are declared, we simply use the original syntax that describes the variable name. In other words, exporting upon declaration does not have an impact on how we import the variables.

import { specialty, isVegetarian } from 'menu';

**Example:**

**Airplane.js**

export let availableAirplanes = [

{name: 'AeroJet',

fuelCapacity: 800,

availableStaff: ['pilots', 'flightAttendants', 'engineers', 'medicalAssistance', 'sensorOperators'],

maxSpeed: 1200,

minSpeed: 300

},

{name: 'SkyJet',

fuelCapacity: 500,

availableStaff: ['pilots', 'flightAttendants'],

maxSpeed: 800,

minSpeed: 200

}

];

export let flightRequirements = {

requiredStaff: 4,

requiredSpeedRange: 700

};

export function meetsStaffRequirements(availableStaff, requiredStaff) {

if (availableStaff.length >= requiredStaff) {

return true;

} else {

return false;

}

};

export function meetsSpeedRangeRequirements(maxSpeed, minSpeed, requiredSpeedRange) {

let range = maxSpeed - minSpeed;

if (range > requiredSpeedRange) {

return true;

} else {

return false;

}

};

**missonControl.js**

import {availableAirplanes, flightRequirements, meetsStaffRequirements, meetsSpeedRangeRequirements} 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();

function displaySpeedRangeStatus() {

availableAirplanes.forEach(function(element) {

console.log(element.name + ' meets speed range requirements:' + meetsSpeedRangeRequirements(element.maxSpeed, element.minSpeed, flightRequirements.requiredSpeedRange));

});

}

displaySpeedRangeStatus();

**Export as**

Named exports also conveniently offer a way to change the name of variables when we export or import them. We can do this with the as keyword.

Let's see how this works. In our **menu.js** example

let specialty = '';

let isVegetarian = function() {

};

let isLowSodium = function() {

};

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

In the above example, take a look at the export statement at the bottom of the file.

1. The as keyword allows us to give a variable name an alias as demonstrated in specialty as chefsSpecial and isVegetarian as isVeg.
2. Since we did not give isLowSodium an alias, it will maintain its original name.

**Import as**

To import named export aliases with the as keyword, we add the aliased variable in our import statement.

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

In **orders.js**

1. We import chefsSpecial and isVeg from the Menu object.
2. Here, note that we have an option to alias an object that was not previously aliased when exported. For example, the isLowSodium object that we exported could be aliased with the as keyword when imported: import {isLowSodium as saltFree} from 'Menu';

Another way of using aliases is to import the entire module as an alias:

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

Carte.chefsSpecial;

Carte.isVeg();

Carte.isLowSodium();

1. This allows us to import an entire module from **menu.js** as an alias Carte.
2. In this example, whatever name we exported would be available to us as properties of that module. For example, if we exported the aliases chefsSpecial and isVeg, these would be available to us. If we did not give an alias to isLowSodium, we would call it as defined on the Cartemodule.

**Airplane.js**

let availableAirplanes = [

{name: 'AeroJet',

fuelCapacity: 800,

availableStaff: ['pilots', 'flightAttendants', 'engineers', 'medicalAssistance', 'sensorOperators'],

maxSpeed: 1200,

minSpeed: 300

},

{name: 'SkyJet',

fuelCapacity: 500,

availableStaff: ['pilots', 'flightAttendants'],

maxSpeed: 800,

minSpeed: 200

}

];

let flightRequirements = {

requiredStaff: 4,

requiredSpeedRange: 700

};

function meetsStaffRequirements(availableStaff, requiredStaff) {

if (availableStaff.length > requiredStaff) {

return true;

} else {

return false;

}

};

function meetsSpeedRangeRequirements(maxSpeed, minSpeed, requiredSpeedRange) {

let range = maxSpeed - minSpeed;

if (range > requiredSpeedRange) {

return true;

} else {

return false;

}

};

export { availableAirplanes as aircrafts, flightRequirements as flightReqs, meetsStaffRequirements as meetsStaffReqs , meetsSpeedRangeRequirements as meetsSpeedRangeReqs};

**missonControl.js**

import {aircrafts, flightReqs, meetsStaffReqs, meetsSpeedRangeReqs} from './airplane';

function displayFuelCapacity() {

aircrafts.forEach(function(element) {

console.log('Fuel Capacity of ' + element['name'] + ': ' + element['fuelCapacity']);

});

}

displayFuelCapacity();

function displayStaffStatus() {

aircrafts.forEach(function(element) {

console.log(element['name'] + ' meets staff requirements: ' + meetsStaffReqs(element['availableStaff'], flightReqs['requiredStaff']) );

});

}

displayStaffStatus();

function displaySpeedRangeStatus() {

aircrafts.forEach(function(element) {

console.log(element['name'] + ' meets speed range requirements:' + meetsSpeedRangeReqs(element['maxSpeed'], element['minSpeed'], flightReqs['requiredSpeedRange']));

});

}

displaySpeedRangeStatus();

# Combining Export Statements

We can also use named exports and default exports together. In **menu.js**:

let specialty = '';

function isVegetarian() {

};

function isLowSodium() {

};

function isGlutenFree() {

};

export { specialty as chefsSpecial, isVegetarian as isVeg };

export default isGlutenFree;

Here we use the keyword export to export the named exports at the bottom of the file. Meanwhile, we export the isGlutenFree variable using the export default syntax.

This would also work if we exported most of the variables as declared and exported others with the export default syntax.

export let Menu = {};

export let specialty = '';

export let isVegetarian = function() {

};

export let isLowSodium = function() {

};

let isGlutenFree = function() {

};

export default isGlutenFree;

Here we use the export keyword to export the variables upon declaration, and again export the isGlutenFree variable using the export default syntax

While it's better to avoid combining two methods of exporting, it is useful on occasion. For example, if you suspect developers may only be interested in importing a specific function and won't need to import the entire default export.

# Combining Import Statements

We can import the collection of objects and functions with the same data.

We can use an import keyword to import both types of variables as such:

import { specialty, isVegetarian, isLowSodium } from './menu';

import GlutenFree from './menu';

**Example:**

**Airplane.js**

export let availableAirplanes = [

{name: 'AeroJet',

fuelCapacity: 800,

availableStaff: ['pilots', 'flightAttendants', 'engineers', 'medicalAssistance', 'sensorOperators'],

maxSpeed: 1200,

minSpeed: 300

},

{name: 'SkyJet',

fuelCapacity: 500,

availableStaff: ['pilots', 'flightAttendants'],

maxSpeed: 800,

minSpeed: 200

}

];

export let flightRequirements = {

requiredStaff: 4,

requiredSpeedRange: 700

};

export function meetsStaffRequirements(availableStaff, requiredStaff) {

if (availableStaff.length >= requiredStaff) {

return true;

} else {

return false;

}

};

export function meetsSpeedRangeRequirements(maxSpeed, minSpeed, requiredSpeedRange) {

let range = maxSpeed - minSpeed;

if (range > requiredSpeedRange) {

return true;

} else {

return false;

}

};

export default meetsSpeedRangeRequirements;

**missonControl.js**

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

import meetsSpeedRangeRequirements 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();

function displaySpeedRangeStatus() {

availableAirplanes.forEach(function(element) {

console.log(element.name + ' meets speed range requirements:' + meetsSpeedRangeRequirements(element.maxSpeed, element.minSpeed, flightRequirements.requiredSpeedRange));

});

}

displaySpeedRangeStatus();

**DEBUGGING JAVASCRIPT CODE**

# Debugging Overview

Any programmer will tell you that it is incredibly common to be making great progress working through a coding problem when all of a sudden an error like this gets thrown at you:

/home/ccuser/workspace/learn-javascript-debugging-code/app.js:1

"my\_name".capitalize();

^

TypeError: "my\_name".capitalize is not a function

...

Alternatively, you might have finally finished implementing a complex function only to find it doesn't work at all as you expected:

console.log('The square root of 4 is ' + squareRoot(4));

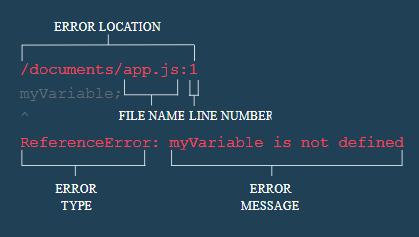
=> 'The square root of 4 is 1.87878787'

If any of these things happen to you, don't fret! In our article <https://news.codecademy.com/errors-in-code-think-differently/>

# Error Stack Traces

You might recognize errors as the scary red text that appears on your screen when you try to run broken code. A piece of software, called a **compiler**, is trying to **translate your code** so that **your computer can understand** and run it. However, the compiler is coming across a piece of **code that it can't interpret.** As a result, it **throws an error back** to you to let you know that it has to stop and why.

This information is logged as an **error stack trace** — a **printed message** containing **information about where the error** occurred, what type of error was thrown, and a description of the error.



**Reading Error Stack Traces**

Now that we know what information we can get from an error stack trace.

/home/ccuser/workspace/learn-javascript-debugging-code/app.js:1

myVariable;

^

ReferenceError: myVariable is not defined

...

1. **In what line did the error occur?** You can almost always find this information on the first line of the stack trace in the following format <file path>/<file name>:<line number>.
2. **What type of error was thrown?** The first word on the fifth line of the stack trace is the type of error that was thrown. In this case, the type of error was ReferenceError.
3. **What is the error message?** The rest of the fifth line after the error type provides an error message, describing what went wrong.

# JavaScript Error Types

Here are three common error types:

* **SyntaxError**: This error will be thrown when a type creates **invalid code** — code that cannot be interpreted by the compiler. When this error is thrown, scan your code to make sure you properly opened and closed all **brackets, braces, and parentheses** and that you didn't include any invalid **semicolons**.
* **ReferenceError**: This error will be thrown if you **try to use a variable that does not exist.** Declared that variable.
* **TypeError**: This error will be thrown if you attempt to perform an operation on a value of the wrong type. For example, **if we tried to use a string method on a number**, it would throw a TypeError.

**There are seven types of built-in JavaScript** errors in total. You can find more information about all of them at <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error>

[**EvalError**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/EvalError)

Creates an instance representing an error that **occurs regarding the global function**[**eval()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/eval).

[**InternalError**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/InternalError)

Creates an instance representing an error that occurs when an internal error in the JavaScript engine is thrown. E.g. **"too much recursion".**

[**RangeError**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RangeError)

Creates an instance representing an error **that occurs when a numeric variable or parameter is outside of its valid range.**

[**URIError**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/URIError)

Creates an instance representing an error that occurs when [**encodeURI()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURI) or [**decodeURI()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/decodeURI)are passed invalid parameters.

**Example:**

myVariable++;

// 1 - What type of error will be thrown on the line above:

ReferenceError

const myString = 42;

myString.substring(0);

// 2 - What type of error will be thrown on the line above:

TypeError

const myRandomNumber; = Math.random();

// 3 - What type of error will be thrown on the line above:

SyntaxError

**Debugging Errors**

Here's a process for efficiently working through your code's errors one by one:

1. **Run your code.** Using the first error's **stack trace**, identify **the error's type**, **description,** and **location**.
2. Go to the file name **and line number indicated** by the error stack trace. Using the error type and description, identify the bug in your code.
3. Fix the bug and re-run your code.
4. Repeat steps 1-3 until your code no longer throws any errors.

# Locating Silent Bugs

Errors thrown by the computer are really useful because they identify the bug type and location for you right away. However**, even if your code runs error-free, it is not necessarily bug-free.**

You may find that your code is consistently returning incorrect values without throwing any errors. A lack of thrown errors does not mean your code logic is completely correct.

An incredibly powerful tool for locating bugs is a method you likely learned very early on in your JavaScript journey: console.log()!

By adding print statements to our code, we can identify where things have gone wrong.

**Debugging with console.log()**

Let's synthesize our workflow from the previous exercise into a reusable set of debugging steps.

1. Go to the beginning of the malfunctioning code**. Print out all starting variables, existing values,** and arguments using console.log**(). If the values are what you expect, move on to the next piece of logic** in the code. If not, you have identified a bug and should skip to step 3.
2. After the next piece of logic in your code, add console.log() statements to ensure updated variables have the values that you now expect and that the block of code is being executed. If that logic is executing properly, continue repeating this step until you find a line not working as expected, then move to step 3.
3. Fix the identified bug and run your code again. If it now works as expected, you've finished debugging! If not, continue stepping through your code using step 2 until it does.

**Example:**

// Returns the string whose first letter is later in the alphabet. If both first letters are equal, returns null.

function getLaterFirstLetter(string1, string2) {

const firstLetter1 = string1.charAt(0);

//console.log("code checking : ", firstLetter1)

const firstLetter2 = string2.charAt(0);

//console.log("code checking : ", firstLetter2)

if (firstLetter1 === firstLetter2) {

return null;

} else if (firstLetter1 < firstLetter2) {

return string2;

// console.log(string2)

} else {

return string1;

}

}

// Should return "blueberry"

console.log("getLaterFirstLetter('avocado', 'blueberry') returns: " + getLaterFirstLetter('avocado', 'blueberry'));

// Should return "zebra"

console.log("getLaterFirstLetter('zebra', 'aardvark') returns : " + getLaterFirstLetter('zebra', 'aardvark'));

// Should return null

console.log("getLaterFirstLetter('astro', 'afar') returns: " + getLaterFirstLetter('astro', 'afar'));

# Finding Documentation

Sometimes once you've tracked down a bug, you might still be confused on how to fix it! Whenever you want to know more about how JavaScript works and what it can do, the best place to go is **documentation**. <https://developer.mozilla.org/en-US/docs/Web/JavaScript>

For example, if we wanted more information on the Numberobject's .isNan() method, we could Google "MDN isNan" and then click the link to the MDN page. If we were looking to see a list of all of the String built-in methods, we might Google "MDN String", click the link to MDN, and then scroll down to the "Methods" section of the documentation.

# Stack Overflow

At this point, you might be thinking to yourself, documentation is good and all, but there's no way it will solve all of my issues! And we totally agree. All programming languages have difficult problems and strange edge cases that appear unexpectedly and are sometimes impossible to solve alone.

If you are ever stuck trying to solve a coding problem, we strongly encourage you to Google for a solution. One of the best sites you will see appear in the search results is [Stack Overflow](https://stackoverflow.com/questions/51165/how-to-sort-strings-in-javascript).

For example, say you are stumped trying to write an alphabetize function. If you search "alphabetize string JavaScript" on Google, [this Stack Overflow search result](https://stackoverflow.com/questions/51165/how-to-sort-strings-in-javascript) will appear. You can quickly scan through the answers on it to see which ones work for you.

**LEARN JAVASCRIPT: ERROR HANDLING**

# Introduction to Error Handling

There are **two categories of programming mistakes:** those that don't prevent our code from running and those that do.

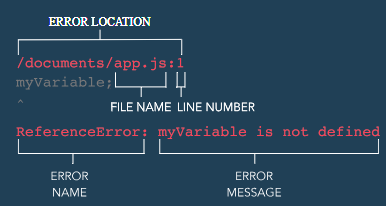
Sometimes, we've written code that successfully returns a value but a different value from what we expected. Our program continues running, and we might not even realize anything went wrong until much later.

For example, what if we tried to move our soup to the table but dropped it because it was too hot? Then our soup-making process is over— there would be no soup.

We can't always stop errors before they occur, but we can include a backup plan in our program to anticipate and respond to the errors to ensure that our program continues running. Error handling is the process of programmatically anticipating and addressing errors. In JavaScript, we handle errors using the keywords try and catch. We try to move the soup to the table, making sure there's someone or something nearby to catch the soup in case we drop it.

# Runtime Errors

Errors contain useful messages that that tell us why our program isn't working or why the error was thrown. **When an error is thrown, our program stops running** and the console displays red text of the error message like so:



When we execute code and a line of code throws an error, that error is referred to as a *runtime error.* There are built-in runtime errors include:

* ReferenceError: when a variable or function cannot be found.
* TypeError: when a value is not a valid type, see the example below:

const reminder = 'Reduce, Reuse, Recycle';

reminder = 'Save the world';

// TypeError: Assignment to constant variable.

console.log('This will never be printed!');

In the example above, when we try to reassign a constant variable reminder, the TypeError is thrown. Code that is written after a thrown runtime error will not be evaluated, **so the console.log() statement will not be evaluated.**

# Constructing an Error

**JavaScript errors are objects that have a name and message property**. Previously, we've seen **how built-in errors alert us** about common mistakes in our code. But, what if we need an **error message** that isn't covered by the built-in errors?

we need to inform a user that **a string passed in as an argument is too short with a custom message**. Such a message **isn't covered by a built-in error,** but we could use **the Error function to make our own error object** with a message that is unique to our program!

console.log(Error('Your password is too weak.'));

// Prints: Error: Your password is too weak.

The Error function **takes an argument** of a string which becomes **the value of the error's message property**.

You might also see errors created with the the new keyword. Both methods will lead to the same functionality. Take a look:

console.log(new Error('Your password is too weak.'));

// Prints: Error: Your password is too weak.

Keep in mind that creating an error is not the same as throwing an error. **A thrown error will cause the program to stop running.**

# The throw Keyword

Creating an error doesn't cause our program to stop — remember, an error must be thrown for it to halt the program.

To throw an error in JavaScript, we use the throw keyword like so:

throw Error('Something wrong happened');

// Error: Something wrong happened

When we use the throw keyword, **the error is thrown and code after throw statement will not execute. Take for example:**

throw Error('Something wrong happened');

// Error: Something wrong happened

console.log('This will never run'); // this part will not execute

the console.log() statement will not **run (just like when a built-in JavaScript error was thrown!).**

**The try...catch Statement**

In JavaScript, we use try...catch statement to **anticipate and handle errors by writing code to address the error and allow our program to continue running** .

try {

throw Error('This error will get caught');

} catch (e) {

console.log(e); // represent thrown error

}

// Prints: This error will get caught

console.log('The thrown error that was caught in the try...catch statement!');

// Prints: 'The thrown error that was caught in the try...catch statement!'

* We have code that follows try inside curly braces {}known as the *try block*.
* Inside the try block we insert code that we **anticipate might throw an error.**
* Since we want to see what happens if an error is thrown in the try block, we throw an error with the message 'This error will get caught'.

Generally speaking, in a try...catch statement, we evaluate code in the try block and if the code throws an error, the code inside the catch block will handle the error for us.

# Handling with try...catch

In the previous exercise we caught an error that we threw,

but we can also use a try...catch**statement to handle built-in errors that are thrown by the JavaScript engine** that is reading and evaluating our code.

const someVar = 'Cannot be reassigned';

try {

someVar = 'Still going to try';

} catch(e) {

console.log(e);

}

// Prints: TypeError: Assignment to constant variable.

In the example above, **we didn't use the throw keyword to throw a custom error**, rather **we tried to re-assign** a constvariable and a TypeError was thrown. Then, in our catch block, we logged the error to the console.

Using a try...catch statement are **used when the is data from an external source that's not written directly in our program.**

* Let's say we expect to grab an array from a database but the information we get back is a string.
* In our program, we could have a function that only works on arrays.
* If that function was called with a string instead of an array we would **get an error and our program** would stop running!

However, we can use a try...catch statement to handle the thrown error for us which allows our program to continue running and we receive a message knowing what went wrong.

**Example:**

/\* there is a function capAllElements() that takes an array of elements and capitalizes each element.

Currently, it's written in a way the function will execute regardless of what argument is passed in but if the argument isn't an array, an error is thrown and our program stops running. Run your code to see what error shows up in the console.\*/

function capAllElements(arr){

    try {

arr.forEach((el, index, array) => {

array[index] = el.toUpperCase();

} )

}

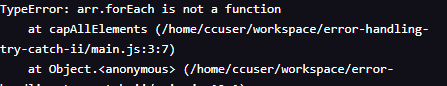
catch(e){

console.log(e);

}

}

capAllElements('Incorrect argument');



**JAVASCRIPT PROMISES**

# Introduction

An **asynchronous operation** is one that **allows** the computer to **"move on" to other tasks** while waiting for the asynchronous operation to complete.

for example: Cleaning our house, involves asynchronous operations such as a dishwasher washing our dishes or a washing machine washing our clothes. While we wait on the completion of those operations, we're free to do other chores.

Similarly, web development makes use of asynchronous operations. Operations like making a network request or querying a database can be time-consuming, but JavaScript allows us to execute other tasks while awaiting their completion.

**What is a Promise?**

**Promises are objects that represent the eventual outcome** of an asynchronous operation. A Promise object can be in one of three states:

* **Pending**: The initial state— the operation has not completed yet.
* **Fulfilled**: The operation has completed successfully and the promise now has a *resolved value*. For example, a request's promise might resolve with a JSON object as its value.
* **Rejected**: The operation has failed and the promise has a reason for the failure. This reason is usually an Error of some kind.

We refer to **a promise as *settled* if it is no longer pending**— it is either fulfilled or rejected.

* **Pending**: The dishwasher is running but has not completed the washing cycle.
* **Fulfilled**: The dishwasher has completed the washing cycle and is full of clean dishes.
* **Rejected**: The dishwasher encountered a problem (it didn't receive soap!) and returns unclean dishes.

If our dishwashing promise is fulfilled, we'll be able to perform related tasks, such as unloading the clean dishes from the dishwasher.

If it's rejected, we can take alternate steps, such as running it again with soap or washing the dishes by hand.

**All promises eventually settle**, enabling us to **write logic** for what to do if the promise fulfills or if it rejects.

**Constructing a Promise Object**

Let’s **construct a promise!** To create a new Promise **object**, we use the new keyword and the Promise **constructor method**:

const executorFunction = (resolve, reject) => { }; /\*promises object\*/

const myFirstPromise = new /\*contructor\*/Promise(executorFunction /\*promises object\*/);

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.

The executor function has **two function parameters**, usually referred to as the **resolve() and reject() functions.** The resolve() and reject() functions aren't defined by the programmer. When the Promise constructor runs, JavaScript will pass **its own** resolve() and reject() functions into the executor function.

* 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().

Let's look at an example executor function in a Promiseconstructor:

const executorFunction = (resolve, reject) => {

if (someCondition) {

resolve('I resolved!');

} else {

reject('I rejected!');

}

}

const myFirstPromise = new Promise(executorFunction);

Let's break down what's happening above:

* We declare a variable myFirstPromise
* myFirstPromise is constructed using new Promise() **which is the Promise constructor method.**
* executorFunction() is passed to the constructor and has two functions as parameters: resolve and reject.
* If someCondition evaluates to true, we invoke resolve()with the string 'I resolved!'
* If not, we invoke reject() with the string 'I rejected!'

In our example, myFirstPromise resolves or rejects based on a simple condition, but, in practice, promises settle based on the results of asynchronous operations.

For example, a database request may fulfill with the data from a query or reject with an error thrown.

**Example:**

const inventory = {

sunglasses: 1900,

pants: 1088,

bags: 1344

};

// Write your code below:

const myExecutor = (resolve, reject) => {

if(inventory.sunglasses > 0 )

{

resolve('Sunglasses order processed.');

}

else{

reject('That item is sold out.');

}

}

const orderSunglasses = () => {

return new Promise(myExecutor);

}

const orderPromise = orderSunglasses();

console.log(orderPromise);

# The Node setTimeout() Function

Knowing how to construct a promise is useful*, but most of the time, knowing how to*consume*, or use, promises will be key*.

Rather than constructing promises, you'll be handling Promise objects returned to you as the result of an asynchronous operation.

These promises will start off pending but settle eventually.

Moving forward, we'll **be simulating this by providing you with functions** that return promises which settle after some time.

To accomplish this, we'll be using setTimeout(). setTimeout() is a Node API 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.

const delayedHello = () => {

console.log('Hi! This is an asynchronous greeting!');

};

setTimeout(delayedHello, 2000);

Here, we invoke setTimeout() with the callback function delayedHello() and 2000. In at least two seconds delayedHello() will be invoked. But why is it "at least" two seconds and not exactly two seconds?

This delay is performed asynchronously— the rest of our program won't stop executing during the delay. Asynchronous JavaScript uses something called the event-loop.

After two seconds, delayedHello() is added to a line of code waiting to be run. Before it can run, any synchronous code from the program will run.

Next, any code in front of it in the line will run. This means it might be more than two seconds before delayedHello() is actually executed.

Let's look at how we'll be using setTimeout() to construct asynchronous promises:

const returnPromiseFunction = () => {

return new Promise((resolve, reject) => {

setTimeout(( ) => {resolve('I resolved!')}, 1000);

});

};

const prom = returnPromiseFunction();

In the example code, we invoked returnPromiseFunction() which returned a promise. We assigned that promise to the variable prom. Similar to the asynchronous promises you may encounter in production, prom will initially have a status of pending.

**Example:**

console.log("This is the first line code in app.js.");

// Keep the line above as the first line of code

// Write your code here:

const usingSTO = () => {

console.log('This is the second line of synchronous code.');

}

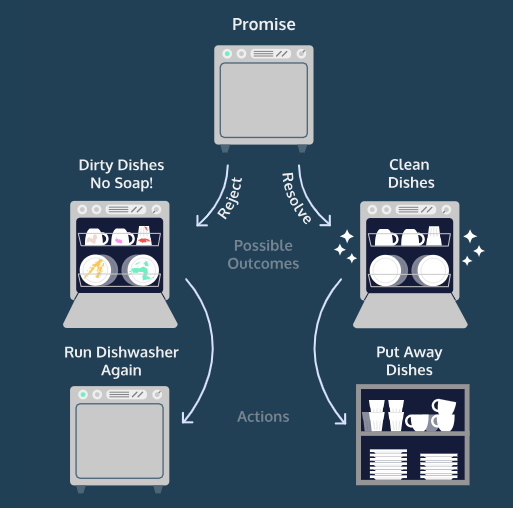
setTimeout(usingSTO, 3000);

// Keep the line below as the last line of code:

console.log("This is the last line of code in app.js.");

**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.



In the case of our dishwasher promise, the dishwasher will run **then**:

* If our promise rejects, this means we have dirty dishes, and we'll add soap and run the dishwasher again.
* If our promise fulfills, this means we have clean dishes, and we'll put the dishes away.

.then() is a higher-order function— it **takes two callback functions as arguments**. We refer to these callbacks as *handlers*. When the promise settles, the appropriate handler will be invoked with that settled value.

* The first handler, sometimes called onFulfilled, is a *success handler*, and it should contain the logic for the promise resolving.
* The second handler, sometimes called onRejected, is a *failure handler*, and it should contain the logic for the promise rejecting.

Important to know, ---

1. We can invoke .then() with one, both, or neither handler! This allows for flexibility, but it can also make for tricky debugging.
2. If the appropriate handler is not provided, instead of throwing an error, .then() will just return a promise with the same settled value as the promise it was called on.
3. One important feature of .then() is that it always returns a promise.

**The onFulfilled and onRejected Functions**

To handle a "successful" promise, or a promise that resolved, we invoke .then() on the promise, passing in a success handler callback function:

const prom = new Promise((resolve, reject) => {

resolve('Yay!');

});

const handleSuccess = (resolvedValue) => {

console.log(resolvedValue);

};

prom.then(handleSuccess); // Prints: 'Yay!'

Let's break down what's happening in the example code:

* prom is a promise which will resolve to 'Yay!'.
* We define a function, handleSuccess(), which prints the argument passed to it.
* We invoke prom's .then() function passing in our handleSuccess() function.
* **Since prom resolves, handleSuccess() is invoked with prom's resolved value, 'Yay', so 'Yay' is logged to the console.**

With typical promise consumption, we won't know whether a promise will resolve or reject, so we'll need to provide the logic for either case. We can pass both an onFulfilled and onRejected callback to .then().

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);

Let's break down what's happening in the example code:

* prom is a promise which will randomly either resolve with 'Yay!'or reject with 'Ohhh noooo!'.
* We pass two handler functions to .then(). The first will be invoked with 'Yay!' if the promise resolves, and the second will be invoked with 'Ohhh noooo!' if the promise rejects.

Let's write some onFulfilled and onRejectedfunctions!

**Example: 2**

**Library.js**

const inventory = {

sunglasses: 10,

pants: 1088,

bags: 1 //check from here make it 2

};

const checkInventory = (order) => {

return new Promise((resolve, reject) => {

setTimeout(() => {

let inStock = order.every(item => inventory[item[0]] >= item[1]);

if (inStock) {

resolve(`Thank you. Your order was successful.`);

} else {

reject(`We're sorry. Your order could not be completed because some items are sold out.`);

}

}, 1000);

})

};

module.exports = { checkInventory };

**app.js**

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

const order = [['sunglasses', 1], ['bags', 2]];

// Write your code below:

const handleSuccess = (resolvedValue) => {

console.log(resolvedValue);

};

const handleFailure = (rejectReason) => {

console.log(rejectReason);

};

checkInventory(order)

.then(handleSuccess, handleFailure);

**Using catch() with Promises**

**One way to write cleaner code** is to follow a principle called***separation of concerns***. Separation of concerns means organizing code into distinct sections each handling a specific task. It enables us to quickly navigate our code and know where to look if something isn't working.

**Remember, .then() will return a promise** with the same settled value as the promise **it was called on if no appropriate handler was provided**. This implementation allows us to separate our resolved logic from our rejected logic. Instead of passing both handlers into one .then(), we can chain a second .then() with a failure handler to a first .then() with a success handler and both cases will be handled.

prom

.then((resolvedValue) => {

console.log(resolvedValue);

})

.then(null, (rejectionReason) => {

console.log(rejectionReason);

});

The .catch() function takes only one argument, onRejected. In the case of a rejected promise.

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

Let's look an example using .catch():

prom

.then((resolvedValue) => {

console.log(resolvedValue);

})

.catch((rejectionReason) => {

console.log(rejectionReason);

});

Let's break down what's happening in the example code:

* prom is a promise which randomly either resolves with 'Yay!' or rejects with 'Ohhh noooo!'.
* We pass a success handler to .then() and a failure handler to .catch().
* If the promise resolves, .then()'s success handler will be invoked with 'Yay!'.
* If the promise rejects, .then() will return a promise with the same rejection reason as the original promise and .catch()'s failure handler will be invoked with that rejection reason.

**Example: 2**

**Library.js**

const inventory = {

sunglasses: 2,

pants: 1088,

bags: 1344

};

const checkInventory = (order) => {

return new Promise((resolve, reject) => {

setTimeout(() => {

let inStock = order.every(item => inventory[item[0]] >= item[1]);

if (inStock) {

resolve(`Thank you. Your order was successful.`);

} else {

reject(`We're sorry. Your order could not be completed because some items are sold out.`);

}

}, 1000);

});

};

module.exports = {checkInventory};

**app.js**

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

const order = [['sunglasses', 1], ['bags', 2]];

const handleSuccess = (resolvedValue) => {

console.log(resolvedValue);

};

const handleFailure = (rejectReason) => {

console.log(rejectReason);

};

// Write your code below:

checkInventory(order)

.then(handleSuccess)

.catch(handleFailure)

**Chaining Multiple Promises**

One common pattern we'll see with asynchronous programming is **multiple operations which depend on each other** to execute or that must be executed in a certain order. We might make one request to a database and use the data returned to us to make another request and so on! Let's illustrate this with another cleaning example, washing clothes:

We take our dirty clothes **and put them in the washing machin**e. If the clothes are cleaned, **then** we'll want to **put them in the dryer**. After the dryer runs, if the clothes are dry, ***then we can fold them and put them*** away.

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

firstPromiseFunction()

.then((firstResolveVal) => {

return secondPromiseFunction(firstResolveVal);

})

.then((secondResolveVal) => {

console.log(secondResolveVal);

});

Let's break down what's happening in the example:

* We invoke a function firstPromiseFunction() which returns a promise.
* We invoke .then() with an anonymous function as the success handler.
* Inside the success handler we **return** a new promise— the result of invoking a second function, secondPromiseFunction() with the first promise's resolved value.
* We invoke a second .then() to handle the logic for the second promise settling.
* Inside that .then(), we have a success handler which will log the second promise's resolved value to the console.

In order for our chain to work properly, we had to return the promise secondPromiseFunction(firstResolveVal). This ensured that the return value of the first .then() **was our second promise rather than the default return** of a new promise with the same settled value as the initial.

**Example:**

**Library.js**

const store = {

sunglasses: {

inventory: 817,

cost: 9.99

},

pants: {

inventory: 236,

cost: 7.99

},

bags: {

inventory: 17,

cost: 12.99

}

};

/\*

checkInventory() expects an order argument and returns a promise. If there are enough items in stock to fill the order, the promise will resolve to an array. The first element in the resolved value array will be the same order and the second element will be the total cost of the order as a number.

\*/

const checkInventory = (order) => {

return new Promise ((resolve, reject) => {

setTimeout(()=> {

const itemsArr = order.items;

//imp to understand

let inStock = itemsArr.every(item => store[item[0]].inventory >= item[1]);

if (inStock){

let total = 0;

itemsArr.forEach(item => {

//imp to understand

total += item[1] \* store[item[0]].cost

});

console.log(`All of the items are in stock. The total cost of the order is ${total}.`);

resolve([order, total]);

} else {

reject(`The order could not be completed because some items are sold out.`);

}

}, generateRandomDelay());

});

};

const processPayment = (responseArray) => {

const order = responseArray[0];

const total = responseArray[1];

return new Promise ((resolve, reject) => {

setTimeout(()=> {

let hasEnoughMoney = order.giftcardBalance >= total;

// For simplicity we've omited a lot of functionality

// If we were making more realistic code, we would want to update the giftcardBalance and the inventory

if (hasEnoughMoney) {

console.log(`Payment processed with giftcard. Generating shipping label.`);

let trackingNum = generateTrackingNumber();

resolve([order, trackingNum]);

} else {

reject(`Cannot process order: giftcard balance was insufficient.`);

}

}, generateRandomDelay());

});

};

const shipOrder = (responseArray) => {

const order = responseArray[0];

const trackingNum = responseArray[1];

return new Promise ((resolve, reject) => {

setTimeout(()=> {

resolve(`The order has been shipped. The tracking number is: ${trackingNum}.`);

}, generateRandomDelay());

});

};

// This function generates a random number to serve as a "tracking number" on the shipping label. In real life this wouldn't be a random number

function generateTrackingNumber() {

return Math.floor(Math.random() \* 1000000);

}

// This function generates a random number to serve as delay in a setTimeout() since real asynchrnous operations take variable amounts of time

function generateRandomDelay() {

return Math.floor(Math.random() \* 2000);

}

module.exports = {checkInventory, processPayment, shipOrder};

**app.js**

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);

});

**Avoiding Common Mistakes**

It can still be easy to make mistakes. we'll go over two common mistakes with promise composition.

**Mistake 1:** Nesting promises instead of chaining them.

returnsFirstPromise()

.then((firstResolveVal) => {

return returnsSecondValue(firstResolveVal)

.then((secondResolveVal) => {

console.log(secondResolveVal);

})

})

Instead of having a clean chain of promises, we've nested the logic for one inside the logic of the other. Imagine if we were handling five or ten promises!

**Mistake 2:** Forgetting to return a promise.

returnsFirstPromise()

.then((firstResolveVal) => {

returnsSecondValue(firstResolveVal)

})

.then((someVal) => {

console.log(someVal);

})

**Since forgetting to return our promise won't throw an error, this can be a really tricky thing to debug!**

**Example:**

**App.js refactor code**

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

const order = {

items: [['sunglasses', 1], ['bags', 2]],

giftcardBalance: 79.82

};

// Refactor the code below:

/\*

checkInventory(order)

.then((resolvedValueArray) => {

return processPayment(resolvedValueArray)

.then((resolvedValueArray) => {

return shipOrder(resolvedValueArray)

.then((successMessage) => {

console.log(successMessage);

});

});

});

\*/

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);

});

**Using Promise.all()**

When done correctly, promise composition is a great way to handle situations where asynchronous operations depend on each other or execution order matters. What if we're dealing with multiple promises, but we don't care about the order?

For us to consider our house clean, we need our clothes to dry, our trash bins emptied, and the dishwasher to run. We need **all** of these tasks to complete but not in any particular order. Furthermore, since they're all getting done asynchronously, they should really all be happening at the same time!

To maximize efficiency we should use ***concurrency***, **multiple asynchronous operations happening together.** With promises, we can do this with the function Promise.all().

Promise.all() **accepts an array of promises** as its **argument** and **returns a single promise.** That single promise will **settle** in one **of two ways:**

* If every promise in the argument array resolves, the single promise returned from Promise.all() will **resolve** with **an array containing the resolve value** from each promise in the argument array.
* If any promise from the argument **array rejects,** the single promise returned from Promise.all() **will immediately reject** with the reason that promise rejected. This behavior is sometimes referred to as ***failing fast*.**

Let's look at a code example:

let myPromises = Promise.all([returnsPromOne(), returnsPromTwo(), returnsPromThree()]);

myPromises

.then((arrayOfValues) => {

console.log(arrayOfValues);

})

.catch((rejectionReason) => {

console.log(rejectionReason);

});

//This condition can be applicable to Board Exams critria

Let's break down what's happening:

* We declare myPromises assigned to invoking Promise.all().
* We invoke Promise.all() with an array of three promises— the returned values from functions.
* We invoke .then() with a success handler which will print the array of **resolved** **values if each promise resolves successfully**.
* We invoke .catch() with a failure handler which will print the **first rejection message if any promise rejects.**

**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);

}

**App.js**

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

const onFulfill = (itemsArray) => {

// console.log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"+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);

**---Ques Quiz---**

1. True or False: The .then() method returns a Promise.

* True

1. What is the fulfilled value of Promise.all()?

* An Array

1. What value is printed to the console ?

* Object

const asyncHello = new Promise((resolve, reject) => {

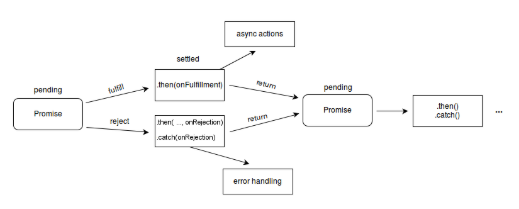
setTimeout(resolve, 1000, 'Hello!');

});

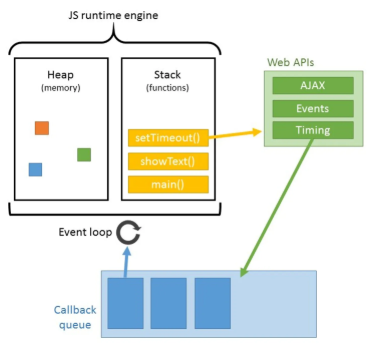
console.log(typeof asyncHello);

**Refrence:**

[**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Promise**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)



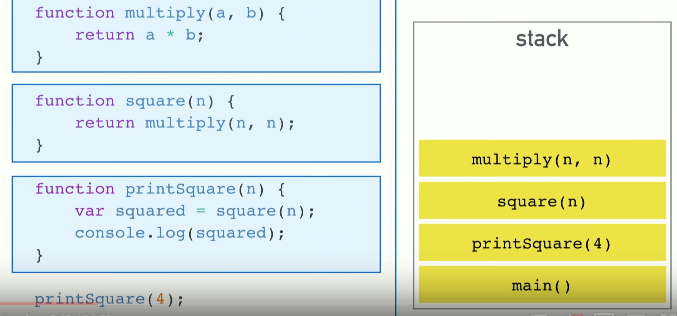
**Event Loops:**



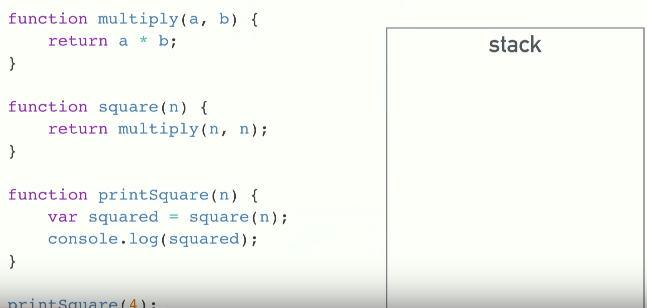
**java script is single threaded prog.lang.**



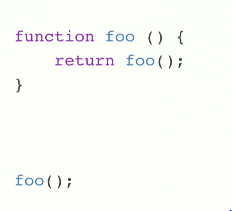
**Call stack**

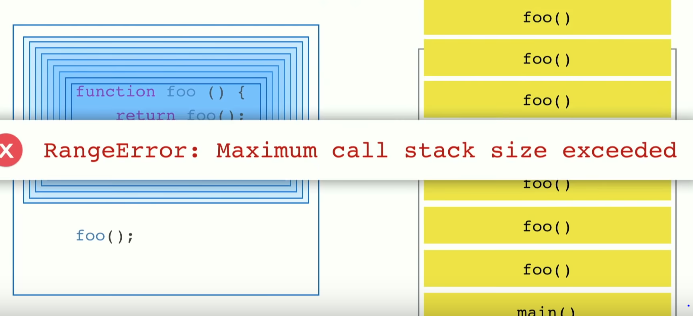


**call stack return 🡪 stack empty**



**Infinite 🡪call stack size exceeded**





**ASYNC AWAIT**

# Introduction

Often in web development, we need to handle asynchronous actions— actions we can wait on while moving on to other tasks. *We make requests to networks, databases, or any number of similar operations.* **JavaScript is non-blocking:** instead of stopping the execution of code while it waits, JavaScript uses an [event-loop](https://youtu.be/8aGhZQkoFbQ) ->

<https://www.youtube.com/watch?v=8aGhZQkoFbQ&feature=youtu.be>

which allows it to efficiently execute other tasks while it awaits the completion of these asynchronous actions.

Originally, JavaScript used callback functions to handle asynchronous actions. The problem with callbacks is that they encourage complexly nested code which quickly becomes difficult to read, debug, and scale.

With ES6, JavaScript integrated native [promises](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) which allow us to write significantly more readable code. JavaScript is continually improving, and ES8 provides a new syntax for handling our asynchronous action, *async...await*. The async...await syntax allows us to write asynchronous code that reads similarly to traditional synchronous, imperative programs.

The async...await syntax is [syntactic sugar](https://en.wikipedia.org/wiki/Syntactic_sugar)

(In computer science **syntactic sugar** is syntax within a [programming language](https://en.wikipedia.org/wiki/Programming_language) that is designed to make things easier to read or to express. It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.)

🡪 it doesn't introduce new functionality into the language, but rather introduces a new syntax for using promises and [generators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator).

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator>

Both of these were already built in to the language. Despite this, async...await powerfully improves the readability and scalability of our code. Let's learn how to use it!

**App.js**

const fs = require('fs');

const promisifiedReadfile = require('./promisifiedReadfile');

// Here we use fs.readfile() and callback functions:

fs.readFile('./file.txt', 'utf-8', (err, data) => {

if (err) throw err;

let firstSentence = data;

fs.readFile('./file2.txt', 'utf-8', (err, data) => {

if (err) throw err;

let secondSentence = data;

console.log(firstSentence, secondSentence)

});

});

// Here we use native promises with our "promisified" version of readfile:

let firstSentence

promisifiedReadfile('./file.txt', 'utf-8')

.then((data) => {

firstSentence = data;

return promisifiedReadfile('./file2.txt', 'utf-8')

})

.then((data) => {

let secondSentence = data;

console.log(firstSentence, secondSentence)

})

.catch((err) => {console.log(err)});

// Here we use promisifiedReadfile() again but instead of using the native promise .then() syntax, we declare and invoke an async/await function:

async function readFiles() {

let firstSentence = await promisifiedReadfile('./file.txt', 'utf-8')

let secondSentence = await promisifiedReadfile('./file2.txt', 'utf-8')

console.log(firstSentence, secondSentence)

}

readFiles()

**promisifiedReadfile.js**

const fs = require('fs');

// Below we create a function for reading files that returns a promise. We converted the fs.readfile() function which uses callbacks. Many of the asynchronous functions you'll encounter already return promises, so this extra step is seldom necessary.

const promisifiedReadfile = (file, encoding) =>

new Promise((resolve, reject) => {

fs.readFile(file, encoding, (err, text) => {

            if (err) {

                return reject(err.message);

}

resolve(text);

});

});

module.exports = promisifiedReadfile

**The async Keyword**

**The async keyword is used to write functions that handle asynchronous actions**. We wrap our asynchronous logic inside a function prepended with the async keyword. Then, we invoke that function.

async function myFunc() {

// Function body here

};

myFunc();

we can also create async function expressions:

const myFunc = async () => {

// Function body here

};

myFunc();

**async functions always return a promise.** This means we can use traditional promise syntax, like .then() and .catch with our async functions. An async function will return in one of three ways:

* 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

In the example above, even though we return 5 inside the function body, what's actually returned when we invoke fivePromise() is a promise with a resolved value of 5.

**Example:**

**App.js**

function withConstructor(num){

return new Promise((resolve, reject) => {

if (num === 0){

resolve('zero');

} else {

resolve('not zero');

}

})

}

withConstructor(0)

.then((resolveValue) => {

console.log(` withConstructor(0) returned a promise which resolved to: ${resolveValue}.`);

})

// Write your code below:

async function withAsync(num){

if (num === 0){

return 'zero';

} else {

return 'not zero';

}

}

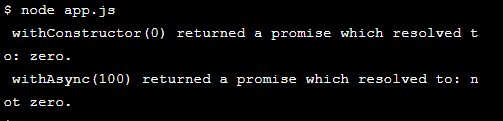
withAsync(100)

.then((resolveValue) => {

console.log(` withAsync(100) returned a promise which resolved to: ${resolveValue}.`);

})

**Output:**



# The await Operator

By itself, it doesn't do much; async functions are almost always used with the additional keyword await inside the function body.

The await keyword can **only be used inside** an asyncfunction. 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.

In the example below, myPromise() is a function that returns a promise which will resolve to the string "I am resolved now!".

async function asyncFuncExample(){

let resolvedValue = await myPromise();

console.log(resolvedValue);

}

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

Within our async function, asyncFuncExample(), we use await to halt our execution until myPromise() is **resolved** and assign its resolved value to the variable resolvedValue. Then we log resolvedValue to the console. We're able to handle the logic for a promise in a way that reads like synchronous code.

**Library.js**

const brainstormDinner = () => {

return new Promise((resolve, reject) => {

console.log(`I have to decide what's for dinner...`)

setTimeout(() => {

console.log(`Should I make salad...?`)

setTimeout(() => {

console.log('Should I make ramen...?')

setTimeout(() => {

console.log('Should I make eggs...?')

setTimeout(() => {

console.log('Should I make chicken...?')

resolve('beans')

}, 1000)

}, 1000)

}, 1000)

}, 1000)

})

}

module.exports = brainstormDinner

**app.js**

const brainstormDinner = require('./library.js')

// Native promise version:

function nativePromiseDinner() {

brainstormDinner().then((meal) => {

     console.log(`I'm going to make ${meal} for dinner.`);

})

}

// async/await version:

//  If there's nothing returned from the function, it will return a promise with a resolved value of undefined.

async function announceDinner() {

// Write your code below:

let resolvedValue = await brainstormDinner();

console.log(`I'm going to make ${resolvedValue} for dinner.`);

}

announceDinner();

# Writing async Functions

We've seen that the await keyword halts the execution of an async function until a promise is no longer pending. Don't forget the await keyword! It may seem obvious, but this can be a tricky mistake to catch because our function will still run— it just won't have the desired results.

We're going to explore this using the following function, which returns a promise that resolves to 'Yay, I resolved!' after a 1 second delay:

let myPromise = () => {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve('Yay, I resolved!')

}, 1000);

});

}

Now we'll write two async functions which invoke myPromise():

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!

In the first async function, noAwait(), we left off the await keyword before myPromise(). In the second, yesAwait(), we included it.

The noAwait() function logs Promise { <pending> } to the console. Without the await keyword, the **function execution wasn't paused.** The console.log() on the following **line was executed before the promise had resolved.**

Remember that the await operator returns the resolved value of a promise. When used properly in yesAwait(), the variable value was assigned the resolved value of the myPromise() promise, whereas in noAwait(), valuewas assigned the promise object itself.

**Example:**

**Library.js**

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(`2. I bought ${beanType} beans because they were on sale.`)

resolve(beanType);

}, 1000)

})}

module.exports = shopForBeans

**app.js**

const shopForBeans = require('./library.js');

async function getBeans() {

console.log(`1. Heading to the store to buy beans...`);

let value = await shopForBeans();

console.log(`3. Great! I'm making ${value} beans for dinner tonight!`);

}

getBeans();

**Handling Dependent Promises**

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

For example, we may make a network request based on a query to a database. In that case, we would need to wait to make the network request until we had the results from the database. With native promise syntax, we use a chain of .then() functions making sure to return correctly each one:

function nativePromiseVersion() {

returnsFirstPromise()

.then((firstValue) => {

console.log(firstValue);

return returnsSecondPromise(firstValue);

})

.then((secondValue) => {

console.log(secondValue);

});

}

Here's how we'd write an async function to accomplish the same thing:

async function asyncAwaitVersion() {

let firstValue = await returnsFirstPromise();

console.log(firstValue);

let secondValue = await returnsSecondPromise(firstValue);

console.log(secondValue);

}

* We mark our function as async.
* Inside our function, we create a variable firstValueassigned await returnsFirstPromise(). This means firstValue is assigned the resolved value of the awaited promise.
* Next, we log firstValue to the console.
* Then, we create a variable secondValue assigned to await returnsSecondPromise(firstValue). Therefore, secondValue is assigned this promise's resolved value.
* Finally, we log secondValue to the console.

Though using the async...await syntax less length of code, easy maintain and debug. It can also makes it easy to store and refer to resolved values from promises further back in our chain which is a much more difficult task with native promise syntax.

**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();

//console.log(type);

let isSoft = await soakTheBeans(type);

//console.log(isSoft);

let dinner = await cookTheBeans(isSoft);

console.log(dinner);

}

makeBeans();

# Handling Errors

When .catch() is used with a long promise chain, there is no indication of where in the chain the error was thrown. This can make debugging challenging.

With async...await, we use try...catch statements for error handling. This makes for easier debugging!

async function usingTryCatch() {

try {

let resolveValue = await asyncFunction('thing that will fail');

let secondValue = await secondAsyncFunction(resolveValue);

} catch (err) {

// Catches any errors in the try block

console.log(err);

}

}

usingTryCatch();

Remember, since async functions return promises we can still use native promise's .catch() with an asyncfunction

async function usingPromiseCatch() {

let resolveValue = await asyncFunction('thing that will fail');

}

let rejectedPromise = usingPromiseCatch();

rejectedPromise.catch((rejectValue) => {

console.log(rejectValue);

})

This is sometimes used in the global scope to catch final errors in complex code.

**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();

# Handling Independent Promises

Remember that await halts the execution of our asyncfunction. This allows us to conveniently write synchronous-style code to handle dependent promises. But what if our async function contains multiple promises which are not dependent on the results of one another to execute?

async function waiting() {

const firstValue = await firstAsyncThing();

const secondValue = await secondAsyncThing();

console.log(firstValue, secondValue);

}

async function concurrent() {

const firstPromise = firstAsyncThing();

const secondPromise = secondAsyncThing();

console.log(await firstPromise, await secondPromise);

}

In the waiting() function, we **pause our function until the first promise resolves**, **then we construct the second promise**. Once that resolves, we print both resolved values to the console.

In our concurrent() function, both promises are constructed without using await. We then await each of their resolutions to print them to the console.

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! Within our asyncfunctions we should still take **advantage of concurrency**, the ability to perform asynchronous actions at the same time.

Note: if we have **multiple truly independent promises** that we would like to execute fully in parallel, we must use individual .then() functions and avoid halting our execution with await.

**Example:**

**Library.js**

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(){

const vegetablePromise = steamBroccoli();

const starchPromise = cookRice();

const proteinPromise = bakeChicken();

const sidePromise = cookBeans();

console.log(`Dinner is served. We're having ${await vegetablePromise}, ${await starchPromise}, ${await proteinPromise}, and ${await sidePromise}.`)

}

serveDinner();

# Await Promise.all()

Another way to **take advantage of concurrency** when we have multiple promises which can be **executed simultaneously** is to await a 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. This promise's resolve value will be an array containing the resolved values of each promise from the argument array.

async function asyncPromAll() {

const resultArray = await Promise.all([asyncTask1(), asyncTask2(), asyncTask3(), asyncTask4()]);

for (let i = 0; i<resultArray.length; i++){

console.log(resultArray[i]);

}

}

Promise.all() allows us to take advantage of asynchronicity— each of the four asynchronous tasks can process concurrently. Promise.all() also has the benefit of **failing fast,** meaning it won't wait for the rest of the asynchronous actions to complete once any one has rejected**. As soon as the first promise in the array rejects, the promise returned from Promise.all() will reject with that reason.** As it was when working with native promises, Promise.all() is a good choice if multiple asynchronous tasks are all required, but none must wait for any other before executing.

**Example:**

**Library.js**

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 serveDinnerAgain(){

const foodArray = await Promise.all([steamBroccoli(), cookRice(), bakeChicken(), cookBeans()]);

let vegetable = foodArray[0];

let starch = foodArray[1];

let protein = foodArray[2];

let side = foodArray[3];

console.log(`Dinner is served. We're having ${vegetable}, ${starch}, ${protein}, and ${side}.`);

}

serveDinnerAgain();

# Introduction to Requests

Have you ever wondered what happens after you click a “**Submit**” button on a web page? where does the **information go**? How is the **information processed**? The answer to the previous questions **revolves around HTTP requests.**

There are many types of HTTP requests. The four most commonly used **types of HTTP** requests **are GET, POST, PUT, and DELETE.** We cover, GET and POST requests. If you want to learn more about the different HTTP requests .

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>

With a **GET** request, we’re **retrieving**, or getting, **information** from *some source* (usually a website). For a **POST** request, we’re **posting information** to a **source** that **will process** the **information** and **send it back.**

Now, we understand how to make GET and POST requests by using JavaScript’s **XHR** object. We’ll also incorporate query strings into our requests.

We’ll use the **Datamuse API for GET** requests and the **Rebrandly URL Shortener API for POST requests**. To complete the exercise on POST, make sure you create a Rebrandly API Key by click on this link.

# 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.**

JavaScript uses an **event loop** to handle **asynchronous function calls**. When a program is run, function calls are made and added to a stack. The **functions** that **make requests** that **need** to **wait** for **servers to respond** then **get sent to a separate queue.** Once the stack has cleared, then the functions in the queue are executed.

Web developers use the event loop to create a smoother browsing experience by deciding when to call functions and how to handle asynchronous events. We’ll be exploring one system of technologies called Asynchronous JavaScript and XML, or AJAX.

To read more about the event loop, read the MDN documentation:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop>

console.log('First message!');

setTimeout(() => {

console.log('This message will always run last...');

}, 0);

console.log('Second message!');

/\*

We’ll be using setTimeout(), which will pass a function

call to the queue. The first argument is a callback and

the second argument is the number of milliseconds the program

must wait before the callback can be run.

\*/

/\*

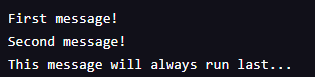
What if we change the 2500 in setTimeout() to be 0?

Essentially the callback doesn’t need to wait before it can

be called

\*/

**OUTPUT**



# XHR GET Requests I

Asynchronous JavaScript and XML (AJAX), **enables** **requests** to be **made** **after** the **initial page load**. Initially, AJAX was used only for XML formatted data, **now** it can be used to **make requests** that have **many different formats.**

[MDN Documentation: Extensible Markup Language (XML)](https://developer.mozilla.org/en-US/docs/XML_introduction).

Similarly, the XMLHttpRequest (XHR) API, named for XML, can be used to make many kinds of requests and supports other forms of data.

Remember**, we use GET to retrieve data from a source.**

what each step does.



# XHR GET Requests II

We are going to reconstruct XHR GET request boilerplate code step-by-step until we have written a complete GET request.

Main.js

//XMLHttpRequest GET

// CREATE NEW OBJECT

const xhr = new XMLHttpRequest();

const url = "https://api-to-call.com/endpoint";

// HANDLE RESPONSE

xhr.responseType= 'json';

xhr.onreadystatechange = () => {

if (xhr.readyState === XMLHttpRequest.DONE) {

return xhr.response;

}

}

xhr.open('GET', url);

xhr.send();

**XHR GET Requests III**

By this point, you have an idea of how to write the boilerplate code for an AJAX request using an XHR object.

In this exercise, you will incorporate that **boilerplate code to make a GET request to the Datamuse API to search for words that rhyme!**

* Datamuse API Documentation

**Example:**

**helperFunction.js**

// Formats response to look presentable on webpage

const renderResponse = (res) => {

// handles if res is falsey

if(!res){

console.log(res.status)

}

// in case res comes back as a blank array

if(!res.length){

responseField.innerHTML = "<p>Try again!</p><p>There were no suggestions found!</p>"

return

}

// creating an array to contain the HTML strings

let wordList = []

// looping through the response and maxxing out at 10

for(let i = 0; i < Math.min(res.length, 10); i++){

// creating a list of words

wordList.push(`<li>${res[i].word}</li>`)

}

// joins the array of HTML strings into one string

wordList = wordList.join("")

// manipulates responseField to render the modified response

responseField.innerHTML = `<p>You might be interested in:</p><ol>${wordList}</ol>`

return

}

// Renders response before it is modified

const renderRawResponse = (res) => {

// taking the first 10 words from res

let trimmedResponse = res.slice(0, 10)

//manipulates responseField to render the unformatted response

responseField.innerHTML = `<text>${JSON.stringify(trimmedResponse)}</text>`

}

// Renders the JSON that was returned when the Promise from fetch resolves.

const renderJsonResponse = (res) => {

// creating an empty object to store the JSON in key-value pairs

let rawJson = {}

for(let key in response){

rawJson[key] = response[key]

}

// converting JSON into a string and adding line breaks to make it easier to read

rawJson = JSON.stringify(rawJson).replace(/,/g, ", \n")

// manipulates responseField to show the returned JSON.

responseField.innerHTML = `<pre>${rawJson}</pre>`

}

**Main.js**

// 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}`;

//boilerplate code

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 GET Requests IV

Now, we will create a request to set a topic and find adjectives that describe the input word using query strings.

A query string contains additional information to be sent with a request. The Datamuse API allows us to retrieve more specific data with query strings attached to the request URL.

A **query string is separated from the URL using a ?character**. After ?, you can then create a parameter which is a key value pair joined by a =. Examine the example below:

'https://api.datamuse.com/words?key=value'

If you want to add an additional parameter you will have to use the & character to separate your parameters. Like so:

'https://api.datamuse.com/words?key=value&anotherKey=anotherValue'

**Example:main.js**

// Information to reach API

const url = 'https://api.datamuse.com/words?';

const queryParams = 'rel\_jjb=';

const additionalParams = '&topics=';

// Selecting page elements

const inputField = document.querySelector('#input');

const topicField = document.querySelector('#topic');

const submit = document.querySelector('#submit');

const responseField = document.querySelector('#responseField');

// AJAX function

const getSuggestions = () => {

const wordQuery = inputField.value;

const topicQuery = topicField.value

const endpoint = `${url}${queryParams}${wordQuery}${additionalParams}${topicQuery}`;

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 I

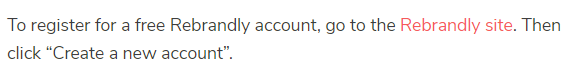
Reminder: If you haven’t already signed up for an API Key from Rebrandly, please read this Rebrandly sign up guide.

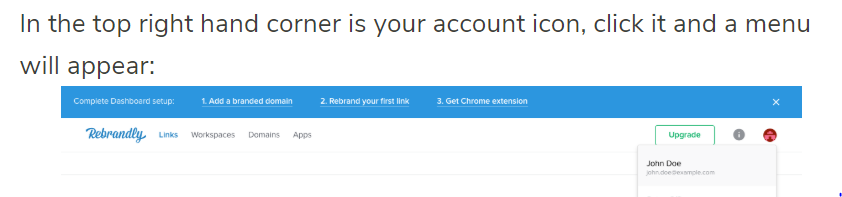
Refrences:

<https://developers.rebrandly.com/docs/get-started>

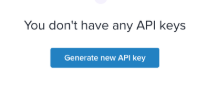
🡪 <https://app.rebrandly.com/>

🡪

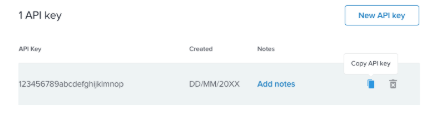
🡪



🡪



🡪



🡪 **f039175659ae4e61a110aa4c827c068c**

Great! By this point you’ve signed up for an API key, and you know the essence of making a GET request. We will be making a POST request using the Rebrandly API.

The major difference between a GET request and POST request is that a POST request requires additional information to be sent through the request. This additional information is sent in the body of the post request.

# XHR POST Requests II

We are going to reconstruct the code from the previous exercise step-by-step until we have written a complete AJAX POST request.



//XMLHttpRequest POST

// CREATE NEW OBJECT

const xhr = new XMLHttpRequest();

const url = "https://api-to-call.com/endpoint";

const data = JSON.stringify({id: '200'});

// HANDLE RESPONSE

xhr.responseType= 'json';

xhr.onreadystatechange = () => {

if (xhr.readyState === XMLHttpRequest.DONE) {

return xhr.response;

}

}

xhr.open('POST', url);

xhr.send(data);

**main.js**

// Information to reach API

const apiKey = 'f039175659ae4e61a110aa4c827c068c';

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();

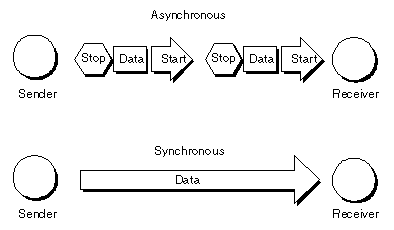
}

shortenButton.addEventListener('click', displayShortUrl);

**REQUESTS II**

# Introduction to Requests with ES6

In the previous lesson, we spent a lot of time dealing with asynchronous data (remember AJAX/ Asynchronous JavaScript And XML?). Many of our web page interactions rely on asynchronous events, so managing these events is essential to good web development.



To make asynchronous event handling easier, promises were introduced in JavaScript in ES6:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>

A **promise is an object that handles asynchronous data**. A promise has three states:

* pending : when a promise is created or waiting for data.
* fulfilled : the asynchronous operation was handled successfully.
* rejected : the asynchronous operation was unsuccessful.

The great thing about promises is that once a promise is fulfilled or rejected, you can chain an additional method to the original promise.

we will explain how to use fetch(), which **uses promises to handle requests**. Then, we will simplify requests using async and await.

We’ll use the Datamuse API for GET requests and Rebrandly URL Shortener API for POST requests. For you to complete the lessons on POST, make sure you create a Rebrandly API Key.

# fetch() GET Requests I

The first type of requests we’re going to tackle are GET requests using fetch()

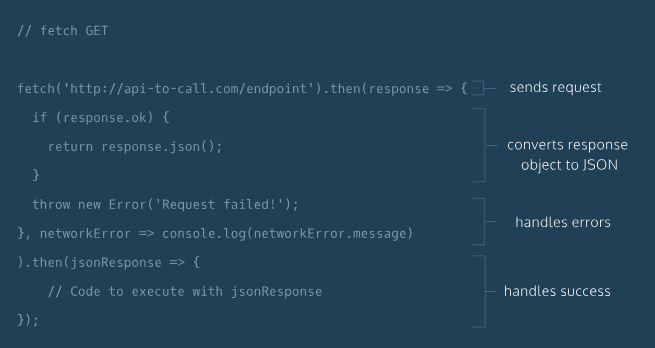
<https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API>

The fetch() function:

* **Creates** a **request object** that **contains** relevant **information that an API needs.**
* Sends that **request object** to the API **endpoint** provided.
* **Returns a promise that** ultimately resolves to a response object, which contains the status of the promise with information the API sent back.

# fetch() GET Requests II

We are going to reconstruct the boilerplate code necessary to create a GET request using the fetch() function step-by-step.



# fetch() GET Requests III

First , you wrote the boilerplate code for a GET request using  fetch() and .then().Now, you’re going to use that code and manipulate it to access the Datamuse API and render information in the browser.

<https://www.datamuse.com/api/>

If the request is successful, you’ll get back an array of words that sound like the word you typed into the input field.

You may get some errors as you complete each step. This is because sometimes we’ve split a single step into one or more steps to make it easier to follow. By the end, you should be receiving no errors.

# fetch() GET Requests IV

Great job making it this far!

In the previous exercise, you created the query URL, called the fetch() function and passed it the query URL and a settings object. Then, you chained a .then() method and passed it two functions as arguments — one to handle the promise if it resolves, and one to handle network errors if the promise is rejected.

Main.js

// 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).then(response => {

if (response.ok) {

return response.json();

}

throw new Error('Request failed!');

}, networkError => {

console.log(networkError.message)

}).then(jsonResponse => {

renderResponse(jsonResponse)

} );

}

// 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);

friendFunctions.js

// Formats response to look presentable on webpage

const renderResponse = (res) => {

// Handles if res is falsey

if(!res){

console.log(res.status);

}

// In case res comes back as a blank array

if(!res.length){

responseField.innerHTML = "<p>Try again!</p><p>There were no suggestions found!</p>";

return;

}

// Creates an empty array to contain the HTML strings

let wordList = [];

// Loops through the response and caps off at 10

for(let i = 0; i < Math.min(res.length, 10); i++){

// creating a list of words

wordList.push(`<li>${res[i].word}</li>`);

}

// Joins the array of HTML strings into one string

wordList = wordList.join("");

// Manipulates responseField to render the modified response

responseField.innerHTML = `<p>You might be interested in:</p><ol>${wordList}</ol>`;

return

}

// Renders response before it is modified

const renderRawResponse = (res) => {

// Takes the first 10 words from res

let trimmedResponse = res.slice(0, 10);

// Manipulates responseField to render the unformatted response

responseField.innerHTML = `<text>${JSON.stringify(trimmedResponse)}</text>`;

}

// Renders the JSON that was returned when the Promise from fetch resolves.

const renderJsonResponse = (res) => {

// Creates an empty object to store the JSON in key-value pairs

let rawJson = {};

for(let key in response){

rawJson[key] = response[key];

}

// Converts JSON into a string and adding line breaks to make it easier to read

rawJson = JSON.stringify(rawJson).replace(/,/g, ", \n");

// Manipulates responseField to show the returned JSON.

responseField.innerHTML = `<pre>${rawJson}</pre>`;

}

# fetch() POST Requests I

In the previous exercise, you successfully wrote a GET request using the fetch API and handled Promises to get word suggestions from Datamuse.

Now, you’re going to learn how to use fetch() to construct POST requests!

Take a look at the diagram.

Notice that the initial call takes two arguments: an endpoint and an object that contains information needed for the POST request. The rest of the request is identical to the GET request.



# fetch() POST Requests II

We are going to reconstruct the code from the previous exercise step-by-step until we have written a complete POST request using fetch()and .then().

//FETCH POST

fetch('https://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 => {

return jsonResponse;

})

# fetch() Post Requests III

In last, you created the boilerplate code for making a POST request using fetch() and .then(). Now, you’re going to update that boilerplate code to allow you to shorten a URL using the Rebrandly URL Shortener API.

<https://developers.rebrandly.com/>

created a Rebrandly API