operator

statements

object

method

variable

JavaScript

console keyword refers to an object, a collection of data and actions,

* Method:

.log() ; built into console

console.log() what we put inside the parentheses will **get printed, or logged, to the console.**

console.log(5); logs 5 to the console.

* *Method Comments:*
* *single line comment* will comment out a single line and is denoted with two forward slashes //preceding it.

// Prints 5 to the console

console.log(5);

* You can also use a single line comment to **comment after a line of code**:

console.log(5); // Prints 5

* **A *multi-line comment***will comment out multiple lines and is denoted with /\* to begin the comment, and \*/ to end the comment.

/\*

This is all commented

console.log(10);

None of this is going to run!

console.log(99);

\*/

You can also use this syntax to **comment something out in the middle of a line** of code:

console.log(/\*IGNORED!\*/ 5); // Still just prints 5

* 7 fundamental Data types 6 primitives and 1 complex (object):

1. ***Number*: Any number**, including numbers with decimals: 4, 8, 1516, 23.42.
2. ***String*: Any grouping of characters** on your keyboard (letters, numbers, spaces, symbols, etc.) surrounded by single quotes: ' ... ' or double quotes " ... ". preferred single quotes.
3. ***Boolean*:** This data type only has two possible values either true or false (without quotes).
4. ***Null*:** This data type represents the **intentional absence of a value**, and is represented by the keyword null (without quotes).
5. ***Undefined*:** This data type is denoted by the keyword undefined (without quotes). It also represents the **absence of a value** though it has a different use than null.
6. ***Symbol*:** A newer feature to the language, symbols are **unique identifiers**, useful in more complex coding.
7. ***Object*** *(only complex data type)*: **Collections of related data.**

* Method Arithmetic operators

*operator* is a character that performs a task in our code.

* 1. Add +

console.log(3 + 4); // Prints 7

* 1. Subtract -

console.log(5 - 1); // Prints 4

* 1. Multiply \*

console.log(4 \* 2); // Prints 8

* 1. Divide /

console.log(9 / 3); // Prints 3

* 1. Modulo/ Remainder %

console.log(11%3); //Prints 2

* Method String Concatenation

+ operator is used on two strings, it appends the right string to the left string:

console.log('hi' + 'ya'); // Prints 'hiya'

console.log('wo' + 'ah'); // Prints 'woah'

console.log('I love to ' + 'code.') // Prints 'I love to code.'

* Method counting string characters:

new piece of data into a JavaScript program, the browser saves it as an **instance of the data type.** Every **string instance** has a **property** called **length that stores the number of characters in that string**. You can **retrieve property information**

console.log('Hello'.length); // Prints 5

* “.” - the dot operator
* Calling methods:

We *call*, or use, these methods by appending an instance with a period (the dot operator), the name of the method, and opening and closing parentheses: ie.

'example string'.methodName()

* console.log() we're calling the .log() method on the console object.
* Method returns a string in all capital letters:

console.log('hello'.toUpperCase()); // Prints 'HELLO'

* Method:

console.log('Hey'.startsWith('H')); // Prints true

returns Boolean

This method also accepts the character 'H' as an input, or *argument*, between the parentheses.

* Method:

.trim().

Use the method to remove the whitespace at the beginning and end of the string

* Object: MATH

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Math

* Method for random numbers:

Call the .random() method from the built-in Math object:

console.log(Math.random()); // Prints a random number between 0 and 1

To generate a random number between 0 and 50, we could multiply this result by 50, like so:

Math.random() \* 50;

* Method for rounding down numbers

Math.floor() takes a decimal number, and rounds down to the nearest whole number.

Math.floor(Math.random() \* 50);

* Object: Standard JS objects, by category

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

* Methods for String.prototype

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

* Method: Math.ceil()

Method from MATH that returns the smallest integer greater than or equal to a number.

* Object: Number

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

* Method: Number.isInteger()

Determine whether the passed value is an integer.

* A *variable* is a container for a value. Little containers for information that live in a computer's memory; Variables label and store data in memory.

1. Create a variable with a descriptive name.
2. Store or update information stored in a variable.
3. Reference or “get” information stored in a variable.

var myName = 'Arya';

console.log(myName);

// Output: Arya

* var:  is a JavaScript *keyword* that creates, or *declares*, a new variable.
* myName is a vaiable’s name
* “=” – assignment operator
* “Arya” is the assigned value (“=”) to var
* *camel casing*. In camel casing you group words into one, the first word is lowercase, then every word that follows will have its first letter uppercased. (e.g. camelCaseEverything).
* General rules for naming variables:

1. Variable names cannot start with numbers.
2. Variable names are case sensitive
3. Variable names cannot be the same as *keywords*.

List of JS keywords:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#Keywords>

var documentation:

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

* let variable:

the variable can be **reassigned a** different value.  We can **declare a variable without assigning** the variable a value; the variable will be automatically **initialised with a value of undefined**

let price;

console.log(price);

// Output: undefined

price = 350;

console.log(price);

// Output: 350

let meal = 'Enchiladas';

console.log(meal);

// Output: Enchiladas

meal = 'Burrito';

console.log(meal);

// Output: Burrito

* const variable:

the variable **cannot be reassigned a different** value, it is a constant.  Try to reassign- TypeError

Need be assigned value from beginning. otherwise - SyntaxError

const myName = 'Gilberto';

console.log(myName);

// Output: Gilberto

If you will need to reassign value later on: use let, otherwise, use const

* **Mathematical Assignment Operators**

let w = 4;

w = w + 1;

console.log(w); // Output: 5

is the same as:

let w = 4;

w += 1;

console.log(w); // Output: 5

let x = 20;

x -= 5; // Can be written as x = x - 5

console.log(x); // Output: 15

let y = 50;

y \*= 2; // Can be written as y = y \* 2

console.log(y); // Output: 100

let z = 8;

z /= 2; // Can be written as z = z / 2

console.log(z); // Output: 4

* **Increment ++ and Decrement -- Operators**

increment increases value by 1 ++

decrement decreases value by 1 –

**variable's value** is **updated *and* assigned as** the new value of that variable.

let a = 10;

a++;

console.log(a); // Output: 11

let b = 20;

b--;

console.log(b); // Output: 19

* String concatenation with variables:

“+” operator can be used to **combine two string values**;

let myPet = 'armadillo';

console.log('I own a pet ' + myPet + '.'); // Output: 'I own a pet armadillo.'

* **String Interpolation**

we can insert, or *interpolate*, variables into strings using *template literals*. template literal is used to log strings together. `{}`

const myPet = 'armadillo';

console.log(`I own a pet ${myPet}.`); // Output: I own a pet armadillo.

template literal is wrapped by backticks ``

${myPet} is a placeholder.

let myName='Don';

let myCity='Burgas';

console.log(`My name is ${myName}. My favorite city is ${myCity}.`);

* typeof operator:

checks the value to its right and *returns*, or passes back, a string of the data type.

const unknown1 = 'foo';

console.log(typeof unknown1); // Output: string

const unknown2 = 10;

console.log(typeof unknown2); // Output: number

const unknown3 = true;

console.log(typeof unknown3); // Output: boolean

* The If keyword:

if (true) {

console.log('This message will print!');

}

// Prints "This message will print!"

The if statement is composed of:

* The if keyword followed by a set of parentheses () which is followed by a *code block*, or *block statement*, indicated by a set of curly braces {}.
* Inside the parentheses (), a condition is provided that evaluates to true or false.
* If the condition is true, the code inside the curly braces {} runs, or *executes*.
* If the condition is false, the block won't execute.

Else statement: if...else statements allow us to automate solutions to yes-or-no questions, also known as *binary decisions*.

if (false) {

console.log('The code in this block will not run.');

} else {

console.log('But the code in this block will!');

}

// Prints "But the code in this block will!"

* **Comparison operators**:

It can be helpful to think of comparison statements as questions.

We can also use comparison operators on different data types like strings:

'apples' === 'oranges' // false

* Less than: <
* Greater than: >
* Less than or equal to: <=
* Greater than or equal to: >=
* Is equal to: ===
* Is NOT equal to: !==

JavaScript, there are operators that work with boolean values known as *logical operators*.

* There are three **logical operators:**
* **the *and* operator (&&);**both conditions *must* evaluate to true for the entire condition to evaluate to true and execute
* **the *or* operator (||);** either condition being true; **Only one of the conditions must evaluate to true**for the overall statement to evaluate to true. If the first condition in an || statement evaluates to true, the **second condition won't even be checked.**
* **the *not* operator**, otherwise known as the ***bang* operator (!)**

let excited = true;

console.log(!excited); // Prints false

let sleepy = false;

console.log(!sleepy); // Prints true

if (stopLight === 'green' && pedestrians === 0) {

console.log('Go!');

} else {

console.log('Stop');

}

if (day === 'Saturday' || day === 'Sunday') {

console.log('Enjoy the weekend!');

} else {

console.log('Do some work.');

}

* Truthy and Falsy values

myVariable has a *truthy* value; when used in a **boolean or conditional context**, it evaluates **to true**because it has been assigned **a non-falsy value.**

let myVariable = 'I Exist!';

if (myVariable) {

console.log(myVariable);

} else {

console.log('The variable does not exist.') ;

}

* ***falsy*— or evaluate to false when checked as a condition**?
* **0**
* **Empty strings like "" or ''**
* **null**which represent when there is no value at all
* **undefined**which represent when a declared variable lacks a value
* **NaN, or Not a Number**

let numberOfApples = 0;

if (numberOfApples){

console.log('Let us eat apples!');

} else {

console.log('No apples left!');

}

// Prints 'No apples left!'

let defaultName;

if (username) {

defaultName = username;

} else {

defaultName = 'Stranger';

}

equivalent to :

let defaultName = username || 'Stranger'; ***short-circuit evaluation due to checking one side at a time*.**

* Ternary operator for if…else… alternative:

let isNightTime = true;

if (isNightTime) {

console.log('Turn on the lights!');

} else {

console.log('Turn off the lights!');

}

with ternary operators:

condition ? firstExpression : secondExpression

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

* The **condition,** isNightTime, is provided before the **?.**
* **Two expressions** follow the **?**and are separated by a **colon :.**
* If the condition evaluates **to true, the first expression** executes.
* If the condition evaluates **to false, the second expression executes**.
* Else… if statements:

let stopLight = 'yellow';

if (stopLight === 'red') {

console.log('Stop!');

} else if (stopLight === 'yellow') {

console.log('Slow down.');

} else if (stopLight === 'green') {

console.log('Go!');

} else {

console.log('Caution, unknown!');

}

When multiple possible outcomes. Starts evaluating from top statement and stops evaluating when first is true- then executes the following block.

* Switch keyword:

provides alternative to else… if in case of multiple possible outcomes.

The switch keyword initiates the statement and is followed by ( ... ), which contains the value that each case will compare. case keyword checks if the expression matches the specified value that comes after it. break keyword tells the computer to exit the block and not execute any more code or check any other cases inside the code block.  At the end of each switch statement, there is a default statement. If none of the cases are true, then the code in the default statement will run

let groceryItem = 'papaya';

switch (groceryItem) {

case 'tomato':

console.log('Tomatoes are $0.49');

break;

case 'lime':

console.log('Limes are $1.49');

break;

case 'papaya':

console.log('Papayas are $1.29');

break;

default:

console.log('Invalid item');

break; }

// Prints 'Papayas are $1.29'

* Functions:

reusable block of code that groups together a sequence of statements to perform a specific task.

Function declarations - binds a function to a name, or an *identifier*.

A function declaration consists of:

* The function keyword.
* The name of the function, or its identifier, followed by parentheses.
* A function body, or the block of statements required to perform a specific task, enclosed in the function’s curly brackets, { }.
* the parameters of the function

Parameters allow functions to accept input(s) and perform a task using these input(s). We use parameters as placeholders for information that will be passed to the function when it is called.

*hoisting* in JavaScript - access to function declarations before they're defined.

example of hoisting:

console.log(greetWorld());

// Output: Hello, World!

function greetWorld() {

console.log('Hello, World!');

}

// not a good practice

https://developer.mozilla.org/en-US/docs/Glossary/Hoisting

* Calling a function:

getReminder()

code inside a function body runs, or *executes*, only when the function is *called*. To call a function in your code, you type the function name followed by parentheses. *function call* executes the function body.

Parameters (treated like variables within a function) and arguments:

function calculateArea(width, height){

console.log(width\*height);

}

Arguments: When calling a function with parameters, we specify the values in the parentheses that follow the function name. The values passed to the function are called *arguments*. Arguments can be values or variables.

calculateArea(10,6);

identifier and arguments; order in which arguments are passed and assigned follows the order that the parameters are declared.

const rectWidth=1;

const rectHeight=2;

calculateArea(rectWidth, rectHeight);

* ES6 Default Parameters:

Default parameters allow **parameters to have a predetermined value** in case there is **no argument passed** into the function or if the **argument is undefined**when called.

function greeting (name = 'stranger') {

console.log(`Hello, ${name}!`)

}

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

greeting() // Output: Hello, stranger!

used the ‘=’ operator to assign the parameter name a default value of 'stranger'

function makeShoppingList(item1='milk', item2='bread', item3='eggs'){

console.log(`Remember to buy ${item1}`);

console.log(`Remember to buy ${item2}`);

console.log(`Remember to buy ${item3}`);

}

* Return- allows functions to produce an output:

When a function is called, the computer runs through the function's code and evaluates the result of calling the function. By default that resulting value is undefined.

To pass back information from the function call, we use a return statement. To create a return statement, we use the return keyword followed by the value that we wish to return. If the value is omitted, undefined is returned instead. Code after ‘return’ is not executed.

function calculateArea(width, height) {

const area= width\*height;

return area;

} // area is the returned value

function rectangleArea(width, height) {

if (width < 0 || height < 0) {

return 'You need positive integers to calculate area!';

}

return width \* height;

}

* Helper functions:

functions being called within another function are often referred to as *helper functions*.

function multiplyByNineFifths(number) {

return number \* (9/5);

};

function getFahrenheit(celsius) {

return multiplyByNineFifths(celsius) + 32;

};

getFahrenheit(15); // Returns 59

* Function Expressions:

define a function inside an expression, **use the function keyword**; in a function expression, the **function name is usually omitted**. A function with **no name is called an *anonymous function***. A **function expression is often stored in a variable** in order to refer to it.

const calculateArea = function (width, height) {

const area= width\*height;

return area;

};

To declare a function expression:

1. **Declare a variable** to make the **variable’s name** be the name, or **identifier, of your function.** Since the release of ES6, it is common practice to use **const**as the keyword to declare the variable.
2. **Assign as that variable's value** an **anonymous function** created by using the function keyword followed by a **set of parentheses with possible parameters**. Then a set of curly braces that contain **the function body.**

To **invoke a function expression**, write the **name of the variable** in which the function is stored **followed by parentheses enclosing any arguments** being **passed** into the function.

variableName(argument1, argument2)

Unlike function declarations, function expressions are **not hoisted so they cannot be called before they are defined.**

* **Arrow Functions**

ES6 introduced ***arrow function syntax***, using the special "**fat arrow" () =>**notation.

First include the parameters inside the ( ) and then add an arrow => that points to the function body surrounded in { } like this:

const rectangleArea = (width, height) => {

let area = width \* height;

return area;

}

* **Concise Body Arrow Functions**

Refactoring the arrow function syntax.  Concise body function:

* 0 parmeters:

const functionName = () => {

};

* 1 parameter:

const functionName = paramOne => {

};

* 2 or more parameters:

const functionName = (paramOne, paramTwo ) => {

};

Function body composed of a **single-line block does not need curly braces. Without the curly braces**, whatever that line evaluates will be **automatically returned; contents of the block** should **immediately follow the arrow =>**and the return keyword can be **removed; *implicit return*.**

* single-line block:

const sumNumbers = number => number + number;

* equivalent multi-line block:

const sumNumbers = number => {

const sum = number + number;

return sum;

};

example:

const squareNum = (num) => {

return num \* num;

};

is equivalent to :

const squareNum = num => num \* num;

* The parentheses around num - removed, single parameter.
* The curly braces { } - removed - a single-line block.
* The return keyword - removed – a single-line block.
* **Blocks and Scope**

block is the code found inside a set of curly braces {}.

Scope is the context in which our variables are declared.

* Global scope:

***global scope***, variables are declared outside of blocks. **global variables;**

* Block scope:

When a variable is **defined inside a block,** it is **only accessible to the code within the curly braces {}**. ; variable has *block scope;* **local variables**

* Scope pollution

**global variables go to the *global namespace***; accessible from **anywhere i**n the program. ***Scope pollution***is when we have **too many global variables** that exist in the **global namespace,** or when we **reuse variables across different scopes**.

let num = 50;

const logNum = () => {

num = 100; // Take note of this line of code

console.log(num);

};

logNum(); // Prints 100

console.log(num); // Prints 100

* Arrays:

New Year's Resolutions:

1. Keep a journal

2. Take a falconry class

3. Learn to juggle

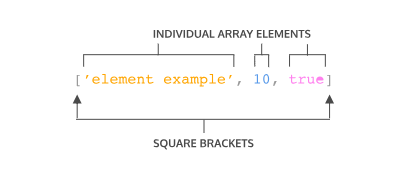
the equivalent in JS array form:

let newYearsResolutions = ['Keep a journal', 'Take a falconry class', 'Learn to juggle'];

arrays can **store any data types** (including **strings, numbers, and Booleans, same data types or an array that holds different data types)** . Like lists, **arrays are ordered**, meaning each **item has a numbered position**.

let concepts = ['creating arrays', 'array structures', 'array manipulation'];

* Creating array via array literal

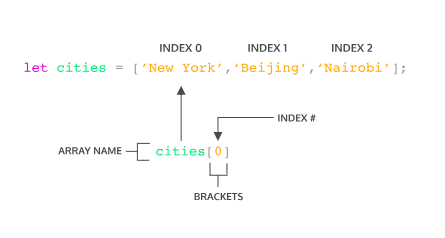


* The **array** is represented by the **square brackets [] and the content inside.**
* Each **content item** inside an array is called **an *element*.**
* There are **three different element**s inside the array.
* Each element inside the array is a **different data type.**
* Accessing elements from an array

**element** in an array has a **numbered position known as its *index*.**

**access individual** items using their **index,** which is similar to **referencing an item in a list based on the item's position.**

Arrays in JavaScript are ***zero-indexed***; positions **start counting from 0 rather** than 1.



In the code snippet above:

* cities is an array that has three elements.
* We're using **bracket notation, [] with the index after the name of the array to access the element.**
* **cities[0] will** access the element **at index 0 in the array cities**.

You can also **access individual characters in a string** **using bracket notation and the index**. For instance, you can write:

const **hello** = 'Hello World';

console.log**(hello[6]);** // Output: W

* Updating array elements:

let seasons = ['Winter', 'Spring', 'Summer', 'Fall'];

seasons[3] = 'Autumn';

console.log(seasons); //Output: ['Winter', 'Spring', 'Summer', 'Autumn']

* Arrays with let and const

Elements in an array declared with **const remain mutable**. Meaning that we **can change the contents of a const array**, but **cannot reassign a new array or a different value**.

let condiments = ['Ketchup', 'Mustard', 'Soy Sauce', 'Sriracha'];

condiments[0]='Mayo';

console.log(condiments);// logs [ 'Mayo', 'Mustard', 'Soy Sauce', 'Sriracha' ]

condiments=['Mayo'];

console.log(condiments);//logs [ 'Mayo' ]

const utensils = ['Fork', 'Knife', 'Chopsticks', 'Spork'];

utensils[3]='Spoon';

console.log(utensils);//logs [ 'Fork', 'Knife', 'Chopsticks', 'Spoon' ]

* The .length property:

array's built-in properties is length - returns the number of items in the array.

access the .length property just like we do with strings.

const newYearsResolutions = ['Keep a journal', 'Take a falconry class']; console.log(newYearsResolutions.length); // Output: 2

we log newYearsResolutions.length by:

We use *dot notation*, .length (.property) to the array,

if we want to know how many elements are in an array, use the .length property.

* the .push() Method:

methods are specifically called on arrays for common tasks

.push() to add elements to end of array:

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

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

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

* Then we **call it like a function. That's because .push() is a function** and one that JS allows us to use right on an array.
* .**push() uses single argument or multiple arguments** separated by **commas.**
* **.push() changes, or *mutates*,** the array. .push()referred to as a ***destructive* array** method since it changes the initial array.
* the .pop() Method:

**removes the last item of an array**.

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

const removed = newItemTracker.pop();

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

console.log(removed); // Output: item 2

* .pop() does not take **any arguments,** it simply **removes the last element** of newItemTracker. ; method **that mutates the initial array**.
* **.pop() returns the value of the last element**.
* More array methods:

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

* .shift() removes first item from array list:

groceryList.shift();

console.log(groceryList);

* .unshift() adds an item at beginning of array list

groceryList.unshift('popcorn');

console.log(groceryList);

* .slice()

**non-mutating method**

*arr*.slice(*[begin[*, *end]]*),

var animals = ['ant', 'bison', 'camel', 'duck', 'elephant'];

console.log(animals.slice(2));

// expected output: Array ["camel", "duck", "elephant"]

removes all before index 2, non-inclusive

**console.log(animals.slice(2, 4));**

**// expected output: Array ["camel", "duck"]**

**removes all before index 2,non inclusive ,removes all from index 4 onwards, inclusive**

console.log(animals.slice(1, 5));

// expected output: Array ["bison", "camel", "duck", "elephant"]

removes all before index 1, non inclusive, removes all from index 5 onwards, inclusive

* .indexOf()

finds index position of a list item (string):

const pastaIndex= groceryList.indexOf('pasta');

console.log(pastaIndex);// output is 4

* Functions/arrays/scope

const flowers = ['peony', 'daffodil', 'marigold'];

function addFlower(arr) {

arr.push('lily');

}

addFlower(flowers);

console.log(flowers); // Output: ['peony', 'daffodil', 'marigold', 'lily']

* **The array was mutated!**

So when you pass an array into a function, if the **array is mutated inside the function**, that **change will be maintained outside the function as well.** You might also see this concept explained as ***pass-by-reference***since what we're actually **passing the function is a reference** to **where the variable memory is stored and changing the memory.**

* Nested Arrays:

arrays can store other arrays.

const nestedArr = [[1], [2, 3]];

to access:

const nestedArr = [[1], [2, 3]];

console.log(nestedArr[1]); // Output: [2, 3]

if we wanted to **access the elements within the nested array** we can ***chain*,** or add on, **more bracket notation with index values**.

const nestedArr = [[1], [2, 3]];

console.log(nestedArr[1]); // Output: [2, 3]

console.log(nestedArr[1][0]); // Output: 2

* Loops:
* For loop:

***iterator variable* is initialized, checked against the stopping condition, and assigned a new value on each loop iteration**.; use a descriptive iterator variable name.

A for **loop contains three expressions separated by ;**inside the parentheses:

1. **an *initialization* starts the loop** and can also be used to **declare the iterator variable.**
2. **a *stopping condition***is the condition that the **iterator variable is evaluated agains**t— **if true the code block will run**, and if it evaluates **to false the code will stop.**
3. **an *iteration statement***is used to **update the iterator variable on each loop**.

Syntax:

for (let counter = 0; counter < 4; counter++) {

console.log(counter);

};// output is 0 1 2 3

* The initialization is let counter = 0, so the loop will start counting at 0.
* The stopping condition is counter < 4, meaning the loop will run as long as the iterator variable, counter, is less 4.
* The iteration statement is counter++. This means after each loop, the value of counter will increase by 1. For the first iteration counter will equal 0, for the second iteration counter will equal 1, and so on.
* The code block is inside of the curly braces, console.log(counter), will execute until the condition evaluates to false. The condition will be false when counter is greater than or equal to 4 — the point that the condition becomes false is sometimes called the *stop condition*.

for (let counter = 3; counter >= 0; counter--){

console.log(counter)

};

* Looping and arrays

To loop through each element in an array, a for loop should use the array's .length property in its condition.

const animals = ['Grizzly Bear', 'Sloth', 'Sea Lion'];

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

console.log(animals[i]);

};

Grizzly Bear

Sloth

Sea Lion

When we use i to iterate through arrays we can think of it as being short-hand for the word **i**ndex.

Notice how our **stopping condition checks that i is less than animals.length.**

const vacationSpots = ['Bali', 'Paris', 'Tulum'];

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

console.log('I would love to visit ' + vacationSpots[i]);

}

* Nested Loops:

often used to compare elements in two arrays

const myArray = [6, 19, 20];

const yourArray = [19, 81, 2];

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

for (let j = 0; j < yourArray.length; j++) {

if (myArray[i] === yourArray[j]) {

console.log('Both loops have the number: ' + yourArray[j]);

}

}

};

let bobsFollowers = ['Ade', 'Ben', 'Carlos', 'Dina'];

let tinasFollowers= ['Carlos', 'Dina','Eduardo'];

let mutualFollowers=[];

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

let bobsFriend=bobsFollowers[i];

for(j=0; j<tinasFollowers.length; j++){

let tinasFriend= tinasFollowers[j];

if (bobsFriend===tinasFriend){

mutualFollowers.push(bobsFriend);

}

}

}

console.log(mutualFollowers);

* While loop:

// A for loop that prints 1, 2, and 3

for (let counterOne = 1; counterOne < 4; counterOne++){

console.log(counterOne);

}

// A while loop that prints 1, 2, and 3

let counterTwo = 1;

while (counterTwo < 4) {

console.log(counterTwo);

counterTwo++;

}

while loop syntax:

* The counterTwo variable is **declared before the loop**; global scope.
* start loop by **keyword ‘while’** ,followed by our **stopping condition**, **or *test condition*;** evaluated before each round of the loop. While true, the block will continue to run. Once false - loop will stop.

syntax of a for loop is ideal when **we know how many times the loop should run, but we don't always know this in advance.**

const cards = ['diamond', 'spade', 'heart', 'club'];

let currentCard;

while( currentCard !=='spade'){

currentCard = cards[Math.floor(Math.random() \* 4)];

console.log(currentCard);

}

* Do…while statements

piece of code **to run at least once** and then **loop based on a specific condition after its initial** run; keep doing it **until a specified condition is no longer met.**

**let countString = '';**

**let i = 0;**

**do {**

**countString = countString + i;**

**i++;**

**} while (i < 5);**

**console.log(countString);**

First, the code **block after the do keyword is executed once.** Then the **condition is evaluated**. If the condition evaluates to true, the block will execute again. The **looping stops when the condition evaluates to false.**

Unlike the while loop, **do...while will run at least once whether or not the condition evaluates to true.**

const firstMessage = 'I will print!';

const secondMessage = 'I will not print!';

// A do while with a stopping condition that evaluates to false

do {

console.log(firstMessage)

} while (true === false);

// A while loop with a stopping condition that evaluates to false

while (true === false){

console.log(secondMessage) };

let cupsOfSugarNeeded= 3;

let cupsAdded= 0;

do {

cupsAdded++

} while (cupsAdded<cupsOfSugarNeeded);

console.log(cupsAdded);

* Break keyword:

we want to **stop a loop from continuing to execute** even though the o**riginal stopping condition** we wrote for our loop **hasn't been met,** we can use the **keyword break.**

Syntax:

for (let i = 0; i < 99; i++) {

if (i > 2 ) {

break;

}

console.log('Banana.');

}

console.log('Orange you glad I broke out the loop!');

Output for the code:

Banana.

Banana.

Banana.

Orange you glad I broke out the loop!

const rapperArray = ["Lil' Kim", "Jay-Z", "Notorious B.I.G.", "Tupac"];

for (let rapperArrayIndex=0;rapperArrayIndex<rapperArray.length;rapperArrayIndex++){

console.log(rapperArray[rapperArrayIndex]);

if(rapperArray[rapperArrayIndex]==='Notorious B.I.G.'){

break

}

}

console.log("And if you don't know, now you know.");

* Higher-order functions:

are functions that accept other functions as arguments and/or return functions as output.

* Functions v Data:

JS functions **behave like any other data type in the language**; assign functions to variables, and reassign them to new variables.

const announceThatIAmDoingImportantWork = () => {

console.log("I’m doing very important work!");

};

we can re-assign the function to a variable with a suitably short name:

const busy = announceThatIAmDoingImportantWork;

busy(); // This function call barely takes any space!

**busy is a variable that holds a *reference* to our original function**.

Our new busy() function can be **invoked with parentheses** as if that was the name we originally gave our function.

we **assign announceThatIAmDoingImportantWork without parentheses** as the value to the busy variable. We want to **assign the value of the function itself**, not **the value it returns when invoked.**

**In JS functions are *first class objects***, **have properties and methods; are a type of object**, [in the documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function" \t "_blank).

**Functions are special because we can invoke them**, but we can **still treat them like any other type of data**.

checkThatTwoPlusTwoEqualsFourAMillionTimes= () => {

for(let i = 1; i <= 1000000; i++) {

if ( (2 + 2) != 4) {

console.log('Something has gone very wrong :( ');

}

}

}

let is2p2=checkThatTwoPlusTwoEqualsFourAMillionTimes;

is2p2();

console.log(is2p2.name);

* **Functions as Parameters**

**we can also pass functions (into other functions) as parameters.**

**A *higher-order*** *function* is a function that either **accepts functions as parameters, returns a function, or both!**

We call the functions that **get passed in as parameters and invoked *callback functions***because they get **called during the execution of the higher-order function**.

When we **pass a function** in as **an argument to another function**, **we don't invoke it**. **Invoking** the function **would evaluate to the return value of that function call**. With **callbacks, we pass in the function itself** by typing the **function name *without* the parentheses** (that would evaluate to the result of calling the function):

const timeFuncRuntime = funcParameter => {

let t1 = Date.now();

funcParameter();

let t2 = Date.now();

return t2 - t1;

}

const addOneToOne = () => 1 + 1;

timeFuncRuntime(addOneToOne);

We wrote a higher-order function, timeFuncRuntime(). It takes in a function as an argument, saves a starting time, invokes the callback function, records the time after the function was called, and returns the time the function took to run by subtracting the starting time from the ending time.

We then invoked timeFuncRuntime() first with the addOneToOne() function - note how we passed in addOneToOne and did not invoke it.

timeFuncRuntime(() => {

for (let i = 10; i>0; i--){

console.log(i);

}

});

In this example, we invoked timeFuncRuntime() with an anonymous function that counts backwards from 10. **Anonymous functions can be arguments too!**

const checkThatTwoPlusTwoEqualsFourAMillionTimes = () => {

for(let i = 1; i <= 1000000; i++) {

if ( (2 + 2) != 4) {

console.log('Something has gone very wrong :( ');

}

}

};

const addTwo = num => num + 2;

const timeFuncRuntime = funcParameter => {

let t1 = Date.now();

funcParameter();

let t2 = Date.now();

return t2 - t1;

};

let time2p2=timeFuncRuntime(checkThatTwoPlusTwoEqualsFourAMillionTimes);

const checkConsistentOutput= (func,val)=> {

let firstTry=func(val);

let secondTry=func(val);

if(firstTry===secondTry){

return firstTry;

}else {

return 'This function returned inconsistent results';

}

};

console.log(checkConsistentOutput(addTwo,3));

* Iterators:

Imagine you had a grocery list and you wanted to know what each item on the list was. You'd have to scan through each row and check for the item. This common task is similar to what we have to do when we want to iterate over, or loop through, an array. One tool at our disposal is the for loop. However, we also have access to built-in array methods which make looping easier.

The built-in JavaScript array methods that help us iterate are called *iteration methods*, at times referred to as *iterators*. Iterators are methods called on arrays to manipulate elements and return values.

In this lesson, you will learn the syntax for these methods, their return values, how to use the documentation to understand them, and how to choose the right iterator method for a given task.

const artists = ['Picasso', 'Kahlo', 'Matisse', 'Utamaro'];

artists.forEach(artist => {

console.log(artist + ' is one of my favorite artists.');

});

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

const squareNumbers = numbers.map(number => {

return number \* number;

});

console.log(squareNumbers);

const things = ['desk', 'chair', 5, 'backpack', 3.14, 100];

const onlyNumbers = things.filter(thing => {

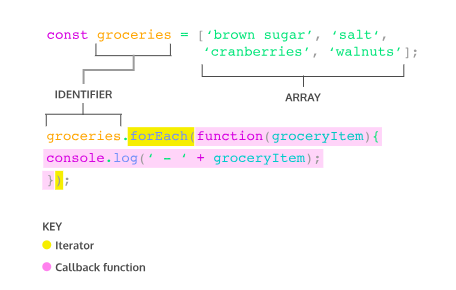
return typeof thing === 'number';

});

console.log(onlyNumbers);

* **The .forEach() Method**

iteration method**: .forEach()**will execute the **same code for each element of an array**.

****

* groceries.forEach() **calls the .forEach() method on the groceries array**.
* .forEach() **takes an argument of callback function.** Remember, a **callback function is a function passed as an argument into another function.**
* **.forEach()**loops through the array and **executes the callback function for each element**. During **each execution, the current element is passed as an argument to the callback function.**
* The **return value for .forEach() will always be undefined.**

Another way to **pass a callback for .forEach() is to use arrow function syntax**.

groceries.forEach(groceryItem =>

console.log(groceryItem));

We can **also define a function beforehand to be used as the callback function**.

function printGrocery(element){

console.log(element);

}

groceries.forEach(printGrocery);

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

fruits.forEach(item =>

console.log('I want to eat a '+item));

* **The .map() Method**

 .map() is called on an array, **it takes an argument of a callback function and returns a new array!**

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

const bigNumbers = numbers.map(number => {

return number \* 10;

});

.map() works in a similar manner to .forEach()— the major difference is **that .map() returns a new array.**

* numbers is an array of numbers.
* bigNumbers will store the **return value of calling .map() on numbers.**
* numbers.map will **iterate through each element in the**numbers array and **pass the element into the callback function.**
* return number \* 10 is **the code we wish to execute upon each element in the array**.

console.log(numbers); // Output: [1, 2, 3, 4, 5]

console.log(bigNumbers); // Output: [10, 20, 30, 40, 50]

const animals = ['Hen', 'elephant', 'llama', 'leopard', 'ostrich', 'Whale', 'octopus', 'rabbit', 'lion', 'dog'];

// Create the secretMessage array below

const secretMessage = animals.map(animal =>

animal[0]);

console.log(secretMessage.join(''));

const bigNumbers = [100, 200, 300, 400, 500];

// Create the smallNumbers array below

const smallNumbers = bigNumbers.map(num =>

num/100);

console.log(smallNumbers)

* **The .filter() Method**

Like .**map(), .filter() returns a new array. returns an array of elements after filtering out certain elements from the original array**.

The **callback function for the .filter()**method **should return true or false**depending on the **element that is passed to it**. **true are added to the new array**.

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

const shortWords = words.filter(word => {

return word.length < 6; });

* const shortWords = declares **a new variable that will store the returned array from** invoking .filter().
* The callback function is an arrow function has a **single parameter, word**.
* word.length < 6; is the **condition in the callback function**.

const randomNumbers = [375, 200, 3.14, 7, 13, 852];

// Call .filter() on randomNumbers below

const smallNumbers=randomNumbers.filter(number => {

return number<250 ;})

console.log(smallNumbers);

const favoriteWords = ['nostalgia', 'hyperbole', 'fervent', 'esoteric', 'serene'];

// Call .filter() on favoriteWords below

const longFavoriteWords=favoriteWords.filter(word => {

return word.length >7;

})

console.log(longFavoriteWords);

* **The .findIndex() Method**

**find the location of an element in an array**.

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

const jumbledNums = [123, 25, 78, 5, 9];

const lessThanTen = jumbledNums.findIndex(num => {

return num < 10; });

* jumbledNums is an array that contains elements that are numbers.
* const lessThanTen = declares a new variable that stores the returned index number from invoking .findIndex().
* The callback function is an arrow function has a single parameter, num. Each element in the jumbledNums array will be passed to this function as an argument.
* num < 10; is the condition that elements are checked against. .findIndex() will return the index of the first element which evaluates to true for that condition.

lessThanTen evaluates to:

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

**If we check what element has index of 3:**

console.log(jumbledNums[3]); // Output: 5

If there **isn't a single element in the array that satisfies the condition in the callback, then .findIndex() will return -1.**

const greaterThan1000 = jumbledNums.findIndex(num => {

return num > 1000; });

console.log(greaterThan1000); // Output: -1

const animals = ['hippo', 'tiger', 'lion', 'seal', 'cheetah', 'monkey', 'salamander', 'elephant'];

const foundAnimal= animals.findIndex(animal => {

return animal==='elephant';

})

const startsWithS=animals.findIndex(anim => {

return anim[0]==='s';

})

**The .reduce() Method**

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

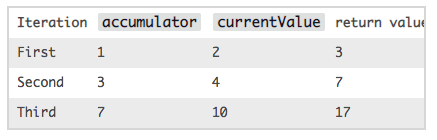
const numbers = [1, 2, 4, 10];

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

return accumulator + currentValue })

console.log(summedNums) // Output: 17

Here are the values of accumulator and currentValue as we iterate through the numbers array:



* numbers is an **array that contains numbers**.
* numbers.reduce() **calls the .reduce() method on the numbers array** and takes in a **callback function as argument.**
* The **callback function** **has two parameters**, **accumulator and currentValue**. The value of **accumulator starts off as the value of the first element in the array** and the **currentValue starts as the second element.**
* As .reduce() iterates through the array, the **return value of the callback function becomes the accumulator value for the next iteration**, **currentValue takes on the value of the current element** in the looping process.

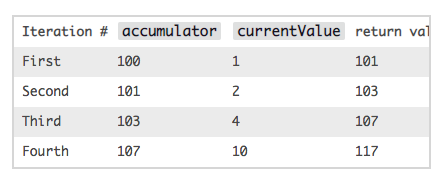
The ***.reduce() method can also take an optional second parameter*** to set an **initial value for accumulator** (remember, the **first argument is the callback function**!).

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

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

return accumulator + currentValue }, 100) // <- Second argument for .reduce()

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



const newNumbers = [1, 3, 5, 7];

const newSum=newNumbers.reduce((accumulator, currentValue) => {

console.log('The value of accumulator: ', accumulator);

console.log('The value of currentValue: ', currentValue);

return accumulator + currentValue;

}, 10)

console.log(newSum);

* **Iterator Documentation**

MDN Array iteration methods ;

[MDN's Array iteration methods page](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array" \l "Iteration_methods" \t "_blank).

1. A short definition.
2. A block with the correct syntax for using the method.
3. A list of parameters the method accepts or requires.
4. The return value of the function.
5. An extended description.
6. Examples of the method's use.
7. Other additional information.

* .some() method

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

some() executes the callback function once for each element present in the array until it finds one where callback returns a *truthy* value; Otherwise, some() returns false.

* .every() method

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

**every()** method tests whether all elements in the array pass the test implemented by the provided function. returns boolean

const words = ['unique', 'uncanny', 'pique', 'oxymoron', 'guise'];

// Something is missing in the method call below

console.log(words.some(word => {

return word.length < 6;

}));

const interestingWords=words.filter(word => {

return word.length>5;

});

console.log(interestingWords.every(word => {

return word.length >5 ;

}));

const cities = ['Orlando', 'Dubai', 'Edinburgh', 'Chennai', 'Accra', 'Denver', 'Eskisehir', 'Medellin', 'Yokohama'];

const nums = [1, 50, 75, 200, 350, 525, 1000];

cities.forEach(city => console.log('Have you visited ' + city + '?'));

// Choose a method that will return a new array

const longCities = cities.filter(city => city.length > 7);

// Choose a method that will return a single value

const word = cities.reduce((acc, currVal) => {

return acc + currVal[0]

}, "C");

console.log(word)

// Choose a method that will return a new array

const smallerNums = nums.map(num => num - 5);

// Choose a method that will return a boolean value

nums.some(num => num < 0);

* **Introduction to Objects**

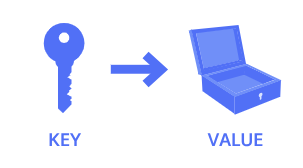
seven fundamental data types in JS, six of those are the primitive data types: **string, number, boolean, null, undefined, and symbol.** JavaScript objects are **containers storing related data and functionality,**

* **Creating Object Literals**

**Objects can be assigned to variables** just like any JS type. We use **curly braces, {}, to designate an *object literal*:**

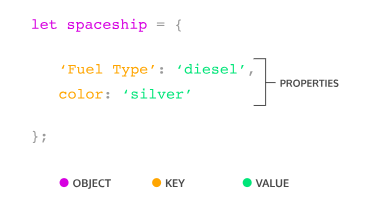
let spaceship = {}; // spaceship is an empty object

We **fill an object with unordered data**. This **data is organized into *key-value pairs***. A **key is like a variable name that points to a location in memory that holds a value.**



A **key's value** can be of ***any data type*** in the language including **functions or other objects.**

**key-value pair** by writing the **key's name, or *identifier***, followed by a **colon and then the value.** We **separate each key-value pair** in an **object literal with a comma (,).** **Keys are strings**, but when we have a **key that does not have any special characters in it**, JavaScript allows us to **omit the quotation marks**:



// An object literal with two key-value pairs

let spaceship = {

'Fuel Type': 'diesel',

color: 'silver'

};

The **spaceship object *has two properties*Fuel Type and color**. **'Fuel Type' has quotation marks** because it **contains a space character.**

let fasterShip= {

color: 'silver',

'Fuel Type': 'Turbo Fuel'

};

* Accessing properties:

**two ways to access an object's property**.

1. dot notation, ..

used dot notation to access the properties and methods of built-in objects and data instances:

'hello'.length; // Returns 5

**With property dot notation, we write the object's name, followed by the dot operator and then the property name (key):**

let spaceship = {

homePlanet: 'Earth',

color: 'silver'

};

spaceship.homePlanet; // Returns 'Earth'

spaceship.color; // Returns 'silver',

let spaceship = {

homePlanet: 'Earth',

color: 'silver',

'Fuel Type': 'Turbo Fuel',

numCrew: 5,

flightPath: ['Venus', 'Mars', 'Saturn']

};

const crewCount=spaceship.numCrew;

const planetArray=spaceship.flightPath;

If we try to access a **property that does not exist on that object**, **undefined** will be returned.

spaceship.favoriteIcecream; // Returns undefined

* **Bracket Notation**

To use bracket notation to access an object's property, **we pass in the property name (key) as a string.**

We ***must* use bracket notation when accessing keys that have numbers, spaces, or special characters in them**.

let spaceship = { '

Fuel Type': 'Turbo Fuel', '

Active Duty': true,

homePlanet: 'Earth',

numCrew: 5

};

spaceship['Active Duty']; // Returns true

spaceship['Fuel Type']; // Returns 'Turbo Fuel'

spaceship['numCrew']; // Returns 5

spaceship['!!!!!!!!!!!!!!!']; // Returns undefined

With **bracket notation** you can also **use a variable inside the brackets** to **select the keys of an object. T**his can be especially helpful when working with functions:

let returnAnyProp = (objectName, propName) => objectName[propName]; returnAnyProp(spaceship, 'homePlanet'); // Returns 'Earth'

If we tried to write our returnAnyProp() function with dot notation (objectName.propName) the computer would look for a key of 'propName' on our object and not the value of the propName parameter.

let spaceship = {

'Fuel Type' : 'Turbo Fuel',

'Active Mission' : true,

homePlanet : 'Earth',

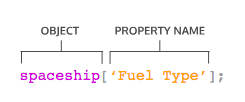
numCrew: 5

};

let propName = 'Active Mission';

let isActive=spaceship['Active Mission'];

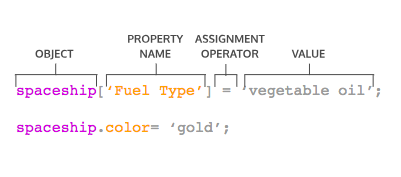
console.log(isActive);



* **Property Assignment**

Objects **are *mutable* meaning** we can update them after created

use either **dot notation, ., or bracket notation, [],** and the **assignment operator, =**to add new key-value pairs to an object or change an existing property.



* If the **property already exists on the object**, whatever **value it held before will be replaced** with the newly **assigned value**.
* If **there was no property with that name**, a **new property will be added** to the object.

we **can't reassign an object declared with const**, we **can still mutate it**, meaning we can **add new properties and change the properties** that are there.

const spaceship = {

type: 'shuttle'

};

**spaceship = {type:**

**'alien'**

**};// TypeError: Assignment to constant variable.**

spaceship.type = 'alien'; // Changes the value of the type property

spaceship.speed = 'Mach 5'; // Creates a new key of 'speed' with a value of 'Mach 5'

You can **delete a property from an object with the delete operator.**

const spaceship = {

'Fuel Type': 'Turbo Fuel',

homePlanet: 'Earth',

mission: 'Explore the universe'

};

**delete spaceship.mission; // Removes the mission property**

let spaceship = {

'Fuel Type' : 'Turbo Fuel',

homePlanet : 'Earth',

color: 'silver',

'Secret Mission' : 'Discover life outside of Earth.'

};

spaceship.color= 'glorious gold';

spaceship.numEngines=7;

delete spaceship['Secret Mission'];

* **Methods**

When the **data stored on an object is a function we call that a *method***. A **property** is what an **object has**, while a **method** is what an **object does.**

*console*is a **global javascript object** and **.log() is a method** on that object.

**Math** is also a **global javascript object** **and .floor()**is a method on it.

We can include methods in our object literals by **creating ordinary, comma-separated key-value pairs. T**he **key serves as our method's name**, while **the value is an anonymous function expression.**

const alienShip = {

**invade: function () {**

**console.log**('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.')

}

};

With the new method syntax introduced in ES6 **we can** **omit the colon and the function keyword**.

const alienShip = {

**invade () {**

**console**.log('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.')

}

};

Object methods are invoked by **appending the object's name** with **the dot operator followed** by the **method name and parentheses**:

**alienShip.invade(); //** Prints 'Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.'

let objectName = {

methodName() {

console.log('The methodName method was invoked!')

},

secondMethodName() {

console.log('The secondMethodName method was invoked!')

}

};

let retreatMessage = 'We no longer wish to conquer your planet. It is full of dogs, which we do not care for.';

let alienShip= {

retreat () {

console.log(retreatMessage);

},

takeOff () {

console.log('Spim... Borp... Glix... Blastoff!');

}

};

alienShip.retreat();

alienShip.takeOff();

* **Nested Objects**

**objects are often nested**— an **object might have another object as a property** which in turn **could have a property that's an array of even more objects**!

In our spaceship object, we want **a crew object**. This will contain all the **crew members** who do important work on the craft. Each of those **crewmembers are objects themselves**. They have **properties like name, and degree,** and they each have unique methods based on their roles. We can also nest other objects in the spaceship such as a telescope o**r nest details about the spaceship's computers inside a parent nanoelectronics object.**

const spaceship = {

telescope: {

yearBuilt: 2018,

model: '91031-XLT',

focalLength: 2032

},

crew: {

captain: {

name: 'Sandra',

degree: 'Computer Engineering',

encourageTeam() {

console.log('We got this!')

}

}

},

engine: {

model: 'Nimbus2000'

},

nanoelectronics: {

computer: {

terabytes: 100,

monitors: 'HD'

},

'back-up': {

battery: 'Lithium',

terabytes: 50

}

}

};

We can **chain operators to access nested properties**.

spaceship.nanoelectronics['back-up'].battery; // Returns 'Lithium'

In the preceding code:

* **First the computer evaluates spaceship.nanoelectronics**, which results in an object containing the back-up and computerobjects.
* We **accessed the back-up object by appending ['back-up'].**
* The back-up object has a **battery propert**y, accessed with .battery which returned the value stored there: 'Lithium'

let spaceship = {

passengers: [{name: 'Carlos'}],

telescope: {

yearBuilt: 2018,

model: "91031-XLT",

focalLength: 2032

},

crew: {

captain: {

name: 'Sandra',

degree: 'Computer Engineering',

encourageTeam() { console.log('We got this!') },

'favorite foods': ['cookies', 'cakes', 'candy', 'spinach']

}

},

engine: {

model: "Nimbus2000"

},

nanoelectronics: {

computer: {

terabytes: 100,

monitors: "HD"

},

backup: {

battery: "Lithium",

terabytes: 50

}

}

};

let capFave= spaceship.crew.captain['favorite foods'][0];

let firstPassenger=spaceship.passengers[0];

* **Pass By Reference**

**Objects are *passed by reference***. This means **when we pass a variable assigned to an object into a function as an argument**, the computer interprets the **parameter name** as pointing to the **space in memory holding that object**.

As a result, **functions which change object properties actually mutate the object permanently** (even when the object is assigned to a const variable).

const spaceship = {

homePlanet : 'Earth',

color : 'silver'

};

let paintIt = obj => {

obj.color = 'glorious gold'

};

paintIt(spaceship); spaceship.color // Returns 'glorious gold'

Our function paintIt() **permanently changed the color of our spaceship object**. However, reassignment of the spaceship variable wouldn't work in the same way:

let spaceship = {

homePlanet : 'Earth',

color : 'red'

};

let tryReassignment = obj => {

obj = {

identified : false,

'transport type' : 'flying'

}

console.log(obj) // Prints {'identified': false, 'transport type': 'flying'}

};

tryReassignment(spaceship) // The attempt at reassignment does not work. spaceship // Still returns {homePlanet : 'Earth', color : 'red'};

spaceship = {

identified : false,

'transport type': 'flying'

}; // Regular reassignment still works.

Let's look at what happened in the code example:

* We declared this spaceship object with let. This allowed us to **reassign it to a new object with identified and 'transport type' properties** with no problems.
* When we tried the same thing **using a function designed to reassign the object passed into it,** the reassignment didn't stick (even though calling console.log() on the object produced the expected result).
* When we **passed spaceship into that function**, **obj became a reference to the memory location of the spaceship object**, but ***not* to the spaceship variable**. This is because the **obj parameter** of **the tryReassignment() function is a variable in its own** right. The body of tryReassignment() has no knowledge of the spaceship variable at all!
* When we did **the reassignment in the body of tryReassignment**(), the **obj variable** came to **refer to the memory location of the object**{'identified' : false, 'transport type' : 'flying'}, while the **spaceship variable was completely unchanged** from its earlier value.

let spaceship = {

'Fuel Type' : 'Turbo Fuel',

homePlanet : 'Earth'

};

let greenEnergy = obj => {

obj['Fuel Type'] = 'avocado oil';

}

let remotelyDisable = obj => {

obj.disabled = true;

}

greenEnergy(spaceship);

remotelyDisable(spaceship);

console.log(spaceship)

* **Looping Through Objects**

We learned how to **iterate through arrays** using their **numerical indexing**, but the **key-value pairs in objects aren't ordered**! [JavaScript has given us alternative solution for iterating through objects with the for...in syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in" \t "_blank).

for...in will execute a given block of code for each property in an object.

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in>

let spaceship = {

crew: {

captain: {

name: 'Lily',

degree: 'Computer Engineering',

cheerTeam() {

console.log('You got this!')

}

},

'chief officer': {

name: 'Dan',

degree: 'Aerospace Engineering',

agree() {

console.log('I agree, captain!')

}

},

medic: {

name: 'Clementine',

degree: 'Physics',

announce() {

console.log(`Jets on!`)

}

},

translator: {

name: 'Shauna',

degree: 'Conservation Science',

powerFuel() {

console.log('The tank is full!')

}

}

}

};

// for...in

for (let **crewMember** in spaceship.crew) {

console.log( `${**crewMember**}: ${spaceship.crew[**crewMember**].name}` )

};

Our for...in will iterate through each element of the spaceship.crew object. In each iteration, the **variable crewMember is set to one of spaceship.crew's keys,** enabling us to log a list of crew members' role and name.

let spaceship = {

crew: {

captain: {

name: 'Lily',

degree: 'Computer Engineering',

cheerTeam() { console.log('You got this!') }

},

'chief officer': {

name: 'Dan',

degree: 'Aerospace Engineering',

agree() { console.log('I agree, captain!') }

},

medic: {

name: 'Clementine',

degree: 'Physics',

announce() { console.log(`Jets on!`) } },

translator: {

name: 'Shauna',

degree: 'Conservation Science',

powerFuel() { console.log('The tank is full!') }

}

}

};

// Write your code below

for (let crewMember in spaceship.crew) {

console.log(`${crewMember}: ${spaceship.crew[crewMember].name}`)

};

for (let crewMember in spaceship.crew) {

console.log(`${spaceship.crew[crewMember].name}: ${spaceship.crew[crewMember].degree}`)

};

* **Advanced Objects Introduction**

const robot = {

model: 'B-4MI',

mobile: true,

greetMaster() {

console.log(`I'm model ${this.model}, how may I be of service?`);

}

}

const massProdRobot = (model, mobile) => {

return {

model,

mobile,

greetMaster() {

console.log(`I'm model ${this.model}, how may I be of service?`);

}

}

}

const shinyNewRobot = massProdRobot('TrayHax', true)

const chargingStation = {

\_name: 'Electrons-R-Us',

\_robotCapacity: 120,

\_active: true,

\_chargingRooms: ['Low N Slow', 'Middle of the Road', 'In and Output'],

set robotCapacity(newCapacity) {

if (typeof newCapacity === 'number') {

this.\_robotCapacity = newCapacity;

} else {

console.log(`Change ${newCapacity} to a number.`)

}

},

get robotCapacity() {

return this.\_robotCapacity;

}

}

* **The this Keyword**

**Objects** are **collections of related data and functionality**. We store that functionality in methods on our objects:

const goat = {

dietType: 'herbivore',

makeSound() {

console.log('baaa');

}

};

In our goat object we have a .makeSound() method. We can invoke the .makeSound() method on goat.

goat.makeSound(); // Prints baaa

Nice, we have a goat object that can print baaa to the console. Everything seems to be working fine. What if we wanted to add a new method to our goat object called .diet() that prints the goat's dietType?

const goat = {

dietType: 'herbivore',

makeSound() {

console.log('baaa');

},

diet() {

console.log(dietType);

}

};

goat.diet(); // Output will be "ReferenceError: dietType is not defined"

inside the scope of the .diet() method, we **don't automatically have access to other properties of the goat object.**

If we change the .diet() method to use the this, the .diet() works! :

const goat = {

dietType: 'herbivore',

makeSound() {

console.log('baaa');

},

diet() {

console.log(this.dietType);

}

};

goat.diet(); // Output: herbivore

The **this keyword references the *calling object***which **provides access to the calling object's** **properties.** In the example above, the **calling object is goat**and by **using this we're accessing the goat object itself**, and then **the dietType pro**perty of goat by using property **dot notation.**

to get the access to the calling object's properties inside a method, you have to use the this keyword!

const robot = {

model: '1E78V2',

energyLevel: 100,

provideInfo(){

return `I am ${this.model} and my current energy level is ${this.energyLevel}`

}

};

console.log(robot.provideInfo());

* **Arrow Functions and this**

If we use the **this keyword** in a method then the value of this is the calling object. However, it becomes a bit more complicated when we start using arrow functions for methods. Take a look at the example below:

const goat = {

dietType: 'herbivore',

makeSound() {

console.log('baaa');

},

diet: () => {

console.log(this.dietType);

}

};

goat.diet(); // Prints undefined

**goat.diet()would log undefined.** So what happened? Notice that in the **.diet() is defined using an arrow function**.

**Arrow functions inherently *bind*,** or tie, an **already defined this value** to the **function itself** that is **NOT the calling object.**

In the code snippet above, the **value of this is the *global object***, which **doesn't have a dietType property** and therefore returns undefined.

To read more about either arrow functions or the global object check out the MDN documentation of [the global object](https://developer.mozilla.org/en-US/docs/Glossary/Global_object" \t "_blank) and [arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions" \t "_blank).

The key takeaway from the example above is to ***avoid* using arrow functions when using this in a method!**

* **Privacy**

However, there **are cases** in which **we don't want other code simply accessing and updating an object's properties.**

***privacy* in objects**, we define it as the idea that **only certain properties should be mutable**

Rather, JavaScript developers follow **naming conventions** that **signal to other developers how to interact with a property.** One common convention is to place **an underscore \_ before the** **name of a proper**ty to mean that the **property should not be altered**. an example of using \_ to prepend a property.

const bankAccount = {

\_amount: 1000

}

above, the **\_amount is not intended to be directly manipulated.**

still possible to **reassign \_amount:**

bankAccount.\_amount = 1000000;

* **Getters**

***Getters***are **methods that get and return the internal properties of an object.**

const person = {

\_firstName: 'John',

\_lastName: 'Doe',

get fullName() {

if (this.\_firstName && this.\_lastName){

return `${this.\_firstName} ${this.\_lastName}`;

} else {

return 'Missing a first name or a last name.';

}

}

}

// To call the getter method:

person.fullName;

// 'John Doe'

* We use the **get keyword followed by a function**.
* We use an ***if...else conditional to check if both \_firstName and \_lastName exist*** (by making sure they both return truthy values) and then **return a different value depending on the result.**
* We can **access the calling object's internal properties** using this. In **fullName, we're accessing both this.\_firstName and this.\_lastName.**
* In the last line we **call fullName on person**. In general, ***getter methods do not need to be called with a set of parentheses.*** Syntactically, it **looks like we're accessing a property.**
* Getters **can perform an action on the data when getting a property.**
* Getters can **return different values using conditionals.**
* In a **getter, we can access the properties of the calling object using this.**

**properties cannot share the same name as the getter/setter function.** If we do so, then calling the method will result in an infinite call stack error. One workaround is to **add an underscore before the property name**

const robot = {

\_model: '1E78V2',

\_energyLevel: 100,

get energyLevel(){

if( typeof this.\_energylevel === 'number'){

return 'My current energy level is ' + this.\_energylevel;

} else {

return 'System malfunction: cannot retrieve energy level'.

}

}

};

console.log(robot.energyLevel);

* **Setters**

**setter methods** which **reassign values of existing properties within an object.**

const person = {

\_age: 37,

set age(newAge){

if (typeof newAge === 'number'){

this.\_age = newAge;

} else {

console.log('You must assign a number to age');

}

}

};

* We can perform a **check for what value is being assigned to this.\_age.**
* When we use the setter method, **only values that are numbers will reassign this.\_age**
* There are **different outputs depending on what values are used to reassign this.\_age**.

person.age = 40;

console.log(person.\_age); // Logs: 40

person.age = '40'; // Logs: You must assign a number to age

**Setter methods** like age **do not need to be called with a set of parentheses.** Syntactically, it **looks like we're reassigning the value of a property.**

advantages to using setter methods **that include checking input, performing actions on properties, and displaying a clear intention for how the object is supposed to be used.** Nonetheless, **even with a setter method**, it is still possible to **directly reassign properties**.

we can still set .\_age directly:

person.\_age = 'forty-five'

console.log(person.\_age); // Prints forty-five

const robot = {

\_numOfSensors: 15;

set numOfSensors(num) {

if (typeof num === 'number' && num >= 0){

this.\_numOfSensors = num;

} else {

console.log('Pass in a number that is greater than or equal to 0');

}

}

};

const robot = {

\_model: '1E78V2',

\_energyLevel: 100,

\_numOfSensors: 15,

get numOfSensors(){

if(typeof this.\_numOfSensors === 'number'){

return this.\_numOfSensors;

} else {

return 'Sensors are currently down.'

}

},

set numOfSensors(num){

if(typeof num === 'number' && num >= 0){

this.\_numOfSensors=num;

} else {

console.log('Pass in a number that is greater than or equal to 0');

}

}

};

console.log(robot.energyLevel);

robot.numOfSensors = 100;

console.log(robot.numOfSensors);

* **Factory Functions**

**create many instances of an object quickly :***factory functions* .

A factory function is a **function that returns an objec**t and can be **reused to make multiple object instances**. Factory functions **can also have parameters** allowing us to **customize the object** returned.

**create an object** to represent monsters in JS. There are many different types ; creating multiple monsters objects, use a factory function that has parameters:

const monsterFactory = (name, age, energySource, catchPhrase) => {

return {

name: name,

age: age,

energySource: energySource,

scare() {

console.log(catchPhrase);

}

}

};

**four parameters** and **returns an object** that has the **properties: name, age, energySource, and scare().**

**call monsterFactory**with the **necessary arguments** and **assign the return value to a variable**:

const ghost = monsterFactory('Ghouly', 251, 'ectoplasm', 'BOO!');

ghost.scare(); // 'BOO!'

const robotFactory = (model,mobile) => {

return{

model:model,

mobile:mobile,

beep(){

console.log('Beep Boop');

}

}

};

const tinCan = robotFactory('P-500', true);

tinCan.beep();// logs ‘Beep Boop’

* **Property Value Shorthand**

ES6 shortcuts for assigning properties to variables known as ***destructuring*.**

We had to **assign each property a key** and **value** even though the key name was the same as the parameter name we assigned to it. A **truncated version of the factory function:**

**const monsterFactory = (name, age) => { return { name: name, age: age } };**

But we can **use a destructuring technique**, called ***property value shorthand***,

The example below works exactly like the example above:

**const monsterFactory = (name, age) => { return { name, age } };**

function robotFactory(model, mobile){

return {

model,

mobile,

beep() {

console.log('Beep Boop');

}

}

}

// To check that the property value shorthand technique worked:

const newRobot = robotFactory('P-501', false)

console.log(newRobot.model)

console.log(newRobot.mobile)

* **Destructured Assignment**

To **extract key-value pairs from objects** and **save them as properties**.

const vampire = {

name: 'Dracula',

residence: 'Transylvania',

preferences: { day: 'stay inside', night: 'satisfy appetite' }

};

If we wanted to **extract the residence property** **as a variable**:

const residence = vampire.residence;

console.log(residence); // Prints 'Transylvania'

**destructuring technique called *destructured assignment***:

In destructured assignment we **create a variable with the name of an object's key** that is wrapped in **curly braces { } and assign to it the object.** **extracting the property by :**

**const { residence } = vampire;**

console.log(residence); // Prints 'Transylvania'

declare a **new variable**residence that **extracts the value of the residence property**.

**use destructured assignment to grab nested properties of an object**:

const { day } = vampire.preferences; // entering the object properties

console.log(day); // Prints 'stay inside'

const robot = {

model: '1E78V2',

energyLevel: 100,

functionality: {

beep() {

console.log('Beep Boop');

},

fireLaser() {

console.log('Pew Pew');

},

}

};

**const {functionality} =robot;**

functionality.beep();

* **Built-in Object Methods**

**built-in methods for Objects**

we have **access to object instance methods like**: .hasOwnProperty(), .valueOf(), and many more! Practice your documentation reading skills and check out: [MDN's object instance documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object" \l "Methods" \t "_blank).

There are also useful **Object class methods such as Object.assign(), Object.entries(), and Object.keys()**

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

* **Object.keys()** method **returns an array of a given object's own property names**, in the **same order as we get with a normal loop.**

const object1 = {

a: 'somestring',

b: 42,

c: false

};

console.log(Object.keys(object1));

// expected output: Array ["a", "b", "c"]

// array like object with random key ordering

var anObj = { 100: 'a', 2: 'b', 7: 'c' };

console.log(Object.keys(anObj)); // console: ['2', '7', '100']

* **Object.entries()** method **returns an array of a given object's own enumerable** **property [key, value] pairs,** in the same order **as that provided by a**[**for...in**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loop (the difference being that a for-in loop enumerates properties in the prototype chain as well) . The **order of the array returned by Object.entries() does not depend on how an object is defined**. If there is a need for certain ordering then the array needs to be sorted first like

Object.entries(obj).sort((a, b) => a[0] - b[0]);.

const object2 = { 0: 'a', 1: 'b', 2: 'c' };

console.log(Object.entries(object2)[2]);

// expected output: Array ["2", "c"]

const result = Object.entries(object2).sort((a, b) => a - b);

console.log(Object.entries(result)[1]);

// expected output: Array ["1", Array ["1", "b"]]

// array like object const obj = { 0: 'a', 1: 'b', 2: 'c' };

console.log(Object.entries(obj)); // [ ['0', 'a'], ['1', 'b'], ['2', 'c'] ]

// non-object argument will be coerced to an object console.log(Object.entries('foo')); // [ ['0', 'f'], ['1', 'o'], ['2', 'o'] ]

// iterate through key-value gracefully

const obj = { a: 5, b: 7, c: 9 };

for (const [key, value] of Object.entries(obj)) {

console.log(`${key} ${value}`); // "a 5", "b 7", "c 9" }

* The **Object.assign()** method is used to **copy the values of all enumerable own properties** from one or more **source objects** to a **target object**. It will **return the target object.**

var obj = { a: 1 };

var copy = Object.assign({}, obj);

console.log(copy); // { a: 1 }

const robot = {

model: 'SAL-1000',

mobile: true,

sentient: false,

armor: 'Steel-plated',

energyLevel: 75

};

// What is missing in the following method call?

const robotKeys = Object.keys(robot);

console.log(robotKeys);

// Declare robotEntries below this line:

const robotEntries = Object.entries(robot)

console.log(robotEntries);

// Declare newRobot below this line:

const newRobot = Object.assign({laserBlaster: true, voiceRecognition: true}, robot);

console.log(newRobot);

* **Introduction to Classes**

JS is an ***OOP* language ; model real-world items**.

how to make *classes*. **Classes are a tool** that developers use to **quickly produce similar objects**.

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++;

}

}

**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.**

**duplicate code and debugging time.**

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**

**created a class called Dog,** and used it to produce a **Dog object.**

***the constructor method****.* JS 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**. **capitalize and CamelCase class names.**
* JS **invokes the constructor()method** every time we create a **new instance of our Dog class.**
* This **constructor() method accepts** one a**rgument**, 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**; set instance's **name property to the name argument.**
* 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.**

constructor(inputOne, inputTwo) {

this.inputOne = inputOne;

this.inputTwo = inputTwo;

}

class Surgeon{

constructor(name,department){

this.\_name=name;

this.\_department=department;

}

};

* **Instance**

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

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'

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 Dog **constructor,** which **sets the name propert**y to 'Halley'.
* Finally, we **log the value saved to the name key in** our halley **object**, which
* **Methods**

**Dog class** that **spins up objects with name and behavior properties**.

add **getters and a method**

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++;

}

}

**add getter methods for name and behavior**.

Notice, we also **prepended our property names** with **underscores (\_name and \_behavior**), which **indicate these properties should not be accessed directly.**

we **add a method named .incrementBehavior().**call .incrementBehavior() on **a Dog instance**, **it adds 1 to the \_behavior property**. we ***did not include commas.***

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');

* **Method Calls**

**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');

**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: **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**

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

we **create two new Dog instances,**nikko and Bradford.

* **Inheritance I**

class Cat {

constructor(name, usesLitter) {

this.\_name = name;

this.\_usesLitter = usesLitter;

this.\_behavior = 0;

}

get name() {

return this.\_name;

}

get usesLitter() {

return this.\_usesLitter;

}

get behavior() {

return this.\_behavior;

}

incrementBehavior() {

this.\_behavior++;

}

}

The Cat class also **contains one additional property (\_usesLitter),** that **holds a boolean value**.

When **multiple classes share properties or methods**, they become **candidates for *inheritance*.**

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

class Animal {

constructor(name) {

this.\_name = name;

this.\_behavior = 0;

}

get name() {

return this.\_name;

}

get behavior() {

return this.\_behavior;

}

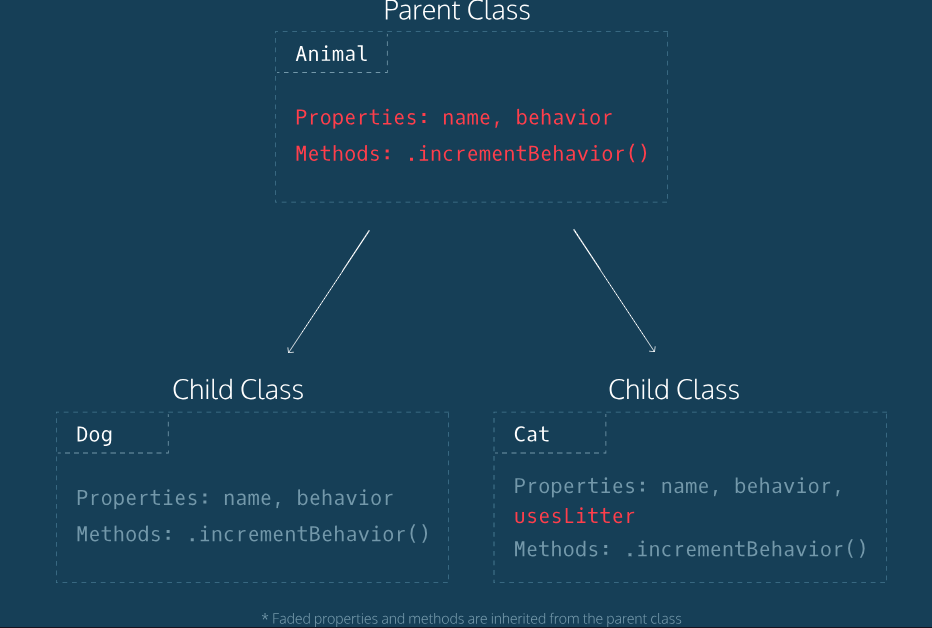
incrementBehavior() {

this.\_behavior++;

}

}

**Animal class contains the properties and methods that the Cat and Dog classes share (name, behavior, .incrementBehavior()).**



* **Inheritance II**

**The Animal class below contains the shared properties and methods of 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++;

}

}

class HospitalEmployee{

constructor(name){

this.\_name=name;

this.\_remainingVacationDays=20;

}

get name(){

return this.\_name;

}

get remainingVacationDays(){

return this.\_remainingVacationDays;

}

takeVacationDays(daysOff){

this.\_remainingVacationDays -= daysOff;

}

}

* **Inheritance III**

**shared properties and methods of our Cat and Dog classes** into a **parent class called Anima**l

class Animal {

constructor(name) {

this.\_name = name;

this.\_behavior = 0;

}

get name() {

return this.\_name;

}

get behavior() {

return this.\_behavior;

}

incrementBehavior() {

this.\_behavior++;

}

}

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;

}

}

we **create a new class** named Cat that **extends the**Animal **class**.

new keywords: **extends and super.**

* The **extends**keyword makes **the methods** of the animal **class** **available** inside the cat **class.**
* **constructor**, called when you **create a new Cat object**, 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 Cat instances.
* **\_usesLitter is a new property** that is unique to the Cat class, so we set it in the Cat constructor.

we call **super on the first line of our constructor()**, then **set the usesLitter property**.

In a **constructor(),** you must always **call the super method before you can use the this keyword** — if not, JS will throw a reference error.

**call super on the first line of subclass constructors.**

Below, we **create a new Cat instance** and call its name with the same syntax as we did with the Dog class:

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

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

we **create a new instance the Cat class**, named bryceCat. We pass it 'Bryce'and false for our name and usesLitter **arguments**.

class HospitalEmployee {

constructor(name) {

this.\_name = name;

this.\_remainingVacationDays = 20;

}

get name() {

return this.\_name;

}

get remainingVacationDays() {

return this.\_remainingVacationDays;

}

takeVacationDays(daysOff) {

this.\_remainingVacationDays -= daysOff;

}

}

class Nurse extends HospitalEmployee{

constructor(name,certifications){

super(name);

this.\_certifications=certifications;

}

}

const nurseOlynyk= new Nurse('Olynyk',['Trauma', 'Pediatrics']);

* **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);

our **Cat class extends Animal**. As a result, the **Cat class has access to the Animal getters and the .incrementBehavior() method.**

console.log(bryceCat.name);

bryceCat.incrementBehavior(); // Call .incrementBehavior() on Cat instance console.log(bryceCat.behavior); // Log value saved to behavior

* **Inheritance V**

class Cat extends Animal {

constructor(name, usesLitter) {

super(name);

this.\_usesLitter = usesLitter;

}

get usesLitter() {

return this.\_usesLitter;

}

}

class Cat {

constructor(name, usesLitter) {

this.\_name = name;

this.\_usesLitter = usesLitter;

this.\_behavior = 0;

}

get name() {

return this.\_name;

}

get usesLitter() {

return this.\_usesLitter;

}

get behavior() {

return this.\_behavior;

}

incrementBehavior() {

this.\_behavior++;

}

}

class Dog extends Animal {

constructor(name) {

super(name);

}

}

The **push()** method adds one or more elements to the end of an array and returns the new length of the array.

var sports = ['soccer', 'baseball'];

var total = sports.push('football', 'swimming');

console.log(sports); // ['soccer', 'baseball', 'football', 'swimming']

console.log(total); // 4

class HospitalEmployee {

constructor(name) {

this.\_name = name;

this.\_remainingVacationDays = 20;

}

get name() {

return this.\_name;

}

get remainingVacationDays() {

return this.\_remainingVacationDays;

}

takeVacationDays(daysOff) {

this.\_remainingVacationDays -= daysOff;

}

}

class Nurse extends HospitalEmployee {

constructor(name, certifications) {

super(name);

this.\_certifications = certifications;

}

get certifications(){

return this.\_certifications;

}

addCertification(newCertification){

this.\_certifications.push(newCertification);

}

}

const nurseOlynyk = new Nurse('Olynyk', ['Trauma','Pediatrics']);

nurseOlynyk.takeVacationDays(5);

console.log(nurseOlynyk.remainingVacationDays);

nurseOlynyk.addCertification('Genetics');

console.log(nurseOlynyk.certifications);

* **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.**

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];

}

}

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**

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).**

**class HospitalEmployee {**

**constructor(name) {**

**this.\_name = name;**

**this.\_remainingVacationDays = 20;**

**}**

**get name() {**

**return this.\_name;**

**}**

**get remainingVacationDays() {**

**return this.\_remainingVacationDays;**

**}**

**takeVacationDays(daysOff) {**

**this.\_remainingVacationDays -= daysOff;**

**}**

**static generatePassword() {**

**return Math.floor(Math.random() \* 10000);**

**}**

**}**

**class Nurse extends HospitalEmployee {**

**constructor(name, certifications) {**

**super(name);**

**this.\_certifications = certifications;**

**}**

**get certifications() {**

**return this.\_certifications;**

**}**

**addCertification(newCertification) {**

**this.certifications.push(newCertification);**

**}**

**}**

**const nurseOlynyk = new Nurse('Olynyk', ['Trauma','Pediatrics']);**

**nurseOlynyk.takeVacationDays(5);**

**console.log(nurseOlynyk.remainingVacationDays);**

**nurseOlynyk.addCertification('Genetics');**

**console.log(nurseOlynyk.certifications);**

**BROWSER COMPATIBILITY AND TRANSPILATION**

* **Introduction**

addressing security vulnerabilities, adding features**, and supporting new HTML, CSS, and JavaScript syntax.**

a **period before a software update** is released when there **are security vulnerabilities and unsupported language syntax; how developers address the gap between the new JavaScript syntax** that they use and **the JavaScript syntax that web browsers recognise**.

**widespread concern** for web developers since **Ecma International**, the organisation responsible for **standardising JS**, released a **new version of it in 2015**, called **ECMAScript2015**, commonly referred to **as ES6.**

new **ES6 syntax**, as it **improved readability and efficiency**. However, **ES6 was not supported by most web browsers,** so developers ran into **browser compatibility issues.**

**two important tools for addressing browser compatibility issues.**

* caniuse.com — A website that provides **data on web browser compatibility for HTML, CSS, and JavaScript features**.
* Babel — **A Javascript library** that you can use to **convert new, unsupported JavaScript (ES6), into an older version (ES5**) that is **recognised by most modern browsers.**

Let's get started by running ES6 JavaScript on a fake old web browser version.

var pasta = "Spaghetti"; // **ES5 syntax**

const meat = "Pancetta"; *// ES6 syntax*

let sauce = "Eggs and cheese"; /*/ ES6 syntax*

// Template literals, like the one below, were introduced in ES6

const carbonara = `You can make carbonara with ${pasta}, ${meat}, and a sauce made with ${sauce}.`;

* **caniuse.com I**

Since Ecma's release of ECMAScript2015 (ES6), software companies have slowly added support for ES6 features and syntax. While most new browser versions support the majority of the ES6 library, **there are still a couple sources of compatibility issues:**

* **Some users have not updated** to the latest, **ES6 supported web browser** version.
* A few **ES6 features, like modules**, are still **not supported** by most web browsers.

Because companies **add support for ES6 features gradually**, it's important for you to know how to **look up browser support on a feature-by-feature basis**. The website [caniuse.com](http://caniuse.com/" \t "_blank) is the best resource for **finding browser compatibility information.**

In ***caniuse*,** you can **enter an ES6 feature**, like **let**, and see the **percentage of browsers that recognise it.** You can also see when each **major web browser** (Chrome, Safari, Edge, etc.) added **support for the keyword.**

* **Why ES6?**

Before we learn how to **set up a JavaScript project that converts ES6 to an older version**, it's worth understanding a few of the **reasons Ecma made such substantial updates**.

Three reasons for the ES5 to ES6 updates are listed below:

* A **similarity to other programming languages** — JavaScript ES6 is syntactically more **similar to other object-oriented programming languages**. This leads to less friction when experienced, non-JavaScript developers want to learn JavaScript.
* **Readability and economy of code**
* Addresses sources of ES5 bugs.

Because ES6 addressed the above issues, Ecma knew that **adoption by web developers would occur quickly**, while **web browser support lagged behind.**

Ecma made the **new syntax backwards compatible,** which means you **can map JavaScript ES6 code to ES5.**

**The following line of code is written with ES6 string interpolation:**

**`You can make carbonara with ${pasta}, ${meat}, and a sauce made with ${sauce}.`**

**In ES5, we used plus signs (+) to interpolate variables into strings.**

**In main.js, use ES5 string interpolation to set carbonara to the same string as above.**

**var pasta = "Spaghetti"; // ES5 syntax**

**var meat = "Pancetta"; // ES6 syntax**

**var sauce = "Eggs and cheese"; // ES6 syntax**

**// Template literals, like the one below, were introduced in ES6**

**var carbonara = "You can make carbonara with" + pasta+","+ meat+", and a sauce made with" sauce+".";**

* **Transpilation With Babel**

Because **ES6 is predictably backwards compatible**, a collection of **JS programmers developed a JS library called *Babel* that *transpiles* ES6 JavaScript to ES5.**

**Transpilation i**s the process of **converting one programming language to another**. Use Babel to transpile the new, **easy-to-write version of JavaScript (ES6) to the old, browser-compatible version of JavaScript (ES5).** In the instructions below, you will **pass JavaScript ES6 code to Babel**, which **will transpile it to ES5** and *write it to a file in the****lib****directory.*

* **Terminal:**

npm install babel-cli

npm install babel-preset-env

installs the two required Babel packages.

npm run build

can view the ES5 code in **./lib/main.js**.

* **npm init**

**one command in your terminal to transpile ES6 code to ES5**.

**setup a JS project that transpiles cod**e when you **run npm** **run build**from the **root directory of a JS project.**

The first step is to **place your ES6 JavaScript file in a directory called src.** From your root directory, the path to the ES6 file is **./src/main.js**

The **initial JS project file structure** is:

project

|\_ src

|\_\_\_ main.js

**Before we install Babel**, we need to **setup our project to use the [node package manager (npm)](https://docs.npmjs.com/getting-started/what-is-npm" \t "_blank).** <https://docs.npmjs.com/about-npm/index.html>

Developers **use *npm* to access and manage Node packages**. **Node packages** are **directories** that contain **JavaScript code written** by other developers. You can use these packages to **reduce duplication of work and avoid bugs.**

**Before we can add Babel to our project directory**, we **need to run npm init.** **The npm init command** creates a **package.json** file in the root directory.

A **package.json** file **contains information** about the **current JS project**. Some of this information includes:

* **Metadata** — This includes a **project title, description, authors, and more.**
* **A list** **of node packages required for the project** — If **another developer wants to run your project**, **npm** looks inside **package.json** and **downloads the packages in this list.**
* **Key-value pairs for command line scripts** — You can use **npm** to run these shorthand scripts to perform some process. In a later exercise, we will add a script that runs Babel and transpiles ES6 to ES5.

If you **have Node installed on your computer**, you can create a **package.json** file by typing npm init into the terminal.

The **terminal prompts you to fill in fields for the project's metadata (name, description, etc.)**

If you **don't want to fill in a field, you can press enter. npm will leave fill these fields with default values** or leave them empty when it creates the **package.json** file.

**After** you **run npm init**your **directory structure will contain the following files and folders**:

project

|\_ src

|\_\_\_ main.js

|\_ package.json

npm adds the **package.json** file to the *same level as the****src****directory.*

*When inside the project directory, run* **npm init**  *command in Terminal.*

* **Install Node Packages**

We use the **npm install command** to **install new Node packages locally**. The **install command** creates a folder called **node\_modules** and copies the package files to it and installs all of the **dependencies for the given package**.

To **install Babel**, we **need to npm install two packages**, *babel-cli and babel-preset-env*. However, **npm installs over one hundred other packages that are dependencies** for **Babel** to run properly.

We **install Babel** with the following two commands:

$ npm install babel-cli -D

$ npm install babel-preset-env –D

The ***babel-cli package*** includes command line Babel tools, and the **babel-preset-env package** has the **code that maps** any JavaScript feature, **ES6 and above (ES6+), to ES5.**

The **-D flag** instructs **npm** to **add each package to a property called devDependencies in package.json.** Once the **project's dependencies are listed in devDependencies,** other developers can run your project without installing each package separately. Instead, **they can simply run npm install**— it instructs npm to look inside **package.json** and **download all of the packages listed in devDependencies.**

Once you **npm install packages**, you can find the **Babel packages** and **all their dependencies** *in the****node\_modules****folder.*

The new directory structure contains the following:

project

|\_ node\_modules

|\_\_\_ .bin

|\_\_\_ ...

|\_ src

|\_\_\_ main.js

|\_ package.json

The ... in the file structure above is a placeholder for 100+ packages that npm installed.

* **.babelrc**

Now that you've **downloaded the Babel packages**, you need to **specify the version of the source JavaScript code ;** inside of a file named **.babelrc**. In your *root directory*, you can ***run touch .babelrc to create this file.***

Your project directory contains the following folders and files:

project

|\_ node\_modules

|\_\_\_ .bin

|\_\_\_ ...

|\_ src

|\_\_\_ main.js

|\_ .babelrc

|\_ package.json

Inside **.babelrc** you need to **define the *preset* for your source JavaScript file**. The **preset specifies the version of your initial JavaScript** file.

To **specify that we are transpiling code from an ES6+ source**, we have to add the following **JS object** into **.babelrc**:

{

"presets": ["env"]

}

When you **run Babel**, it looks in **.babelrc** **to determine the version of the initial JavaScript file**. In this case*,****["env"]*instructs Babel to transpile any code from versions ES6 and later.**

* **Babel Source Lib**

We need to *specify a script* in **package.json** that **initiates the ES6+ to ES5 transpilation**.

Inside of the **package.json** file, there is a *property named "scripts"*that **holds an object for specifying command line shortcuts**. It looks like this:

... "scripts": { "test": "echo \"Error: no test specified\" && exit 1" }, ...

In the code above, the **"scripts" property contains an object with one property called "test".**

Below the "test" property, we **will add a script that runs Babel** like this:

... "scripts": { "test": "echo \"Error: no test specified\" && exit 1", **"build": "babel src -d lib"** }

In the **"scripts" object** above, we **add a property called "build".** The property's value, *"babel src -d lib",* is a **command line method** that **transpiles ES6+ code to ES5.**

Let's consider each argument in the method call:

* babel — The Babel **command call responsible for transpiling code**.
* src — Instructs Babel to **transpile all JS code inside the src directory**.
* -d — Instructs Babel to **write the transpiled code to a directory.**
* lib — Babel writes the **transpiled code to a directory called lib**.
* **Build**

In the last exercise, we wrote the following script in **package.json**:

"build": "babel src -d lib"

Now, we **need to call "build" from the command line to transpile and write ES5 code to a directory called lib.**

From the command line, we type:

***npm run build***

The *command above runs the build script* in **package.json**.

Babel **writes the ES5 code** to a file named **main.js** (it's always the same name as the original file), inside of **a folder called lib**. The resulting directory structure is:

project

|\_ lib

|\_\_\_ main.js

|\_ node\_modules

|\_\_\_ .bin

|\_\_\_ ...

|\_ src

|\_\_\_ main.js

|\_ .babelrc

|\_ package.json

Notice, the directory contains a new folder named **lib**, with one file, called **main.js**.

The **npm run build** command will *transpile all JS files inside of the****src****folder*. This is helpful as you build larger JavaScript projects — **regardless of the number of JavaScript files**, you only need to **run one command (npm run build) to transpile all of your code**.

* Intermediate JS modules:
* **Hello Modules**

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

By **separating code** with **similar logic into files called modules**, we can:

* **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**

**defining a module in one file and making the module available for use in another file**.

*how to define a module* and *how to export it* using the **statement module.exports.**

In ***menu.js*** we write:

let Menu = {};

Menu.specialty = "Roasted Beet Burger with Mint Sauce";

module.exports = Menu;

1. **let Menu** = {}; **creates the object** that **represents the module**Menu. The statement contains an **uppercase variable named**Menu which is **set equal to an object.**
2. **Menu.specialty**is defined as a **property of the Menu module**. We add data to the Menu object by **setting properties on that object** **and giving** the **properties a value.**
3. "Roasted Beet Burger with Mint Sauce"; is the **value stored in the Menu.specialty property.**
4. module.exports = Menu; **exports the Menu object** as a **module**. **module is a variable** that represents **the module,** and **exports exposes the module as an object.**

The *pattern we use to export modules* is thus:

1. **Define an object to represent the module.**

**create empty object and save it within a variable.**

1. **Add data or behavior to the module.**
2. **Expor**t 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.***

want the **module** to control the menu's data and behavior, and we want a separate file to handle placing an order. 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);

}

placeOrder();

We now have the **entire behavior of Menu available in order.js**. Here, we notice:

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 JS 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**.
4. The **placeOrder() function** then uses the **Menu module** in its function definition. By calling **Menu.specialty**, we access the property specialty defined in the Menu module.
5. We can then **invoke the function using placeOrder()**

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.**

const Airplane = require('./1-airplane.js');

function displayAirplane(){

console.log(Airplane.myAirplane);

}

displayAirplane();

* **module.exports II**

We can also **wrap any collection of data and functions in an ob**ject, 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.

File 1:

let Airplane={};

module.exports={

myAirplane:"CloudJet",

displayAirplane: function(){

return this.myAirplane;

}

};

File 2:

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

console.log(Airplane.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*.

**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.

let Airplane= {availableAirplanes:[]};

Airplane. availableAirplanes=[

{

name:'AeroJet',

fuelCapacity:800

},

{

name:'SkyJet',

fuelCapacity:500

}

];

export default Airplane;

* **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.

import Airplane from './airplane';

function displayFuelCapacity() {

Airplane.availableAirplanes.forEach(function(element){

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

});

}

displayFuelCapacity();

let Airplane= {

availableAirplanes:[]

};

Airplane. availableAirplanes=[

{

name:'AeroJet',

fuelCapacity:800

},

{

name:'SkyJet',

fuelCapacity:500

}

];

export default Airplane;

* **Named Exports**

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

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. 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.
4. specialty and isVegetarian are exported, while isLowSodium is not exported, since it is not specified.

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};**

* **Named Imports**

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

In the **order.js** file:

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.**

**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.

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;

}

};

* **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';

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));

})

};

function displaySpeedRangeStatus(){

availableAirplanes.forEach(function(element)

{

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

});

};

displayStaffStatus();

displaySpeedRangeStatus();

* **Export as**

**Named exports** also conveniently **offer a way to change the name of variables** **when we export or import them.** ; 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 };

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.**

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 };

* **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 Carte module.

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

function displayFuelCapacity() {

aircrafts.forEach(function(element){

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

});

}

function displayStaffStatus(){

aircrafts.forEach(function(element){

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

})

};

function displaySpeedRangeStatus(){

aircrafts.forEach(function(element)

{

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

});

};

displayStaffStatus();

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.

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;

}

}

function meetsSpeedRangeRequirements(maxSpeed, minSpeed, requiredSpeedRange){

let range= maxSpeed -minSpeed;

if (range > requiredSpeedRange){

return true;

} else {

return false;

}

};

export default meetsSpeedRangeRequirements;

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

* **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';

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);

});

}

function displayStaffStatus(){

availableAirplanes.forEach(function(element){

console.log(element.name + 'meets staff requirements: ' + meetsStaffRequirements(element.availableStaff, flightRequirements.requiredStaff));

})

};

function displaySpeedRangeStatus(){

availableAirplanes.forEach(function(element)

{

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

});

};

displayStaffStatus();

displaySpeedRangeStatus();

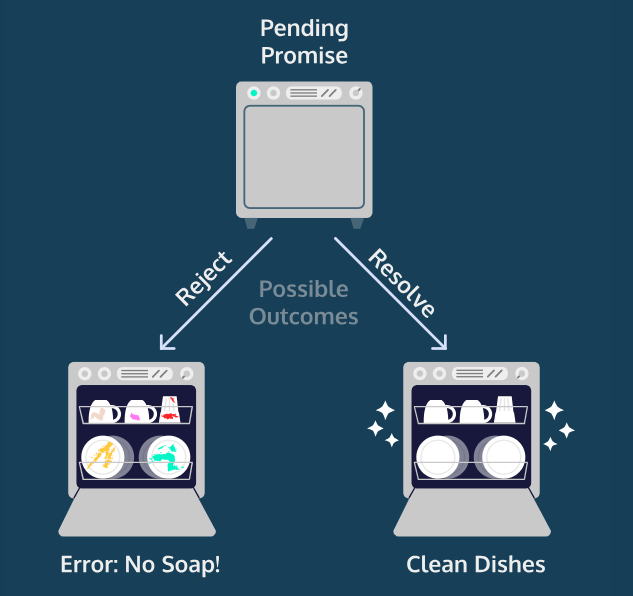
* **JAVASCRIPT PROMISES**
* **Introduction**

An ***asynchronous operation***is one that **allows the computer to "move on" to other tasks** *while waiting for the asynchronous operation to complete*. Asynchronous programming means that time-consuming operations don't have to bring everything else in our programs to a halt.

Cleaning our house, for example, 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.**

how modern JavaScript handles asynchronicity using the Promise object, introduced with ES6. Let's get started!



* **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*. Let's think of a dishwasher as having the states of a promise:

* **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 **promise is fulfilled, we'll be able to perform related tasks**. **If it's rejected, we can take alternate steps**.

**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**

To **create a new Promise object**, we use the **new keyword** and the **Promise constructor method**:

const **executorFunction** = (resolve, reject) => { };

const myFirstPromise = new Promise(**executorFunction)**;

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, JS 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 functio**n 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 Promise constructor:

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.**

* **The Node setTimeout() Function**

how to *consume*, or use, promises will be key, **handling Promise objects returned to you** as the **result of an asynchronous operation**. These promises *will start off pending but settle eventually*.

**functions** that **return promises** which **settle after some time**.

setTimeout() is a Node API that **uses callback functions** to **schedule tasks to be performed after a delay**.

*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**.

This **delay is performed asynchronously**— the **rest of our program won't stop executing during the delay.** Asynchronous JS uses ***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.

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();

**returnPromiseFunction() which returned a promise**; promise to the **variable prom.** Similar to the asynchronous promises you may encounter in production, **prom will initially have a status of pending.**

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

const usingSTO = () => {

console.log("This is NOT the first line of synchronous code.");

};

setTimeout(usingSTO, 500);

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

* **Consuming Promises**

**initial state of an asynchronous promise is pending**, but **guarantee that it will settle.**

tell the computer ***what should happen after settling?*** **Promise objects** come with an aptly named ***.then() method.*** It allows us to say, "*I have a promise, when it settles,****then****here's what I want to happen..*."

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.**

We can ***invoke .then() with one, both, or neither handler!***

**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.** One important feature of **.then() is that it always returns a promise.**

* **The onFulfilled and onRejected Functions**

To **handle a "successful" promise**, **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.*

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);

* **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.

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

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

**app.js:**

const handleSuccess = (resolvedValue) => {

console.log(resolvedValue);

};

const handleFailure = (rejectReason) => {

console.log(rejectReason);

};

checkInventory(order)

.then(handleSuccess, handleFailure);

const inventory = {

sunglasses: 1900,

pants: 1088,

bags: 1344

};

**library.js:**

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 };

* **Using catch() with Promises**

cleaner code is to follow a principle called ***separation of concerns***; organizing code **into distinct sections each handling a specific task**.

Remember, .**then() will return a promise** with the ***same settled value as the promise it was called on if no appropriate handler was provided.***

*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);

});

we follow a common convention of **putting each part of this chain on a new line to make it easier to read.** To create *even more readable code, we can use a different promise function:****.catch()***

The **.catch() function takes only one argument, onRejected.** In the case of a **rejected promise,** this **failure handler will be invoked with the reason for rejection**. Using***.catch() accomplishes the same thing as using a .then() with only a failure handler.***

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.**

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

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

const handleSuccess = (resolvedValue) => {

console.log(resolvedValue);

};

const handleFailure = (rejectReason) => {

console.log(rejectReason);

};

checkInventory(order).then(handleSuccess).catch(handleFailure);

* **Chaining Multiple Promises**

asynchronous programming is **multiple operations which depend on each other to execute or that must be executed in a certain order.**

We take our dirty clothes and put them in the washing machine. 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****.*

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) => {

return shipOrder(resolvedValueArray);

})

.then((successMessage) => {

console.log(successMessage);

})

.catch((errorMessage) => {

console.log(errorMessage);

});

* **Avoiding Common Mistakes**

**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);

})

})

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

* We invoke returnsFirstPromise() **which returns a promise.**
* We invoke .**then() with a success handler.**
* Inside the success handler, we invoke **returnsSecondValue()**with **firstResolveVal which will return a new promise.**
* We invoke a second .then() to handle the logic for **the second promise settling** all **inside** the first then()!
* Inside that second .then(), **we have a success handler** which will **log the second promise's resolved value to the console.**

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

returnsFirstPromise()

.then((firstResolveVal) => {

returnsSecondValue(firstResolveVal)

})

.then((someVal) => {

console.log(someVal);

})

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

* We **invoke returnsFirstPromise() which returns a promise**.
* We invoke**.then() with a success handler.**
* Inside the success handler, **we create our second promise**, ***but we forget to return it!***
* We invoke a second .then(). **It's supposed to handle the logic for the second promise**, but since we didn't return**, this .then() is invoked on a promise with the same settled value as the original promise!**

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

**app.js:**

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);

});

* + **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**?

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

myPromises

.then((arrayOfValues) => {

console.log(arrayOfValues);

})

.catch((rejectionReason) => {

console.log(rejectionReason);

});

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.**

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

const onFulfill = (itemsArray) => {

console.log(`Items checked: ${itemsArray}`);

console.log(`Every item was available from the distributor. Placing order now.`);

};

const onReject = (rejectionReason) => {

console.log(rejectionReason);

};

// Write your code below:

const checkSunglasses = checkAvailability('sunglasses', 'Favorite Supply Co.');

const checkPants = checkAvailability('pants', 'Favorite Supply Co.');

const checkBags = checkAvailability('bags', 'Favorite Supply Co.');

Promise.all([checkSunglasses, checkPants, checkBags])

.then(onFulfill)

.catch(onReject);

* Async Await
* Introduction

**Introduction**

,

we need to **handle asynchronous actions**— actions **we can wait on** while moving on to other tasks. We make r**equests to networks, databases, or any number of similar operations**.

JS is **non-blocking**: instead of stopping the execution of code while it waits, JavaScript uses **an [event-loop](https://youtu.be/8aGhZQkoFbQ" \t "_blank)** 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. With **ES6**, JavaScript **integrated native [promises](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise" \t "_blank)**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" \t "_blank)**— 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" \t "_blank). Both of these were already built in to the language.

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()

* **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'll be using async function declarations throughout this lesson, but 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**

async 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.

async function ***returns a promise with a resolved value equal to the return value of that function.***

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}.`);

})

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}.`);

})

* **The await Operator**

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

In most situations, we're dealing with **promises that were returned from functions.** Generally, these functions are through a library. We can **await the resolution of the promise it returns inside an async function**. 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.**

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:

async function announceDinner() {

let resolveValue= await brainstormDinner();

console.log(`I'm going to make ${resolveValue} 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***.

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!

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(), value was assigned the promise object itself.***

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();

/\*

This is the shopForBeans function. It uses a setTimeout() function to simulate a time-consuming asynchronous action. The function returns a promise with a resolved value of a string representing a type of bean. It settles on a random beanType from the beanTypes array using Math.random().

\*/

const shopForBeans = () => {

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

const beanTypes = ['kidney', 'fava', 'pinto', 'black', 'garbanzo'];

setTimeout(()=>{

let randomIndex = Math.floor(Math.random() \* 4)

let beanType = beanTypes[randomIndex];

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

resolve(beanType);

}, 1000)

})

}

module.exports = shopForBeans

* **Handling Dependent Promises**

**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);

});

}

Let's break down what's happening in the nativePromiseVersion() function:

* Within our function we use **two functions which return promises***: returnsFirstPromise() and returnsSecondPromise().*
* We invoke returnsFirstPromise() and ensure that the first promise resolved by using .then().
* In the callback of our first .then(), we log the resolved value of the first promise, firstValue, and then **return returnsSecondPromise(firstValue).**
* We use another .then() to print the **second promise's resolved value to the console.**

async function asyncAwaitVersion() {

let firstValue = await returnsFirstPromise();

console.log(firstValue);

let secondValue = await returnsSecondPromise(firstValue);

console.log(secondValue);

}

Let's break down what's happening in our asyncAwaitVersion() function:

* We mark **our function as async**.
* Inside our function, we **create a variable firstValue**assigned **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 secondValueassigned to await returnsSecondPromise(firstValue).** Therefore, secondValue is **assigned this promise's resolved value.**
* Finally, we log secondValue to the console.

Given the two versions of the function, **the async...await version more closely resembles synchronous code**, which helps developers maintain and debug their code. The async...await syntax 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.

const shopForBeans = () => {

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

const beanTypes = ['kidney', 'fava', 'pinto', 'black', 'garbanzo'];

setTimeout(()=>{

let randomIndex = Math.floor(Math.random() \* 4)

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}

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

async function makeBeans() {

let type = await shopForBeans();

let isSoft = await soakTheBeans(type);

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.**

With **async...await, we use try...catch statements** for ***error handling. By*** using this syntax, not only are we able to handle errors in the same way we do with synchronous code, but we can also **catch both synchronous and asynchronous errors.**

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 async function

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.***

//This function returns true 50% of the time.

let randomSucess = () => {

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 = randomSucess();

If(success){

resolve('Bean Souffle');

} else {

reject('Dinner is ruined!');

}

}, 1000);

})

};

module.exports = cookBeanSouffle;

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

async function hostDinnerParty(){

try{

let firstPromiseReturned = await cookBeanSouffle();

console.log(`${firstPromiseReturned} is served!`);

} catch(error) {

console.log(error);

console.log('Ordering a pizza!');

}

}

hostDinnerParty();

* **Handling Independent Promises**

Remember that **await halts the execution of our async function**. 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 async functions we should still take advantage of ***concurrancy, the ability to perform asynchronous actions at the same time.***

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

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

async function serveDinner(){

let vegetablePromise = steamBroccoli();

let starchPromise = cookRice();

let proteinPromise = bakeChicken();

let sidePromise = cookBeans();

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

}

serveDinner();

let cookBeans = () => {

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

setTimeout(()=>{

resolve('beans')

}, 1000)

})

}

let steamBroccoli = () => {

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

setTimeout(()=>{

resolve('broccoli')

}, 3000)

})

}

let cookRice = () => {

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

setTimeout(()=>{

resolve('rice')

}, 2000)

})

}

let bakeChicken = () => {

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

setTimeout(()=>{

resolve('chicken')

}, 1000)

})

}

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

* **Await Promise.all()**

Another way to take advantage of **concurrenc**y 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]);

}

}

In our above example, we ***await the resolution of a Promise.all().*** This Promise.all() was *invoked with an argument array containing four promises* (returned from required-in functions). Next, we **loop through our resultArray**, and log each item to the console. The first element in resultArray is the **resolved value of the asyncTask1() promise, the** second is the value of the asyncTask2() promise, and so on.

**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.***

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

async function serveDinnerAgain(){

let 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();

* Requests
* **Introduction to Requests**

There are **many types of HTTP requests**. The four most commonly used types of **HTTP requests** are **GET, POST, PUT, and DELETE**. In this lesson, we'll cover **GET and POST** requests. If you want to learn more about the different HTTP requests, we recommend the following documentation:

<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.

In this lesson, we will explain 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.**

<https://www.codecademy.com/articles/rebrandly-signup>

* **HTTP Requests**

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

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.*

JS 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 request*s** 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>

* **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.

<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**. Take a look at the boilerplate code in the diagram to see how to make an XHR GET request.



* **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.***

The **XMLHttpRequest object** is **used in JavaScript to create and send requests**. To create a new instance of an object, you would use the new keyword. Like so:

JSON is **JavaScript Object Notation**, which is **how the response is going to be formatted**.

.onreadystatechange of xhr will contain an **event listener which will execute when the readyState property changes.**

const xhr = new XMLHttpRequest();

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

xhr.responseType = 'json';

xhr.onreadystatechange = () => {

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

return xhr.response;

}

};

//conditional statement checks to see if the request has finished.

xhr.open('GET', url);

xhr.send();

.open() creates a new request and the arguments passed in determine the type and URL of the request.

* **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!**

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

'rel\_rhy=' is the start of a parameter for the query string. This parameter will narrow your search to words that rhyme.

// Information to reach API

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

const queryParams = 'rel\_rhy=';

// Selecting page elements

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

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

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

// AJAX function

const getSuggestions = () => {

const wordQuery = inputField.value;

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

const xhr = new XMLHttpRequest();

xhr.responseType = 'json';

xhr.onreadystatechange = () => {

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

renderResponse(xhr.response)

}

}

xhr.open('GET',endpoint);

xhr.send();

}

// Clear previous results and display results to webpage

const displaySuggestions = (event) => {

event.preventDefault()

while(responseField.firstChild){

responseField.removeChild(responseField.firstChild)

};

getSuggestions();

};

submit.addEventListener('click', displaySuggestions);

* **XHR GET Requests IV**

In the previous exercise, you made a ***GET request to the Datamuse API to find words that rhyme***. In this exercise, 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.**

<https://en.wikipedia.org/wiki/Query_string>

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 =.***

'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'

the & character at the start of the string is used to separate our parameters. The = character will join the key 'topics' to a value.

// 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**

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.

Feel free to refer to the XHR POST diagram at any point while completing this exercise:

The **XMLHttpRequest object is used in JavaScript to interact with servers.**

**JSON.stringify()**will **convert a value to a JSON string**. By ***converting the value to a string, we can then send the data to a server.***

***.onreadystatechange will contain the event handler that will be called when xhr's state changes.***

***if conditional will check to see if the request has finished.***

***The response property will contain the data that we're getting back from the POST request.***

***.open() creates a new request and the arguments passed in determine what type of request is being made and where it's being made to.***

***const xhr = new XMLHttpRequest();***

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

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

***xhr.responseType= 'json';***

***xhr.onreadystatechange = () => {***

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

***return xhr.response;***

***}***

***}***

***xhr.open('POST',url);***

***xhr.send(data);***

* **XHR Post Requests III**

In this exercise, you'll be making a **POST request to the Rebrandly API to shorten a URL**. You’re now going to incorporate the previous lesson’s boilerplate code into making an actual POST request!

// Information to reach API

const apiKey = '<Your API Key>';

const url = 'https://api.rebrandly.com/v1/links';

// Some page elements

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

const shortenButton = document.querySelector('#shorten');

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

// AJAX functions

const shortenUrl = () => {

const urlToShorten = inputField.value;

const data = JSON.stringify({destination: urlToShorten});

const xhr = new XMLHttpRequest();

xhr.responseType = 'json';

xhr.onreadystatechange = () => {

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

renderResponse(xhr.response);

}

}

xhr.open('POST', url);

xhr.setRequestHeader('Content-type', 'application/json');

xhr.setRequestHeader('apikey', apiKey);

xhr.send(data);

}

// Clear page and call AJAX functions

const displayShortUrl = (event) => {

event.preventDefault();

while(responseField.firstChild){

responseField.removeChild(responseField.firstChild);

}

shortenUrl();

}

shortenButton.addEventListener('click', displayShortUrl);

// We're including this information because the API expects to see an object with a key destination that has a value of a URL.

* **Introduction to Requests with ES6**

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.**

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

* **fetch() GET Requests I**

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 III**

first argument in fetch() determines the endpoint of the request.

.then() will fire only after the promise status of fetch() has been resolved.

The reason we're testing the ok property of the response object is that it will be a Boolean value. If there were no errors response.okwill be true and then your code will return response.json().

fetch('https://api-to-call.com/endpoint')

.then(response => {

if (response.ok){

return response.json();

}

throw new Error('Request failed!');

},

networkError => {

console.log(networkError.message);

})

.then(jsonResponse => {

return jsonResponse

});

* **fetch() GET Requests III**

, you’re going to use that code and manipulate it to access the Datamuse API and render information in the browser.

* [Datamuse API](https://www.datamuse.com/api/" \t "_blank).

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

queryParams will be the start of your query string and will narrow your search to words that sounds like your word.

then add a query string to the URL with all the necessary parameters.

/ 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);

});

};

// Clears previous results and display results to webpage

const displaySuggestions = (event) => {

event.preventDefault();

while(responseField.firstChild){

responseField.removeChild(responseField.firstChild);

}

getSuggestions();

};

submit.addEventListener('click', displaySuggestions);

* **fetch() GET Requests IV**

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**.

take the **information that was returned with the promise** and **manipulate the webpage!**

// 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();

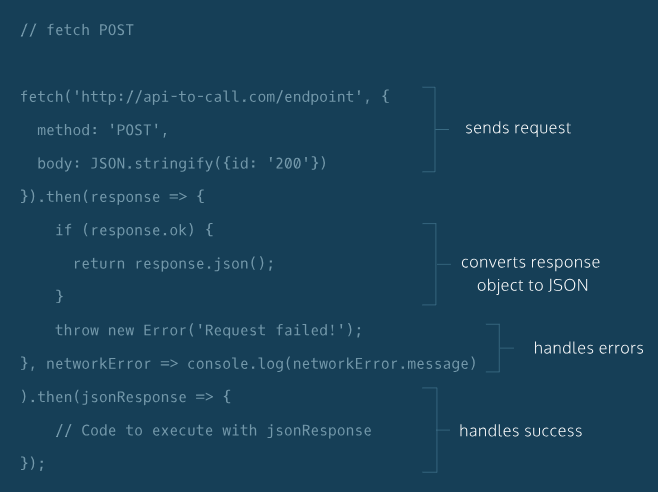
};

submit.addEventListener('click', displaySuggestions);

* fetch() POST Requests I

to **use fetch() to construct POST requests!**

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.



calling fetch() and providing an endpoint.

This second argument of fetch() determines that this request is a POST request and what information will be sent to the API.

 code inside .then() will execute when the promise returned from fetch() is resolved.

throw new Error('Request failed!');

This error will be raised if we get back some status error.

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

creating data is to prepare the information needed to send in the body.

// Information to reach API

const apiKey = '<f428afeffc674d79992d647b40fae9a2>';

const url = 'https://api.rebrandly.com/v1/links';

// Some page elements

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

const shortenButton = document.querySelector('#shorten');

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

// AJAX functions

const shortenUrl = () => {

const urlToShorten = inputField.value;

const data = JSON.stringify({destination: urlToShorten});

fetch(url,{

method: 'POST',

headers:{

'Content-type': 'application/json',

'apikey': apiKey

},

body: data

});

}

// Clear page and call AJAX functions

const displayShortUrl = (event) => {

event.preventDefault();

while(responseField.firstChild){

responseField.removeChild(responseField.firstChild)

}

shortenUrl();

}

shortenButton.addEventListener('click', displayShortUrl);

* **fetch() POST Requests IV**

In the previous exercise you’ve positioned yourself to **make the POST request by providing the endpoint and the object containing all the necessary information.** In this exercise **you’ll handle the response.**

The **request returns a Promise** which will **either be resolved or rejected**. If the **promise resolves**, you can **use that information and the ok status**.

// Information to reach API

const apiKey = '<f428afeffc674d79992d647b40fae9a2>';

const url = 'https://api.rebrandly.com/v1/links';

// Some page elements

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

const shortenButton = document.querySelector('#shorten');

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

// AJAX functions

const shortenUrl = () => {

const urlToShorten = inputField.value;

const data = JSON.stringify({destination: urlToShorten});

fetch(url,{

method: 'POST',

headers:{

'Content-type': 'application/json',

'apikey': apiKey

},

body: data

})

.then(response => {

if(response.ok){

return response.json();

}

throw new Error('Request failed!');

}, networkError => {

console.log(networkError.message);

})

}

// Clear page and call AJAX functions

const displayShortUrl = (event) => {

event.preventDefault();

while(responseField.firstChild){

responseField.removeChild(responseField.firstChild)

}

shortenUrl();

}

shortenButton.addEventListener('click', displayShortUrl);

* **fetch() POST Requests V**

add another .then() to the chain to finally make the information available to your webpage!

// Information to reach API

const apiKey = '<f428afeffc674d79992d647b40fae9a2>';

const url = 'https://api.rebrandly.com/v1/links';

// Some page elements

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

const shortenButton = document.querySelector('#shorten');

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

// AJAX functions

const shortenUrl = () => {

const urlToShorten = inputField.value;

const data = JSON.stringify({destination: urlToShorten});

fetch(url,{

method: 'POST',

headers:{

'Content-type': 'application/json',

'apikey': apiKey

},

body: data

})

.then(response => {

if(response.ok){

return response.json();

}

throw new Error('Request failed!');

}, networkError => {

console.log(networkError.message);

})

.then(jsonResponse => {

renderResponse(jsonResponse);

})

}

// Clear page and call AJAX functions

const displayShortUrl = (event) => {

event.preventDefault();

while(responseField.firstChild){

responseField.removeChild(responseField.firstChild)

}

shortenUrl();

}

shortenButton.addEventListener('click', displayShortUrl);

**Linking JavaScript**

We can **link a JavaScript file to HTML** by including it **as the src of a <script> tag** inside of an HTML file, like this:

***<script src='js/main.js'></script>***

This line of code **will link the file located at js/main.js.** By linking **js/main.js** in the **index.html** file, we ***are asking the browser to run our JavaScript code each time index.html loads.***

alert is a JavaScript function that will create a pop-up window with text inside it.

…………

</div>

</div>

<script src='js/main.js'></script>

</body>

</html>

* **Document Object Model**

The ***Document Object Model,*** commonly referred to as the ***DOM*'**, is the term for ***elements* in an HTML file**. Elements are any HTML code denoted by HTML tags, like <div>, <a>, or <p>.

Let's **use JavaScript to interact with the DOM.**

We can **select an HTML element with JavaScript by selecting its class attribute,** like this:

var header = document.getElementsByClassName('example-class-name');

This would find an element like this in the HTML:

<div class='example-class-name'> ... </div>

var skillset = document.getElementsByClassName('skillset');

var skillset = document.getElementsByClassName('skillset');

alert(skillset);

* **jQuery**

We've just covered **how to select HTML elements using the syntax: document.getElementsByClassName**. ***simpler way to select DOM elements***?

To **better interact with DOM elements, we can use a *library***. A library is a **set of code that contains useful pre-written functions that help with certain tasks**; great library for interacting with the DOM is***jQuery***. **jQuery is a library written in JavaScript**.

In order to use jQuery, we need to:

1. Include **jQuery in our project**. jQuery is a **library,** which **means it is a set of code in a file**, therefore we will **need to link that file in our HTML** in order to access it.

Once we link it in our HTML file, we can use its functions and syntax in our **js/main.js** file.

1. Once linked, we'll need to make sure **our HTML is loaded** **before we run our jQuery and JavaScript code.**

This will **prevent our jQuery and JavaScript code from running before the elements they select are rendered.**

**Linking to jQuery library, before link to js/main.js:**

<script src='https://code.jquery.com/jquery-3.1.0.min.js'></script>

<script src='js/main.js'></script>

</body>

</html>

**jQuery has a built in function to check if the page is ready before it will run our code**. After the main function, write this code:

$(document).ready(main);

we put **main** inside the **parentheses of ready**. ***main here is a callback***, which means that **our code will wait until the document**(in other words, the DOM) is **loaded**, or ready. When it is, ***then* it will execute the main function.** **jQuery calls back to the main function, therefore it's a callback.** In the event that our HTML and CSS took 5 minutes to load, **this code would wait until it loaded completely before running**

 function main(){};

$(document).ready(main);

* **jQuery Selectors**

With plain JavaScript we selected an HTML element with this code:

**document.getElementsByClassName('skillset');**

With jQuery we can select the same element with:

**$('.skillset');**

1. We can wrap **any CSS selector,** like class, id, or tag, with **$('.example-class') to select it with jQuery.**
2. The **selectors jQuery uses are the exact same as CSS selectors**. For instance, if there's an element with a class of supporting-text, you could select it with $('.supporting-text'). Another example, if an **element had an id of 'header'**, you could **select it with $('#header').**

It is a common convention to name variables that hold jQuery selectors with a dollar sign $.

function main(){

var $skillset = $('.skillset');

alert($skillset);

};

$(document).ready(main);

* **hide**

First off, it would be nice to **make the page fade in when loaded.**

To make a page fade in, **it must first be hidden.** We can **hide elements with jQuery** with a ***function named hide.***

We can hide elements with jQuery, like this:

$('.my-selector').hide();

1. We **attached the hide function directly to the jQuery selector**.
2. The hide function will add the CSS property *display: none*to the DOM element from the page, which will hide it.

function main(){

$('.skillset').hide();

};

$(document).ready(main);

added the CSS property *display: none*to the DOM element

* **fadeIn**

In order to **fade in the skillset element,** we can use a the **jQuery function named fadeIn.**

True to its name, fadeIn **will fade an element in over a period of time in milliseconds.** It looks like this:

$('.example-class').fadeIn(400);

1. We **must start with an element that is not currently displayed on the page**.
2. Just like before, we **can attach fadeIn directly to a jQuery selector.**
3. Within fadeIn's parentheses, we can **specify how long we want the fade to last in milliseconds.** 400 is the default.
4. The example code will fade in the '.example-class' element over 0.4 seconds.

function main(){

$('.skillset').hide();

$('.skillset').fadeIn(1000);

};

$(document).ready(main);

* **click**

The next feature we'd like to build is **making the 'Recent Projects' clickable**. When **clicked**, the button should show the individual projects, and when **clicked again**, it should hide the projects.

In order **to make an element clickable**, we need to **write jQuery that listens to an element for a click event**. jQuery can do this with an **event listener function named *on('click').***

This **function will wait for a click event**, and when one occurs, it will **execute a provided function.** The syntax looks like this:

$('.example-class').on('click', function() {

// execute the code here when .example-class is clicked.

});

1. **$('.example-class')**selects an HTML element with the class example-class.
2. **.on('click', function() { ... }) adds a click listener to the selector**. When it's clicked **the function will execute** the code within its block.

* **show**

To make our **projects visible** when the 'Recent Projects' button is clicked, **jQuery provides a function named *show****,* which is the **opposite of *hide.***

To show an element, the syntax looks as such:

$('.example-class').show();

1. show is **attached directly to the jQuery selector**.
2. show will change the CSS attribute ***display: none*to a visible display property**, therefore **showing the element.**

function main(){

$('.skillset').hide();

$('.skillset').fadeIn(1000);

$('.projects').hide();

$('.projects-button').on('click', function(){

$('.projects').show();

});

};

$(document).ready(main);

* **toggle**

When we click on a 'Recent Projects' button, the projects show. Next, let's hide the projects if we click the 'Recent Projects' button again.

jQuery provides a ***function named toggle***that is helpful in this situation. **toggle will hide or show an element, each time it is triggered.** The syntax looks like this:

$('example-class').toggle();

1. toggle can be **called directly on an jQuery selector.**
2. When toggle is executed, it will **hide or show the element that the selector points to.** If the element **is currently hidden**, **toggle will show the element,** and vice versa.

function main(){

$('.skillset').hide();

$('.skillset').fadeIn(1000);

$('.projects').hide();

$('.projects-button').on('click', function(){

$('.projects').toggle();

});

};

$(document).ready(main);

* **toggleClass**

Let's add one more feature: when **we click the 'Recent Projects' button** the **background color and text color should change.**

We can toggle a CSS class with a jQuery function named ***toggleClass.*** The syntax looks like this:

$('.example-class').toggleClass('active')

1. **.toggleClass**is a **function that will toggle a CSS class on the jQuery selector** it's **connected to.** If the element **has the class applied to it**, ***toggleClass*** will **remove it,** and if the element **does not have the class, it will add it.**
2. **'active'**is the **class** that we will **toggle on and off**. Notice that toggleClass does not require us to include the period before 'active' since it's already expecting a CSS class.

function main(){

$('.skillset').hide();

$('.skillset').fadeIn(1000);

$('.projects').hide();

$('.projects-button').on('click', function(){

$('.projects').toggle();

$('.projects-button').toggleClass('active');

});

};

$(document).ready(main);

* **this**

In the last exercise, we were **toggling every 'Recent Projects' button** instead of only the one we clicked on.

We **can select the specific element we clicked on** with the ***jQuery selector $(this).***

The syntax looks like this:

$('.example-class').on('click', function() {

$(this).toggleClass('active');

});

1. **$(this) selects the clicked element**. If there are **multiple elements with a class of .example-class,**this will ***only toggle the class of the one that is clicked on.***
2. Notice that $(this) does not require quotes around it, since it is not a CSS class. Instead, this is a JavaScript keyword.
3. **$(this) behaves just like our other selectors**. We can attach toggleClass or toggle to it in the same way.

function main(){

$('.skillset').hide();

$('.skillset').fadeIn(1000);

$('.projects').hide();

$('.projects-button').on('click', function(){

$('.projects').toggle();

$(this).toggleClass('active');

});

};

$(document).ready(main);

* **next**

In order to **toggle the projects in each section**, we will need to **use $(this) to select the button we clicked on**. The issue is that **$(this) refers to the projects-button**in our current code, and **not the projects themselves**.

We need a way to **select the projects elements next to the button that we clicked**.

Luckily, jQuery can **select elements logically**. In **index.html**, notice that the **projects-button element is directly followed by the projects list.** Therefore the **projects element is *next* after it.**

jQuery has a function named ***next*** to help us **select elements that are next to another** **element.** If we have this in our HTML:

<div class='item-one'> ... </div>

<div class='item-two'> ... </div>

If we wanted to hide item-two, we could write:

$('.item-one').next().hide();

1. We **can attach next**to **any jQuery selector** to select the **next direct element.**
2. Then, we can **attach any jQuery function to next().** In this case, we attached hide, which would hide the next element after the $('.item-one') element.

function main(){

$('.skillset').hide();

$('.skillset').fadeIn(1000);

$('.projects').hide();

$('.projects-button').on('click', function(){

$(this).next().toggle();

$(this).toggleClass('active');

});

};

$(document).ready(main);

* **text**

Since we have a few areas to click on, it may be helpful to show users **which sections they have viewed** by **changing the text inside the 'Recent Projects' buttons.**

When a **user clicks on a button**, we will **permanently change the text** of the button to 'Projects Viewed'.

We can **change the text** of an element with the **jQuery function text**. It's syntax looks like this:

$('.my-selector').text('Hello world!');

1. text **attaches directly to a jQuery selector.**
2. Inside of **text's parentheses**, we can **provide text that will become the text of our DOM element**. The text we supply will **replace any existing text**, and if the element has no pre-existing text, text will add it.

* **slideToggle**

The last feature we'd like to add is an aesthetic one. Right now when we click the 'Recent Projects' buttons, the **projects appear instantly.**

Let's instead **make the projects slide onto the page when we click the 'Recent Projects'** **button** and then **slide off the page when we click the button again**.

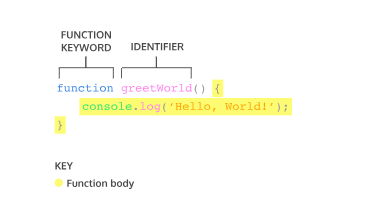
jQuery provides a **method named *slideToggle***that can **animate an element's entrance and exit.** The syntax looks like this:

$('.example-class').slideToggle(400);

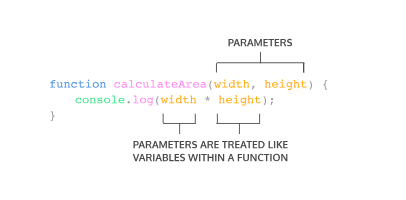
1. slideToggle can be **called directly on a jQuery selector**.
2. slideToggle also takes **a parameter of milliseconds** that the animation should last. The default is 400 milliseconds, or 0.4 seconds.

SUMMARY:

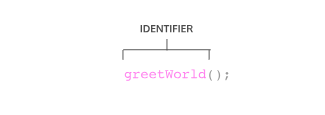
* Data is printed, or logged, to the console, a panel that displays messages, with console.log().
* You can write single-line comments with // and multi-line comments between /\* and \*/.
* There are 7 fundamental data types in JavaScript: strings, numbers, booleans, null, undefined, symbol, and object.
* Numbers are any number without quotes: 23.8879
* Strings are characters wrapped in single or double quotes: 'Sample String'
* The built-in arithmetic operators include +, -, \*, /, and %.
* Objects, including instances of data types, can have properties, stored information. The properties are denoted with a . after the name of the object, for example: 'Hello'.length.
* Objects, including instances of data types, can have methods which perform actions. Methods are called by appending the object or instance with a period, the method name, and parentheses. For example: 'hello'.toUpperCase().
* We can access properties and methods by using the ., dot operator.
* Built-in objects, including Math, are collections of methods and properties that JavaScript provides.
* Variables hold reusable data in a program and associate it with a name.
* Variables are stored in memory.
* The var keyword is used in pre-ES6 versions of JS.
* let is the preferred way to declare a variable when it can be reassigned, and const is the preferred way to declare a variable with a constant value.
* Variables that have not been initialized store the primitive data type undefined.
* Mathematical assignment operators make it easy to calculate a new value and assign it to the same variable.
* The + operator is used to concatenate strings including string values held in variables
* In ES6, template literals use backticks ` and ${} to interpolate values into a string.
* The typeof keyword returns the data type (as a string) of a value.
* An if statement checks a condition and will execute a task if that condition evaluates to true.
* if...else statements make binary decisions and execute different code blocks based on a provided condition.
* We can add more conditions using else if statements.
* Comparison operators, including <, >, <=, >=, ===, and !== can compare two values.
* The logical and operator, &&, or "and", checks if both provided expressions are truthy.
* The logical operator ||, or "or", checks if either provided expression is truthy.
* The bang operator, !, switches the truthiness and falsiness of a value.
* The ternary operator is shorthand to simplify concise if...else statements.
* A switch statement can be used to simplify the process of writing multiple else if statements. The break keyword stops the remaining cases from being checked and executed in a switch statement.
* A *function* is a reusable block of code that groups together a sequence of
* statements to perform a specific task.
* A *function declaration* :



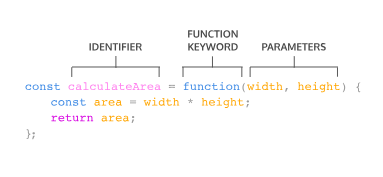
* A parameter is a named variable inside a function's block which will be assigned the value of the argument passed in when the function is invoked:



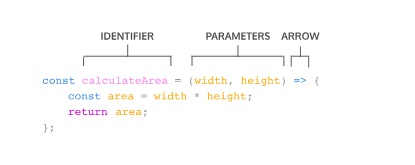
* To *call* a function in your code:



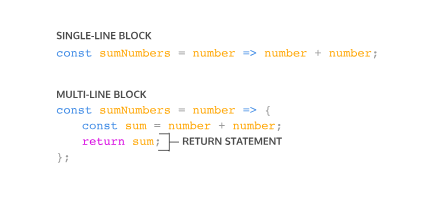
* ES6 introduces new ways of handling arbitrary parameters through *default parameters* which allow us to assign a default value to a parameter in case no argument is passed into the function.
* To return a value from a function, we use a *return statement*.
* To define a function using *function expressions*:



* To define a function using *arrow function notation*:



* Function definition can be made concise using concise arrow notation:



* It's good to be aware of the differences between function expressions, arrow functions, and function declarations. As you program more in JavaScript, you'll see a wide variety of how these function types are used.
* **Scope** is the idea in programming that some variables are accessible/inaccessible from other parts of the program.
* **Blocks** are statements that exist within curly braces {}.
* **Global scope** refers to the context within which variables are accessible to every part of the program.
* **Global variables** are variables that exist within global scope.
* **Block scope** refers to the context within which variables that are accessible only within the block they are defined.
* **Local variables** are variables that exist within block scope.
* **Global namespace** is the space in our code that contains globally scoped information.
* **Scope pollution** is when too many variables exist in a namespace or variable names are reused.
* Arrays are lists that store data in JavaScript.
* Arrays are created with brackets [].
* Each item inside of an array is at a numbered position, or index, starting at 0.
* We can access one item in an array using its index, with syntax like: myArray[0].
* We can also change an item in an array using its index, with syntax like myArray[0] = 'new string';
* Arrays have a length property, which allows you to see how many items are in an array.
* Arrays have their own methods, including .push() and .pop(), which add and remove items from an array, respectively.
* Arrays have many methods that perform different tasks, such as .slice() and .shift(), you can find documentation at the [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array" \t "_blank) website.
* Some built-in methods are mutating, meaning the method will change the array, while others are not mutating. You can always check the documentation.
* Variables that contain arrays can be declared with let or const. Even when declared with const, arrays are still mutable. However, a variable declared with const cannot be reassigned.
* Arrays mutated inside of a function will keep that change even outside the function.
* Arrays can be nested inside other arrays.
* To access elements in nested arrays chain indices using bracket notation.
* while loops allow for different types of stopping conditions
* Stopping conditions are crucial for avoiding infinite loops.
* do...while loops run code at least once— only checking the stopping condition after the first execution
* The break keyword allows programs to leave a loop during the execution of its block
* Abstraction allows us to write complicated code in a way that's easy to reuse, debug, and understand for human readers
* We can work with functions the same way we would any other type of data including reassigning them to new variables
* JavaScript functions are first-class objects, so they have properties and methods like any object
* Functions can be passed into other functions as parameters
* A higher-order function is a function that either accepts functions as parameters, returns a function, or both
* .forEach() is used to execute the same code on every element in an array but does not change the array and returns undefined.
* .map() executes the same code on every element in an array and returns a new array with the updated elements.
* .filter() checks every element in an array to see if it meets certain criteria and returns a new array with the elements that return truthy for the criteria.
* .findIndex() returns the index of the first element of an array which satisfies a condition in the callback function. It returns -1 if none of the elements in the array satisfies the condition.
* .reduce() iterates through an array and takes the values of the elements and returns a single value.
* All iterator methods takes a callback function that can be pre-defined, or a function expression, or an arrow function.
* You can visit the [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array" \t "_blank) to learn more about iterator methods (and all other parts of JavaScript!).
* Objects store collections of *key-value* pairs.
* Each key-value pair is a property—when a property is a function it is known as a method.
* An object literal is composed of comma-separated key-value pairs surrounded by curly braces.
* You can access, add or edit a property within an object by using dot notation or bracket notation.
* We can add methods to our object literals using key-value syntax with anonymous function expressions as values or by using the new ES6 method syntax.
* We can navigate complex, nested objects by chaining operators.
* Objects are mutable—we can change their properties even when they're declared with const.
* Objects are passed by reference— when we make changes to an object passed into a function, those changes are permanent.
* We can iterate through objects using the For...in syntax.
* The object that a method belongs to is called the *calling object*.
* The this keyword refers the calling object and can be used to access properties of the calling object.
* Methods do not automatically have access to other internal properties of the calling object.
* The value of this depends on where the thisis being accessed from.
* We cannot use arrow functions as methods if we want to access other internal properties.
* JavaScript objects do not have built-in privacy, rather there are conventions to follow to notify other developers about the intent of the code.
* The usage of an underscore before a property name means that the original developer did not intend for that property to be directly changed.
* Setters and getter methods allow for more detailed ways of accessing and assigning properties.
* Factory functions allow us to create object instances quickly and repeatedly.
* There are different ways to use object destructuring: one way is the property value shorthand and another is destructured assignment.
* As with any concept, it is a good skill to learn how to use the documentation with objects!
* You're ready to start leveraging more elegant code for creating and accessing objects in your code!
* *Classes* are templates for objects.
* Javascript calls a *constructor* method when we create a new instance of a class.
* *Inheritance* is when we create a parent class with properties and methods that we can extend to child classes.
* We use the extends keyword to create a subclass.
* The super keyword calls the constructor() of a parent class.
* Static methods are called on the class, but not on instances of the class.
* ES5 — The old JavaScript version that is supported by all modern web browsers.
* ES6 — The new(er) JavaScript version that is *not*supported by all modern web browsers. The syntax is more readable, similar to other programming languages, and addresses the source of common bugs in ES5.
* caniuse.com — a website you can use to look up HTML, CSS, and JavaScript browser compatibility information.
* Babel — A JavaScript package that transpiles JavaScript ES6+ code to ES5.
* npm init — A terminal command that creates a **package.json** file.
* **package.json** — A file that contains information about a JavaScript project.
* npm install — A command that installs Node packages.
* babel-cli — A Node package that contains command line tools for Babel.
* babel-preset-env — A Node package that contains ES6+ to ES5 syntax mapping information.
* **.babelrc** — A file that specifies the version of the JavaScript source code.
* "build" script — A **package.json** script that you use to tranpsile ES6+ code to ES5.
* npm run build — A command that runs the build script and transpiles ES6+ code to ES5.
* For future reference, here is a list of the steps needed to set up a project for transpilation:
* Initialize your project using npm init and create a directory called **src**
* Install babel dependencies by running
* npm install babel-cli -D npm install babel-preset-env -D
* Create a **.babelrc** file inside your project and add the following code inside it:
* { "presets": ["env"] }
* Add the following script to your scripts object in **package.json**:
* "build": "babel src -d lib"
* Run npm run build whenever you want to transpile your code from your **src** to **lib** directories.
* *Modules* in JavaScript are reusable pieces of code that can be exported from one program and imported for use in another program.
* module.exports exports the module for use in another program.
* require() imports the module for use in the current program.
* ES6 introduced a more flexible, easier syntax to export modules:
* default exports use export default to export JavaScript objects, functions, and primitive data types.
* named exports use the export keyword to export data in variables.
* named exports can be aliased with the askeyword.
* import is a keyword that imports any object, function, or data type.
* Promises are JavaScript objects that represent the eventual result of an asynchronous operation.
* Promises can be in one of three states: pending, resolved, or rejected.
* A promise is settled if it is either resolved or rejected.
* We construct a promise with by using the newkeyword and passing an executor function to the Promise constructor method.
* setTimeout() is a Node function which delays the execution of a callback function using the event-loop.
* We use .then() with a success handler callback containing the logic for what should happen if a promise resolves.
* We use .catch() with a failure handler callback containing the logic for what should happen if a promise rejects.
* Promise composition enables us to write complex, asynchronous code that's still readable. We do this by chaining multiple .then()'s and .catch()'s.
* To use promise composition correctly, we have to remember to return promises constructed within a .then().
* We should chain multiple promises rather than nesting them.
* To take advantage of concurrency, we can use Promise.all().
* async...await is syntactic sugar built on native JavaScript promises and generators.
* We declare an async function with the keyword async.
* Inside an async function we use the awaitoperator to pause execution of our function until an asynchronous action completes and the awaited promise is no longer pending .
* await returns the resolved value of the awaited promise.
* We can write multiple await statements to produce code that reads like synchronous code.
* We use try...catch statements within our async functions for error handling.
* We should still take advantage of concurrency by writing async functions that allow asynchronous actions to happen in concurrently whenever possible.
* You’ve done an amazing job navigating through making XHR GET and POST requests! Take some time to review the core concepts before moving on to the next lesson.
* JavaScript is the language of the web because of its asynchronous capabilities. AJAX, which stands for Asynchronous JavaScript and XML, is a set of tools that are used together to take advantage of JavaScript's asynchronous capabilities.
* There are many HTTP request methods, two of which are GET and POST.
* GET requests only request information from other sources.
* POST methods can introduce new information to other sources in addition to requesting it.
* GET requests can be written using an XMLHttpRequest object and vanilla JavaScript.
* POST requests can also be written using an XMLHttpRequest object and vanilla JavaScript.
* Writing GET and POST requests with XHR objects and vanilla JavaScript requires constructing the XHR object using new, setting the responseType, creating a function that will handle the response object, and opening and sending the request.
* To add a query string to a URL endpoint you can use ? and include a parameter.
* To provide additional parameters, use & and then include a key-value pair, joined by =.
* Determining how to correctly write the requests and how to properly implement them requires carefully reading the documentation of the API with which you're working.
* How to link a JavaScript file to an HTML file using a <script> tag.
* jQuery is a library to help JavaScript interact with HTML elements.
* We can make sure our page is ready to go with $(document).ready(). Then, we can pass in a function to ready that will execute when the page is loaded.
* jQuery uses the same selector names as CSS.
* We can hide elements with hide, and show them with show.
* We can make elements appear with fadeIn.
* on('click') functions allow us to make HTML elements clickable. When an element is clicked, the click function will execute the function we provide. It's full sytnax looks like:
* $('.example-class').on('click', function() { // Execute when .example-class is clicked });
* toggle will toggle an element on and off the page.
* $(this) will select the specific element that was clicked if placed inside a click function.
* toggleClass can toggle a class on and off.
* We can select elements next to each other with next.
* text will replace a DOM element's text with text we specify.
* slideToggle will make an element slide into and out of the page with an animation.