The ULTIMATE JavaScript Fundamentals Guide

Terms & Definitions

- \n
- + operator
- Addition operator
- alert()
- API
- API parameters
- API URL
- append()
- Arguments
- Arithmetic operators
- Arrays
- Assignment operator
- async keyword
- await keyword
- Block scope
- Boolean
- Boolean data type
- Bracket notation
- Callback function
- Calling
- CamelCase
- Change event
- classList property
- classList.add()
- classList.remove
- Comparison operators
- Compound assignment operators
- Concatenating
- Conditional statement
- Console
- console.log()
- const keyword
- const vs. let vs. var
- Context
- createElement()
- Data type
- Date()
- Debugging
- Decomposition
- defer attribute
- Delimiter
- disabled property
- DOM
- DOM events
- DOM tree
- Dot notation
- Elements
- else if keyword
- else keyword
- Endpoints
- Event handler
- Event listener
- Expressions
- Factory functions
- fetch()
- Floating point number
- for loop
- for...of loop
- for...in loop
- forEach()
- Function
- Function body
- Function expression
- Function scope
- getHours()
- Global scope
- if keyword
- Index
- innerHTML property
- innerText property
- Input event
- Integer
- Iterating
- JSON files
- json()
- Keydown event
- Keys
- Key-value pair
- let keyword
- Loop
- Loop body
- match()
- matches()
- Math.floor()
- Math.max()
- Math.min()
- Math.random()
- Method (Lesson 4)
- Method (Lesson 10)
- Modal window
- Modulus (%) operator
- Mouse events
- Nested if statement
- Null
- Number()
- Object literal
- Objects
- Parameters
- Primitive data types
- prompt()
- Properties
- querySelector()
- querySelectorAll
- Refactored code
- Regular expression
- Reserved keywords
- REST APIs
- return keyword
- Scope
- Statement
- String
- style property
- Template literals
- this keyword
- toFixed()
- toLowerCase()
- toUpperCase()
- trim()
- Type conversion
- Undefined
- value property
- Values
- Variable

Lesson 1 - Getting Started with JavaScript

Variable - A tool for pointing towards information. The associated value can vary or change. Variables can be declared using the keyword `var` (Lesson 1) and the keywords `let` and `const` (Lesson 9).

Values - Information stored in a variable. Specifically, a value is a sequence of bits that is interpreted according to some data type (e.g., number, string, boolean).

Assignment operator - An operator that assigns a value to a variable. The assignment operator uses the equal sign (=). See the Javascript Operators Cheatsheet (Lesson 3) for a list of assignment operators.

Statement - A single instruction to the program. Often, semicolons appear at the end of a statement to show it's complete.

```
var cerealTypes = 16;
```

```
console.log("We love JS!");
```

String - A series of characters, like numbers, letters, and symbols. Strings will have quotes around them to group the characters and keep them in a sequence.

```
var vacationSpot = "beach";
```

```
var phoneNumber = "555-555-1234";
```

CamelCase - The standard naming convention for variables in JavaScript. The first words are all lowercase letters, while each proceeding word begins with an uppercase letter. Examples: `bankDeposit`, `userInputDate`, and `ageLimit18`

Console - An environment in your browser where you can execute, or run, JavaScript. The console lets you see the output of your code and troubleshoot errors. In CodeSandbox, your console is located under the "Console" tab. In Google Chrome, go to More Tools > Developer Tools > Console tab.

console.log() - A method to log out a message to the console.

```
console.log("Party time! Excellent!);
// Party time! Excellent!
```

```
var cats = 4;
console.log("I have" + cats + "cats.");
// I have 4 cats.
```

+ operator - An operator that uses the plus sign (+) to combine strings and variables.

```
var name = "Giorno Giovanna";
console.log("His name is" + name + ".");
// His name is Giorno Giovanna.
```

Concatenating - The process of joining strings together using the + operator.

```
var ringMetal = "gold";
console.log("She gave her a" + ringMetal + "ring.");
// She gave her a gold ring.
```

Back to Top ↑

Lesson 2 - Data Types & Arithmetic Operators

Template literals - Output strings using placeholders and backticks (`). Compared to outputting strings with single or double quotes and the plus operator, template literals

make it easier to output multi-line strings and combine variables with strings. In addition, you can calculate expressions inside the string.

```
var jewelry = "watch";
var event = "dinner";

console.log(`They wore a ${jewelry} to ${event}.`);
// They wore a watch to dinner.
```

```
var pizzaType = "veggie";
var slicesEaten = 4;
console.log(`The ${pizzaType} pizza has ${8 - slicesEaten} slices left.`);
// The veggie pizza has 4 slices left.
```

Expressions - Code that results in a value. For example, expressions can result in numeric, string, and logical values (Lesson 3).

```
console.log(8 - 5);
// 3
```

```
console.log("I love" + " coding.");
// I love coding.
```

```
console.log(5<8);
// true
```

Integer - A whole number, like 5100 or -258. Integers can be positive or negative.

Floating point number - A number with a decimal, like 2134.3625 or -562.12. Floating point numbers can be positive or negative.

Addition operator - An operator to add two numbers together. The addition operator

uses the plus sign (+).

```
var applesBananas = 5 + 8;
console.log(applesBananas);
// 13
```

```
var floor1 = 10;
var floor2 = 15;
console.log(`There are ${floor1 + floor2} tables in the restaurant.`);
// There are 25 tables in the restaurant.
```

Arithmetic operators - Symbols for math operations, like the addition (+), subtraction (-), multiplication (*), and division (/) operators. See the JavaScript Operators Cheatsheet (Lesson 3) for a complete list of arithmetic operators.

Data type - The type of value a variable points to. Examples include numbers, strings, booleans (Lesson 3), undefined, null, arrays (Lesson 8), and objects (Lesson 10).

Primitive data types - Values with only a single value, like numbers, strings, booleans (Lesson 3), undefined, and null.

Undefined - A variable with no value assigned to it.

```
var happiness;

console.log(happiness);
// undefined
```

Null - A data type that represents an intentionally empty, or non-existent, value.

```
var ideas = null;
```

```
console.log(ideas);
// null
```

Type conversion - Changing one value to a different value to complete an operator. Type conversion is beneficial for changing strings into numbers so you can calculate them.

Number() - Convert a string into a number. Number() is useful when gathering input from a user and then changing it to a number so that you can calculate a value.

```
var tvShows = Number("23");
var movies = 12;
console.log(tvShows + movies);
// 35
```

prompt() - Displays a field to gather information from the user. Users will see a pop-up dialog box on their screen asking for input.

```
var favoriteGenre = prompt("What's your favorite music genre?");
console.log(favoriteGenre);
```

```
var oldFunds = 1500;
var newFunds = Number(prompt("How much funds were raised?"));
console.log(
`The fundraiser total is now $${oldFunds + newFunds}!.`
);
```

toFixed() - Convert a number data type into a string and then round to a specified number of decimal places. Add a number inside toFixed() to specify the number of decimal places to round to.

```
var taxAmount = 7.23335651;
console.log(taxAmount.toFixed(2));
// 7.23
```

```
var tempFahrenheit = 98.6785;
console.log(`Her temperature is ${tempFahrenheit.toFixed(1)}.`);
// Her temperature is 98.7.
```

```
var people = 27;
var payout = 800.29;
console.log(`You won $${(payout / people).toFixed(2)}.`);
// You won $29.64.
```

Back to Top ↑

Lesson 3 - Comparisons & Conditionals

Conditional statement - Code that will only run if a condition is true.

Boolean - Represent just two values: true or false.

Boolean data type - A primitive data type with true or false values.

```
var lightsOn = true;
var fanOn = false;
console.log(lightsOn);
//true
```

Comparison operators - Operators that use symbols to compare two or more values, like >, <, and ===. See the JavaScript Operators Cheatsheet (Lesson 3) for a complete list of comparison operators.

if keyword - Keyword to use in a statement to test a condition. If the condition evaluates to true, then the program runs the code inside the if block. You won't include a semicolon

after the condition.

```
var hotWeather = true;

if (hotWeather === true) {
  console.log("Wear shorts and a tank top today!");
}
// Wear shorts and a tank top today!
```

else keyword - Keyword to use in a statement to perform another action if the previous condition evaluates to false.

```
var hotWeather = false;

if (hotWeather === true) {
  console.log("Wear shorts and a tank top today!");
} else {
console.log("Grab a sweater, it might be chilly.");
}
// Grab a sweater, it might be chilly.
```

else if keyword - Keyword to use in a statement to test a new condition, and then perform an action if the previous condition evaluates to false. As soon as a condition evaluates to true, the code block that the condition is associated with runs and the conditional block is exited, regardless if there are subsequent conditions that would also evaluate to true.

```
var hotWeather = false;
var snowyWeather = true;
var windyWeather = true;

if (hotWeather === true) {
  console.log("Wear shorts and a tank top today!");
} else if (snowyWeather === true)  {
console.log("Put on a heavy jacket and boots!");
} else if (windyWeather === true)  {
```

```
console.log("Time to slip on your windbreaker.");
} else {
console.log("Grab a sweater, it might be chilly.");
}
// Put on a heavy jacket and boots!
```

alert() - Displays a pop-up message for users to see. The prompt includes an OK button for users to click and close the pop-up.

```
alert("Hello, welcome to my site!");
```

Date() - A method to retrieve the current date.

```
var weekday = new Date().toLocaleString("en-US", { weekday: "long" });
```

getHours() - A method to retrieve the current time. The time will reflect the 24-hour clock, AKA military time.

```
var time = new Date().getHours();
```

Back to Top ↑

Lesson 4 - JS, HTML, & CSS

---

defer attribute - Instructs the browser to load the script after the page has loaded. The attribute creates a faster loading experience for the user because all the HTML renders first, even if the JavaScript hasn't run yet. It also makes sure the HTML elements are loaded so the JavaScript can modify them. You'll add the <script> tag and defer attributes in the head section of the HTML page.

```
<!DOCTYPE html>
```

```
<html>
<head>
<script src="js/script.js" defer></script>
</head>
```

DOM - Short for Document Object Model, the DOM represents the structure and content of a web page. The document is the web page. The objects include HTML elements, text, and attributes.

DOM tree - A graphical representation of the DOM which shows relationships between objects. The DOM tree is useful for determining how to access different objects on the document.

Methods - JavaScript actions performed on objects. Examples of methods include console.log(), prompt(), alert(), , and querySelector(). Methods are also a type of object property (Lesson 10).

querySelector() - A method to access the first element that matches a specified selector. To select multiple items, you'll need to use an array (Lesson 8) with querySelectorAll() (Lesson 9).

```
var available = document.querySelector("h3");
var mainTitle = document.querySelector(".main-title");
var firstItem = document.querySelector("ul li");
var intro = document.querySelector(".intro p");

console.log(available, mainTitle, firstItem, intro);
// <h3>We're here for you every day of the week.</h3>
// <h1 class="main-title">Ryan's Roses</h1>
// <li>Today's Specials</li>
// <p>Available today</p>
```

```
var firstImg = document.querySelector("img");
firstImg.src = "img/lulu.jpeg";
```

```
firstImg.alt = "Lulu bouquet";

console.log(firstImg);
// <img src="img/lulu.jpeg" alt="Lulu bouquet"></img>
```

style property - A property that allows you to change the style of an element. If the property name is two words, like background-color, change it to one word using camelCase (backgroundColor).

```
var intro = document.querySelector(".intro p");

intro.style.color = "purple";
intro.style.fontSize = "3em";
intro.style.fontStyle = "italic";

console.log(intro);
// <p style="color: purple; font-size: 3em; font-style:
italic;">Available today</p>
```

innerText property - A property that accesses the text within an element. This property is useful when you want to change or retrieve the text inside an element.

```
var firstCaption = document.querySelector("figcaption");
firstCaption.innerText = "The Lulu.";

console.log(firstCaption);
// <figcaption>The Lulu.</figcaption>
```

innerHTML property - A property that changes the HTML of an element on the page. This property is useful for updating or adding elements to a page.

```
firstCaption.innerHTML = "The <strong>Lulu</strong>";

console.log(firstCaption);
//<figcaption>The<strong>Lulu</strong></figcaption>
```

```
var intro = document.querySelector(".intro p");

intro.innerHTML = 'Available <span
class="increase__size">today</strong>';

console.log(intro);
// <p>Available<span class="increase__size">today</span></p>
```

Debugging - Identifying and removing errors in your code.

Back to Top ↑

Lesson 5 - Events & Event Listeners

DOM events - Actions that happen in the document (web page). Events can be triggered by the browser or by the user. In this class, you'll use mouse, change, keydown, and input events. See Mozilla's Event Reference page for a complete list of events.

Mouse events - An event that happens when a pointing device, like a mouse, joysticks, keyboard, or adaptive switch interacts with the document. Common mouse events are `"click"`, `"mouseover"`, and `"select"`.

Event listener - A method that "listens" for events to happen and then takes action. Use the method `addEventListener()` to listen for events in the DOM.

```
var title = document.querySelector("h1");

title.addEventListener("mouseover");
```

Event handler - A function that runs code when an event occurs.

```
var title = document.querySelector("h1");
```

```
title.addEventListener("mouseover", function () {
      title.innerText = "Let's PARTY!";
      title.style.color = "maroon";
});
```

Function - A block of code that can be called or invoked to run as many times as needed without repeating code. Functions are vital to writing streamlined JavaScript. Lesson 6 contains a full dive into functions.

Function body - The part of the function that contains the statements that specify what the function does. Curly braces surround the function body.

classList property - A property to add, remove, or toggle CSS classes on an element. This property let's you apply (or remove) multiple styles at once. You can use the classList property with the add() and remove() methods: `classList.add()` and `classList.remove()`.

classlist.add() - A method to add a new class.

```
var darkModeButton = document.querySelector(".dark-mode");
var body = document.querySelector("body");

darkModeButton.addEventListener("click", function () {
body.classList.add("dark-palette");
});
```

classlist.remove() - A method to remove a new class.

```
var lightModeButton = document.querySelector(".light-mode");

lightModeButton.addEventListener("click", function () {
```

```
body.classList.remove("dark-palette");
});
```

Modal window - A web page element that overlays a box in front of a web page. A modal is also called a lightbox.

Back to Top ↑

Lesson 6 - Functions

Function expression - A syntax for writing functions that begins with a variable name and then uses the `function` keyword to define the function.

```
var welcome = function () {
console.log();
};
```

Reserved keywords - A word that can't be used as a variable name in JavaScript. See a complete list of reserved keywords.

Parameters - Placeholders for values you want to pass to the function. If there's more than one parameter, separate the parameters with a comma.

```
var welcome = function (name) {
  console.log(`Welcome, ${name}. Have a great day!`);
}
```

Calling - An action which will cause a function to run. If the function expects arguments, you must provide them in the function call.

```
var welcome = function (name) {
```

```
  console.log(`Welcome, ${name}. Have a great day!`);
}

welcome("Sadie");
// Sadie
```

Arguments - Values passed to the function when it's called. If there's more than one argument, separate the arguments with a comma.

```
var addTogether = function (num1, num2) {
  console.log(num1 + num2);
};

addTogether(13, 72); // 85
addTogether(36, -2.88); // 33.12
```

return keyword - A keyword to return the value of a function and end its execution. Use the `return` keyword to make the result of a function available to other parts of your code. Unlike `console.log()` which only outputs a message to the console, the `return` keyword allows a value to be used by other parts of the code, including `console.log()`.

```
var addTogether = function (num1, num2) {
  return num1 + num2;
};

alert(addTogether(36, -2.88));
console.log(addTogether(13, 72)); // 85

var lunchForTwo = addTogether(24.56, 18.99);
console.log(lunchForTwo); // 43.55
```

Lesson 7 - Keydown & Change Events

Callback function - A function that's passed to another function as an argument. For example, an event handler is a callback function.

```
button.addEventListener("click", function () {
  cat.classList.add("show");
});
```

Keydown event - An event that occurs when a key is pressed on a keyboard, like a letter, number, or Enter key. Inside the callback function for a keydown event, you'll pass a parameter that will hold the event object. Most coders use `e` as the parameter to represent "event."

```
var body = document.querySelector("body");

document.addEventListener("keydown", function (e) {
  // console.log(e);
  if (e.key === "l") {
    body.classList.add("light");
  }
});
```

Nested if statement - An if statement testing the condition of another if, else if, or else statement.

In this example, the second if statement (`if (body.classList.contains("light"))`) is the nested if statement.

```
var body = document.querySelector("body");

document.addEventListener("keydown", function (e) {
 // console.log(e);
 if (e.key === "l") {
   body.classList.add("light");
 } else if (e.key === "d") {
   if (body.classList.contains("light")) {
     body.classList.remove("light");
   }
 }
```

```
});
```

Change event - An event that occurs when the user changes a drop-down list (i.e., the <select> element) or input areas like the <input> or <textarea> elements. Inside the callback function for a change event, you'll pass a parameter that will represent the change event. Most coders use `e` as the parameter to represent "event."

```
var fave = document.querySelector("#favorite");
var heading = document.querySelector("h1");
var selection = "regular";

fave.addEventListener("change", function (e) {
 selection = e.target.value;
 if (selection === "stealth") {
   heading.innerText = "Stealth Quincy 😎";
 } else if (selection === "party") {
   heading.innerText = "Party Quincy 🥳";
 } else {
   heading.innerText = "Quincy";
 }
});
```

toUpperCase() - A method for converting a string value into all uppercase letters.

```
var louder = "Speak up, please!"

console.log(louder.toUpperCase());
// SPEAK UP, PLEASE!
```

Math.floor() - A method for rounding a number down to the next whole number.

```
var seatingCapacity = 1256.3;

console.log(Math.floor(seatingCapacity));
//1256
```

Math.random() - A method for producing a random number between 0 and 1. Multiply it by another number to output a larger random number. Pair it with `Math.floor()` to round

the number to the nearest whole number.

```
console.log(Math.random());
// 0.15884857919099582
```

```
console.log(Math.random() * 36);
// 18.873096475917126
```

```
console.log(Math.floor(Math.random() * 12));
// 10
```

Back to Top ↑

Lesson 8 - Arrays & Loops

Arrays - A data type that contains one or more values. You'll add square brackets around the array values (elements). See the Javascript Arrays Cheatsheet (Lesson 8) for a list of array methods.

```
var timeOfDay = [6, "noon", 8, "morning", "evening", 12];
```

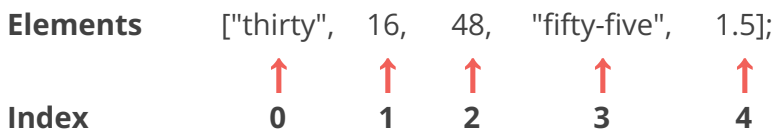
You can also create an empty array to add items.

```
var medicine = [];
```

Elements - The values stored in an array. Elements can be strings, numbers, and floating point numbers data types.

```
var ages = ["thirty", 16, 48, "fifty-five", 1.5];
```

Index - The position of an element in an array. In JavaScript, the first element starts at index 0. The second element would start at index 1, and so on.



Loop - A statement that allows you to repeat code multiple times.

Iterating - Each time a loop runs through a block of code. Each pass of the loop is called an iteration.

for loop - A type of loop that iterates through a block of code a designated number of times. Examples of for loops include the for...of loop (Lesson 8) and the for...in loop (Lesson 11).

for...of loop - A type of for loop that iterates over the values of an array. A `for...of` loop only has access to the values of an array, not the index. You can use the `for...of` loop with the for...in loop (Lesson 11) to loop through multiple object properties.

```
var timeOfDay = [6, "noon", 8, "morning", "evening", 12];

for (var time of timeOfDay) {
  console.log(`It is ${time}.`);
}
// It is 6.
// It is noon.
// It is 8.
// It is morning.
// It is evening.
// It is 12.
```

Loop body - The loop section where you'll write the statements you want to execute on each loop iteration.

forEach() - A method to iterate through elements in an array and then execute a function for each array item. Unlike `for...of` loops, forEach() lets you access the array elements' value and index.

```
var timeOfDay = [6, "noon", 8, "morning", "evening", 12];

timeOfDay.forEach(function (time, index) {
  console.log(`The ${time} element is at position ${index}.`);
});
// The 6 element is at position 0.
// The noon element is at position 1.
// The 8 element is at position 2.
// The morning element is at position 3.
// The evening element is at position 4.
// The 12 element is at position 5.
```

Modulus operator - An operator to return the remainder of two numbers divided. The modulus operator uses the percent sign (%). The modulus operator is also called the "modulo" operator.

```
var candy = 14;
var kids = 4;

console.log(`There are ${candy % kids} pieces of candy remaining.`);
// There are 2 pieces of candy remaining.
```

```
var num = 45;

if (num % 2 === 0) {
  console.log("This is an even number.");
} else {
  console.log("This is an odd number.");
};
// This is an odd number.
```

Back to Top ↑

Lesson 9 - Scope

Scope - The context where variables are visible to certain parts of your program. Scope can be divided into global scope, function scope, and block scope.

Context - The place the code is evaluated and executed, like inside a function or loop.

Global scope - The context for the whole program. Globally scoped variables are available to any part of the program.

Function scope - The context inside a function. Variables defined within a function are scoped only to that function or nested functions.

Block scope - The context inside a block of code. Unlike declaring variables with var, declaring your variables with let and const keeps variables in block scope.

let keyword - A keyword to declare variables and prevent them from being accessed outside the block they were declared in. Use `let` inside code blocks (e.g., loops, if/else if statements) and when you want to reassign the value of a variable.

```
if (numOfDrinks === 5) {
  let soda = "lemon-lime";
  console.log(soda);
}
// lemon-lime

console.log(soda);
// ReferenceError: soda is not defined
```

const keyword - A keyword to declare variables to constrain a variable to block scope and prevent the value from being reassigned. Using `const` will prevent data types like

strings, booleans, and numbers from being reassigned to a different value. For data types like arrays and objects, `const` will prevent reassigning the variable but still allow you to modify the elements inside the array/object.

```
const numOfDrinks = 5;

const drinks = function () {
  const tea = 6 + numOfDrinks;
  console.log(tea);
};

drinks();
// 11
```

If you try to reassign a variable declared with const, you'll receive an error in the console like “TypeError: Assignment to constant variable” or "<variable name> is read-only" when attempting to reassign a variable.

```
const numOfDrinks = 5;
numOfDrinks = 7;

console.log(numOfDrinks);
// SyntaxError: /script.js: "numOfDrinks" is read-only
```

const vs. let vs. var - For most uses, you'll want to use const to declare your variables, except when you need to reassign variables (`let`) or you're working with legacy code (`var`).

	const	let	var
Function scoped	Yes	Yes	Yes
Block scoped	Yes	Yes	No
Reassignable	No	Yes	Yes
Redeclareable	No	No	Yes
Summary	Use `const` as the default way to declare variables.	Declare variables with `let` when you need to reassign the	Use `var` when working with legacy code that already

	You'll use `const` 95% of the time.	value of your variables, like in a loop or if/else if statement.	uses var. Declaring with `var` is also helpful when learning to write code and scope issues aren't a factor.

value property - A property to capture the content entered into a text box.

createElement() - A method to create a new HTML element.

append() - A method to add elements at the end of another DOM element, like a list.

querySelectorAll() - A method to select all the elements that match a specific selector. The querySelectorAll() returns a list of elements in an array-like structure..

```
addShowButton.addEventListener("click", function () {
  const show = showInput.value;
  if (show !== "") {
    let listItem = document.createElement("li");
    listItem.innerText = show;
    showList.append(listItem);
    let shows = document.querySelectorAll(".show-list li");
    showCount.innerText = shows.length;
  }
});
```

length property - A property to identify the number of elements in an array.

```
var daysOfWeek = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];
console.log(daysOfWeek.length);
// 5
```

Refactored code - Code that was restructured without changing or adding to its functionality, usually with the goal to make it more readable, better performing, or both.

disabled property - A property to indicate if an element can be interacted with or not. The `disabled` property uses the boolean values of true and false. For example, if the `disabled` property is set to `true` for a button, the user can no longer click the button.

```
assignButton.addEventListener("click", function () {
  assignItems();
  assignButton.disabled = true;
});
```

Math.min() - A method that finds the smallest value passed to it. Use the spread (...) operator to send individual array elements to Math.min() instead of the whole array.

```
console.log(Math.min(2, -12, 71));
// 12
```

```
const employees = [12, 68, 333, 56, 1250];
const smallestNum = Math.min(...employees);

console.log(smallestNum);
// 12
```

Math.max() - A method that finds the largest value in an array. Use the spread (...) operator to send individual array elements to Math.max() instead of the whole array.

```
console.log(Math.max(2, -12, 71));
// 71
```

```
const employees = [12, 68, 333, 56, 1250];
const largestNum = Math.max(...employees);

console.log(largestNum);
// 1280
```

Back to Top ↑

Lesson 10 - Objects & Methods

Objects - A data type used to group multiple properties and their corresponding values into a single, unordered entity. Like an object in real life, a JavaScript object represents a thing with characteristics (properties), like a person, animal, instrument, or house. An object is a collection of key-value pairs.

Properties - Different values of an object. A property represents the different characteristics of your object. Properties can be any data type, like a number, string, array, boolean, or function.

Dot notation - A syntax to assign or access the property of an object using a period between the object name and property.

```
const cat = {};

cat.name = "Fred";
cat.nickname = "Flooficus";
cat.age = 5;
cat.isSleeping = true;
cat.favoriteToys = ["spring", "ping pong balls", "bird stuffy"]
cat.pet = function () {
  return "purrrrrrrr";
}
```

Method (object property) - A function that's a property in an object. Use the `return` keyword to return the value of the method and make the result available to other parts of your code.

Methods can be added to an object following its creation:

```
const cat = {};

cat.pet = function () {
  return "purrrrrrrr";
}
```

Methods can also be created inside an object literal:

```
const cat = {
  pet: function () {
    return "purrrrrrrr";
  }
};
```

Keys - An object's unique elements which are used to access its values. Keys are also known as "identifiers" or "names." An object's keys must be unique and cannot be duplicated in the same object.

Key-value pair - An object's property consisting of a key and its associated value.

Dot Notation

key-value pair

```
house.color = "blue"
```

object **key** **value**

Bracket Notation

key-value pair

```
house["color"] = "blue"
```

object **key** **value**

Object Literal

```
const house = {      ← object
  color: "blue"      ← key-value pair
};
```

Factory Function (w/parameter)

```
const createHouse = function (color) {
  const house = {    ← object
    color: color     ← key-value pair
  };
```

```
    return house;
};
```

Bracket notation - A syntax to access or assign the property of an object using square brackets around the between property. Add quotation marks around the property name inside the square brackets.

```
const cat = {
  name: "Fred",
  nickname: "Flooficus",
  age: 5,
  isSleeping: true,
  favoriteToys: ["spring", "ping pong balls", "bird stuffy"],
  pet: function () {
    return "purrrrrrrr";
  }
};

cat["color"] = "orange";

console.log(cat["isSleeping"]);
// true
```

Object literal - A collection of key-value pairs inside the object's curly braces, separated by a comma. The key and value are separated by a colon (:). You can add or change existing properties of an object literal by using either dot or bracket notation and the `=` assignment operator.

```
const cat = {
  name: "Fred",
  nickname: "Flooficus",
  age: 5,
  isSleeping: true,
  favoriteToys: ["spring", "ping pong balls", "bird stuffy"],
  pet: function () {
    return "purrrrrrrr";
  }
};
```

```
cat.isSleeping = false;
cat["color"] = "orange";

console.log(cat);
// {name: "Fred", nickname: "Flooficus", age: 5, isSleeping: false,
favoriteToys: Array(3)...}
```

this keyword - In a method, the `this` keyword allows you to reference another property from the same object.

An example of the `this` keyword used with a method that's outside the object declaration:

```
const house = {
  windows: 20
};

house.windowWash = function () {
  if (this.windows > 15) {
    return `That's a lot of windows to wash!`;
  }
};

console.log(house.windowWash());
//That's a lot of windows to wash!
```

Here's an example of `this` keyword used with a method that's declared in an object literal:

```
const house = {
  windows: 20,
  windowWash: function () {
    if (this.windows > 15);
    return `That's a lot of windows to wash!`;
  }
};

console.log(house.windowWash());
//That's a lot of windows to wash!
```

Compound assignment operators - An assignment operator that combines the

assignment operator (=) with an arithmetic operator (+, -, *, /, and %). Compound assignment operators provide a shorter, cleaner syntax for performing calculations. See the JavaScript Operators Cheatsheet (Lesson 3) for a full list of assignment operators.

```
let paperclips = 10;
paperclips += 2;
console.log(paperclips);
// 12
```

```
let candy = 15;
candy %= 6;
console.log(`There's ${candy} candies leftover.`);
// There's 3 candies leftover.
```

Back to Top ↑

Lesson 11 - Factory Functions

Factory functions - Patterns to create multiple objects. Factory functions let you quickly build several objects that share the same characteristics, AKA properties. You must return your object at the bottom of your factory function. You'll use factory functions when you want to create and manage multiple objects that have the same characteristics (e.g., color) that are expressed differently (e.g., blue, green, yellow).

```
const createContact = function () {
  const contact = {
    name: "Noelle Silva",
    phoneNum: "555-555-1234",
    isNew: true,
    message: function () {
      this.isNew = true;
      console.log("You've added a new contact!");
    }
  };
  return contact;
};
```

```
console.log(createContact());
// {name: "Noelle Silva", phoneNum: "555-555-1234", isNew: true, message: f
message()}
```

You can provide parameters to your factory function in order to make your object more flexible and easy to reuse:

```
const createContact = function (name, phone) {
  const contact = {
    name: name,
    phoneNum: phone,
    isNew: true,
    message: function () {
      this.isNew = true;
      console.log("You've added a new contact!");
    }
  };
  return contact;
};

const contact1 = createContact("Noelle Silva", "555-555-1234");
const contact2 = createContact("Yami Sukehiro", "555-321-5555");

console.log(contact1, contact2);
// {name: "Noelle Silva", phoneNum: "555-555-1234", isNew: true, message: f
message()}
// {name: "Yami Sukehiro", phoneNum: "555-321-5555", isNew: true, message:
f message()}
```

for...in loop - A type of `for` loop that will allow you to loop over an object's key-value pairs. When looping over objects, you may want to access just the keys, just the values, or both the keys and the values.

```
const createContact = function (name, phone) {
  const contact = {
    name: name,
    phoneNum: phone,
    isNew: true,
    message: function () {
```

```
      this.isNew = true;
      console.log("You've added a new contact!");
    }
  };
  return contact;
};

const contact1 = createContact("Noelle Silva", "555-555-1234");

for (let key in contact1) {
  console.log(key, contact1[key]);
}
// name  Noelle Silva
// phoneNum  555-555-1234
// isNew true
// message ƒ message() {}
```

To loop through multiple objects, add the objects to an array and then loop through the array using the for...of loop (Lesson 8). After the `for...of` loop, nest the `for...in` loop to access the object's key, value, or keys and values.

```
const contact1 = createContact("Noelle Silva", "555-555-1234");
const contact2 = createContact("Yami Sukehiro", "555-321-5555");

const contactsArray = [contact1, contact2];

for (let contact of contactsArray) {
  for (let key in contact) {
    console.log(key, contact[key]);
  }
}
// name  Noelle Silva
// phoneNum  555-555-1234
// isNew true
// message ƒ message() {}
// name  Yami Sukehiro
// phoneNum  555-321-5555
// isNew true
// message ƒ message() {}
```

Back to Top ↑

Lesson 12 - Intro to APIs

API - An Application Programming Interface (API) is a way to allow information from an internal or external source to interact with your program.

API URL - The address to get access to the API. The API developers determine the API URL and associated endpoints and parameters, which can be found in the API's documentation. To access specific data from the API, you'll need the API URL combined with endpoints and possibly parameters.

Example API URLs:

- https://quote-garden.herokuapp.com/api/v3/
- https://api.tvmaze.com/

JSON files - A type of text file used for exchanging data. Most programming languages can interpret JSON files. JSON stands for JavaScript Object Notation. JSON files end with a .json file extension. Install an extension on your browser, JSON Formatter, to reformat JSON data and make it easier to read.

REST APIs - A type of API for making use of HTTP requests. You'll use the REST API's documentation to discover the API URL, endpoints, and parameters.

Endpoints - The "end" of the API URL that determines the type of information available.

- Example API endpoint for all quotes: https://quote-garden.herokuapp.com/api/v3/**quotes**

- Example API endpoint for a subset of quotes: https://quote-garden.herokuapp.com/api/v3/**quotes/random**

API parameters - Placeholders for data in the API URL. You'll add a question mark (?) between the endpoint and the parameters. If the parameter has more than one word,

replace the space between the words with the "%20" character. To chain more than one parameter together, add the ampersand (&) sign between the parameters.

Example API URL with a single parameter:
https://quote-garden.herokuapp.com/api/v3/quotes?**author=maya%20angelou**

Example API URL with multiple parameters (separated by a "&" sign):
https://quote-garden.herokuapp.com/api/v3/quotes?**author=maya%20angelou&limit=1**

fetch() - A method to allow you to get resources over a network, like data from an API.

async keyword - A keyword to enable asynchronous communication between your program and the API.

await keyword - A keyword that tells the program to wait on that line in the function until the API data are received.

json() - A method to parse (interpret) the JSON data from the API call and transform it into a JavaScript object.

```
const getData = async function () {
  const res = await fetch(
    "https://quote-garden.herokuapp.com/api/v3/quotes?author=beyonce"
  );
  const data = await res.json();
  console.log(data);
};
getData();
```

```
const getShows = async function () {
  const showRequest = await fetch("https://api.tvmaze.com/schedule/web");
  const data = await showRequest.json();
  console.log(data);
};
getShows()
```

```
// {61) [Object, Object, Object, Object, Object, Object, Object, Object,
Object, Object, ...]
```

Back to Top ↑

Lesson 13 - Project: Guess the Word Game

Decomposition - A computer science term that means breaking down a larger problem into smaller problems. Decomposition makes tackling a large project easier by breaking it into smaller problems that need to be solved.

Regular expression - A sequence of characters that lets you find text that matches a specific pattern. You'll use a regular expression when searching or replacing text. Regular expressions are also called "regex" or "regexp" for short. To learn more about regular expressions, check out this JavaScript Regex article.

match() - A method used with a regular expression to search the strings and match them to the regular expression.

```
const str = 'CanyoufindthesecretchocolatesnackIhaveinthislongstring';
const snackMatch = str.match(/chocolate/);

if (snackMatch) {
  console.log("Found the chocolate!")
};
// Found the chocolate!
```

Delimiter - A character to separate words in a string.

\n - A delimiter to create a line break (AKA newline).

```
console.log("First,\nsecond,\nand third!");
// First
// second,
// and third!
```

trim() - A method to remove extra whitespace before and after a string.

```
var happiness = "    Happiness is bug-free code.     ";

console.log(happiness);
//     Happiness is bug-free code.

console.log(happiness.trim());
// Happiness is bug-free code.
```

Back to Top ↑

Lesson 14 - Projects: GitHub Repo Gallery

---

matches() - A method to check if the target element (i.e., where the user clicks on the page) matches a specific selector.

```
const h2 = document.querySelectorAll('h2');
for (let heading of h2){
    if (heading.matches(".highlight")){
        heading.style.backgroundColor = "yellow";
    }
}
```

Input event - An event triggered when the value of the <input> element changes, like when a user inputs text in the search box. Inside the callback function for an input event, you'll pass a parameter that will hold the data for the text input. Most coders use `e` as the parameter to represent "event."

```
const namefield = document.querySelector("input.name");

namefield.addEventListener("input", function(e) {
  console.log(e.target.value)
}
```

toLowerCase() - A method for converting a string value into all lowercase letters.

```
var quiet = "PLEASE Lower Your Voice"

console.log(quiet.toLowerCase());
// please lower your voice
```

Back to Top ↑

codecademy

Arrays

Property `.length`

The `.length` property of a JavaScript array indicates the number of elements the array contains.

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

numbers.length // 4
```

Index

Array elements are arranged by *index* values, starting at `0` as the first element index. Elements can be accessed by their index using the array name, and the index surrounded by square brackets.

```
// Accessing an array element
const myArray = [100, 200, 300];

console.log(myArray[0]); // 100
console.log(myArray[1]); // 200
console.log(myArray[2]); // 300
```

Method `.push()`

The `.push()` method of JavaScript arrays can be used to add one or more elements to the end of an array. `.push()` mutates the original array returns the new length of the array.

```
// Adding a single element:
const cart = ['apple', 'orange'];
cart.push('pear');

// Adding multiple elements:
const numbers = [1, 2];
numbers.push(3, 4, 5);
```

Method `.pop()`

The `.pop()` method removes the last element from an array and returns that element.

```
const ingredients = ['eggs', 'flour', 'chocolate'];

const poppedIngredient = ingredients.pop(); // 'chocolate'
console.log(ingredients); // ['eggs', 'flour']
```

Mutable

JavaScript arrays are *mutable*, meaning that the values they contain can be changed. Even if they are declared using `const`, the contents can be manipulated by reassigning internal values or using methods like `.push()` and `.pop()`.

```
const names = ['Alice', 'Bob'];

names.push('Carl');
// ['Alice', 'Bob', 'Carl']
```

Arrays

Arrays are lists of ordered, stored data. They can hold items that are of any data type. Arrays are created by using square brackets, with individual elements separated by commas.

```
// An array containing numbers
const numberArray = [0, 1, 2, 3];

// An array containing different data types
const mixedArray = [1, 'chicken', false];
```

Beginner's Essential

Javascript Cheat Sheet

The language of the web.

WebsiteSetup

Table of Contents

Javascript Basics

Including JavaScript in an HTML Page

```
<script type="text/javascript">
  //JS code goes here
</script>
```

Call an External JavaScript File

```
<script src="myscript.js"></script><code></code>
```

Including Comments

```
//
```

Single line comments

```
/* comment here */
```

Multi-line comments

Variables

var, const, let

```
var
```

The most common variable. Can be reassigned but only accessed within a function. Variables defined with var move to the top when code is executed.

```
const
```

Cannot be reassigned and not accessible before they appear within the code.

```
let
```

Similar to const, however, let variable can be reassigned but not re-declared.

Data Types

```
var age = 23
```

Numbers

```
var x
```

Variables

```
var a = "init"
```

Text (strings)

```
var b = 1 + 2 + 3
```

Operations

```
var c = true
```

True or false statements

```
const PI = 3.14
```

Constant numbers

```
var name = {firstName:"John", lastName:"Doe"}
```

Objects

Objects

```
var person = {
  firstName:"John",
  lastName:"Doe",
  age:20,
  nationality:"German"
};
```

Arrays

```
var fruit = ["Banana", "Apple", "Pear"];
```

Array Methods

`concat()`

Join several arrays into one

`indexOf()`

Returns the first position at which a given element appears in an array

`join()`

Combine elements of an array into a single string and return the string

`lastIndexOf()`

Gives the last position at which a given element appears in an array

`pop()`

Removes the last element of an array

`push()`

Add a new element at the end

`reverse()`

Reverse the order of the elements in an array

`shift()`

Remove the first element of an array

`slice()`

Pulls a copy of a portion of an array into a new array

`sort()`

Sorts elements alphabetically

`splice()`

Adds elements in a specified way and position

`toString()`

Converts elements to strings

`unshift()`

Adds a new element to the beginning

`valueOf()`

Returns the primitive value of the specified object

Operators

Basic Operators

```
+     Addition
-     Subtraction
*     Multiplication
/     Division
(..)  Grouping operator
%     Modulus (remainder)
++    Increment numbers
--    Decrement numbers
```

Comparison Operators

```
==    Equal to
===   Equal value and equal type
!=    Not equal
!==   Not equal value or not equal type
>     Greater than
<     Less than
>=    Greater than or equal to
<=    Less than or equal to
?     Ternary operator
```

Logical Operators

```
&&    Logical and
||    Logical or
!     Logical not
```

Bitwise Operators

```
&     AND statement
|     OR statement
~     NOT
^     XOR
<<    Left shift
>>    Right shift
>>>   Zero fill right shift
```

Functions

```
function name(parameter1, parameter2, parameter3) {
  // what the function does
}
```

Outputting Data

`alert()`

Output data in an alert box in the browser window

`confirm()`

Opens up a yes/no dialog and returns true/false depending on user click

`console.log()`

Writes information to the browser console, good for debugging purposes

`document.write()`

Write directly to the HTML document

`prompt()`

Creates an dialogue for user input

Global Functions

`decodeURI()`

Decodes a Uniform Resource Identifier (URI) created by encodeURI or similar

`decodeURIComponent()`

Decodes a URI component

`encodeURI()`

Encodes a URI into UTF-8

`encodeURIComponent()`

Same but for URI components

`eval()`

Evaluates JavaScript code represented as a string

`isFinite()`

Determines whether a passed value is a finite number

`isNaN()`

Determines whether a value is NaN or not

`Number()`

Returns a number converted from its argument

`parseFloat()`

Parses an argument and returns a floating point number

`parseInt()`

Parses its argument and returns an integer

Loops

```
for (before loop; condition for loop; execute after loop) {
  // what to do during the loop
}
```

`for`

The most common way to create a loop in Javascript

`while`

Sets up conditions under which a loop executes

`do while`

Similar to the while loop, however, it executes at least once and performs a check at the end to see if the condition is met to execute again

`break`

Used to stop and exit the cycle at certain conditions

`continue`

Skip parts of the cycle if certain conditions are met of 7 24

If - Else Statements

```
if (condition) {
  // what to do if condition is met
} else {
  // what to do if condition is not met
}
```

Strings

```
var person = "John Doe";
```

Escape Characters

```
\'    – Single quote
\"    – Double quote
\\    – Backslash
\b    – Backspace
\f    – Form feed
\n    – New line
\r    – Carriage return
\t    – Horizontal tabulator
```

`\v` — Vertical tabulator

String Methods

`charAt()`

Returns a character at a specified position inside a string

`charCodeAt()`

Gives you the unicode of character at that position

`concat()`

Concatenates (joins) two or more strings into one

`fromCharCode()`

Returns a string created from the specified sequence of UTF-16 code units

`indexOf()`

Provides the position of the first occurrence of a specified text within a string

`lastIndexOf()`

Same as indexOf() but with the last occurrence, searching backwards

`match()`

Retrieves the matches of a string against a search pattern

`replace()`

Find and replace specific text in a string

`search()`

Executes a search for a matching text and returns its position

`slice()`

Extracts a section of a string and returns it as a new string

`split()`

Splits a string object into an array of strings at a specified position

`substr()`

Similar to slice() but extracts a substring depended on a specified number of characters

`substring()`

Also similar to slice() but can't accept negative indices

`toLowerCase()`

Convert strings to lowercase

`toUpperCase()`

Convert strings to uppercase

`valueOf()`

Returns the primitive value (that has no properties or methods) of a string object

Regular Expressions

Pattern Modifiers

```
e — Evaluate replacement
i — Perform case-insensitive matching
g — Perform global matching
m — Perform multiple line matching
s — Treat strings as single line
x — Allow comments and whitespace in pattern
U — Non Greedy pattern
```

Brackets

```
[abc]      Find any of the characters between the brackets
[^abc]     Find any character not in the brackets
[0-9]      Used to find any digit from 0 to 9
[A-z]      Find any character from uppercase A to lowercase z
(a|b|c)    Find any of the alternatives separated with |
```

Metacharacters

```
.    — Find a single character, except newline or line terminator
\w   — Word character
\W   — Non-word character
\d   — A digit
\D   — A non-digit character
\s   — Whitespace character
\S   — Non-whitespace character
\b   — Find a match at the beginning/end of a word
\B   — A match not at the beginning/end of a word
\0   — NUL character
\n   — A new line character
\f   — Form feed character
\r   — Carriage return character
\t   — Tab character
\v   — Vertical tab character
```

```
\xxx — The character specified by an octal number xxx
\xdd — Character specified by a hexadecimal number dd
\uxxxx — The Unicode character specified by a hexadecimal number xxxx
```

Quantifiers

```
n+    — Matches any string that contains at least one n
n*    — Any string that contains zero or more occurrences of n
n?    — A string that contains zero or one occurrences of n
n{X}  — String that contains a sequence of X n's
n{X,Y} — Strings that contains a sequence of X to Y n's
n{X,} — Matches any string that contains a sequence of at least X n's
n$    — Any string with n at the end of it
^n    — String with n at the beginning of it
?=n   — Any string that is followed by a specific string n
?!n   — String that is not followed by a specific string n
```

Numbers and Math

Number Properties

MAX_VALUE

The maximum numeric value representable in JavaScript

MIN_VALUE

Smallest positive numeric value representable in JavaScript

NaN

The "Not-a-Number" value

NEGATIVE_INFINITY

The negative Infinity value

POSITIVE_INFINITY

Positive Infinity value

Number Methods

toExponential()

Returns a string with a rounded number written as exponential notation

toFixed()

Returns the string of a number with a specified number of decimals

`toPrecision()`

String of a number written with a specified length

`toString()`

Returns a number as a string

`valueOf()`

Returns a number as a number

Math Properties

```
E          Euler's number
LN2        The natural logarithm of 2
LN10       Natural logarithm of 10
LOG2E      Base 2 logarithm of E
LOG10E     Base 10 logarithm of E
PI         The number PI
SQRT1_2    Square root of 1/2
SQRT2      The square root of 2
```

Math Methods

`abs(x)`

Returns the absolute (positive) value of x

`acos(x)`

The arccosine of x, in radians

`asin(x)`

Arcsine of x, in radians

`atan(x)`

The arctangent of x as a numeric value

`atan2(y,x)`

Arctangent of the quotient of its arguments

`ceil(x)`

Value of x rounded up to its nearest integer

`cos(x)`

The cosine of x (x is in radians)

`exp(x)`

Value of Ex

`floor(x)`

The value of x rounded down to its nearest integer

`log(x)`

The natural logarithm (base E) of x

`max(x,y,z,...,n)`

Returns the number with the highest value

`min(x,y,z,...,n)`

Same for the number with the lowest value

`pow(x,y)`

X to the power of y

`random()`

Returns a random number between 0 and 1

`round(x)`

The value of x rounded to its nearest integer

`sin(x)`

The sine of x (x is in radians)

`sqrt(x)`

Square root of x

`tan(x)`

The tangent of an angle

Dealing with Dates

Setting Dates

`Date()`

Creates a new date object with the current date and time

`Date(2017, 5, 21, 3, 23, 10, 0)`

Create a custom date object. The numbers represent year, month, day, hour, minutes, seconds, milliseconds. You can omit anything you want except for year and month.

`Date("2017-06-23")`

Date declaration as a string

Pulling Date and Time Values

`getDate()`

Get the day of the month as a number (1-31)

`getDay()`

The weekday as a number (0-6)

`getFullYear()`

Year as a four digit number (yyyy)

`getHours()`

Get the hour (0-23)

`getMilliseconds()`

The millisecond (0-999)

`getMinutes()`

Get the minute (0-59)

`getMonth()`

Month as a number (0-11)

`getSeconds()`

Get the second (0-59)

`getTime()`

Get the milliseconds since January 1, 1970

`getUTCDate()`

The day (date) of the month in the specified date according to universal time (also available for day, month, fullyear, hours, minutes etc.)

`parse`

Parses a string representation of a date, and returns the number of milliseconds since January 1, 1970

Set Part of a Date

`setDate()`

Set the day as a number (1-31)

`setFullYear()`

Sets the year (optionally month and day)

`setHours()`

Set the hour (0-23)

`setMilliseconds()`

Set milliseconds (0-999)

`setMinutes()`

Sets the minutes (0-59)

`setMonth()`

Set the month (0-11)

`setSeconds()`

Sets the seconds (0-59)

`setTime()`

Set the time (milliseconds since January 1, 1970)

`setUTCDate()`

Sets the day of the month for a specified date according to universal time (also available for day, month, fullyear, hours, minutes etc.)

DOM Node

Node Properties

`attributes`

Returns a live collection of all attributes registered to and element

`baseURI`

Provides the absolute base URL of an HTML element

`childNodes`

Gives a collection of an element's child nodes

`firstChild`

Returns the first child node of an element

`lastChild`

The last child node of an element

`nextSibling`

Gives you the next node at the same node tree level

`nodeName`

Returns the name of a node

`nodeType`

Returns the type of a node

`nodeValue`

Sets or returns the value of a node

`ownerDocument`

The top-level document object for this node

`parentNode`

Returns the parent node of an element

`previousSibling`

Returns the node immediately preceding the current one

`textContent`

Sets or returns the textual content of a node and its descendants

Node Methods

`appendChild()`

Adds a new child node to an element as the last child node

`cloneNode()`

Clones an HTML element

`compareDocumentPosition()`

Compares the document position of two elements

`getFeature()`

Returns an object which implements the APIs of a specified feature

`hasAttributes()`

Returns true if an element has any attributes, otherwise false

`hasChildNodes()`

Returns true if an element has any child nodes, otherwise false

`insertBefore()`

Inserts a new child node before a specified, existing child node

`isDefaultNamespace()`

Returns true if a specified namespaceURI is the default, otherwise false

`isEqualNode()`

Checks if two elements are equal

`isSameNode()`

Checks if two elements are the same node

`isSupported()`

Returns true if a specified feature is supported on the element

`lookupNamespaceURI()`

Returns the namespaceURI associated with a given node

`lookupPrefix()`

Returns a DOMString containing the prefix for a given namespaceURI, if present

`normalize()`

Joins adjacent text nodes and removes empty text nodes in an element

`removeChild()`

Removes a child node from an element

`replaceChild()`

Replaces a child node in an element

Element Methods

`getAttribute()`

Returns the specified attribute value of an element node

`getAttributeNS()`

Returns string value of the attribute with the specified namespace and name

`getAttributeNode()`

Gets the specified attribute node

`getAttributeNodeNS()`

Returns the attribute node for the attribute with the given namespace and name

`getElementsByTagName()`

Provides a collection of all child elements with the specified tag name

`getElementsByTagNameNS()`

Returns a live HTMLCollection of elements with a certain tag name belonging to the given namespace

`hasAttribute()`

Returns true if an element has any attributes, otherwise false

`hasAttributeNS()`

Provides a true/false value indicating whether the current element in a given namespace has the specified attribute

`removeAttribute()`

Removes a specified attribute from an element

`removeAttributeNS()`

Removes the specified attribute from an element within a certain namespace

`removeAttributeNode()`

Takes away a specified attribute node and returns the removed node

`setAttribute()`

Sets or changes the specified attribute to a specified value

`setAttributeNS()`

Adds a new attribute or changes the value of an attribute with the given namespace and name

`setAttributeNode()`

Sets or changes the specified attribute node

`setAttributeNodeNS()`

Adds a new namespaced attribute node to an element

Working with the Browser

Window Properties

closed

Checks whether a window has been closed or not and returns true or false

defaultStatus

Sets or returns the default text in the statusbar of a window

document

Returns the document object for the window

frames

Returns all <iframe> elements in the current window

history

Provides the History object for the window

innerHeight

The inner height of a window's content area

innerWidth

The inner width of the content area

length

Find out the number of <iframe> elements in the window

location

Returns the location object for the window

name

Sets or returns the name of a window

navigator

Returns the Navigator object for the window

opener

Returns a reference to the window that created the window

outerHeight

The outer height of a window, including toolbars/ scrollbars

`outerWidth`

The outer width of a window, including toolbars/ scrollbars

`pageXOffset`

Number of pixels the current document has been scrolled horizontally

`pageYOffset`

Number of pixels the document has been scrolled vertically

`parent`

The parent window of the current window

`screen`

Returns the Screen object for the window

`screenLeft`

The horizontal coordinate of the window (relative to screen)

`screenTop`

The vertical coordinate of the window

`screenX`

Same as screenLeft but needed for some browsers

`screenY`

Same as screenTop but needed for some browsers

`self`

Returns the current window

`status`

Sets or returns the text in the statusbar of a window

`top`

Returns the topmost browser window

Window Methods

`alert()`

Displays an alert box with a message and an OK button

`blur()`

Removes focus from the current window

`clearInterval()`

Clears a timer set with setInterval()

`clearTimeout()`

Clears a timer set with setTimeout()

`close()`

Closes the current window

`confirm()`

Displays a dialogue box with a message and an OK and Cancel button

`focus()`

Sets focus to the current window

`moveBy()`

Moves a window relative to its current position

`moveTo()`

Moves a window to a specified position

`open()`

Opens a new browser window

`print()`

Prints the content of the current window

`prompt()`

Displays a dialogue box that prompts the visitor for input

`resizeBy()`

Resizes the window by the specified number of pixels

`resizeTo()`

Resizes the window to a specified width and height

`scrollBy()`

Scrolls the document by a specified number of pixels

`scrollTo()`

Scrolls the document to specific coordinates

`setInterval()`

Calls a function or evaluates an expression at specified intervals

`setTimeout()`

Calls a function or evaluates an expression after a specified interval

`stop()`

Stops the window from loading

Screen Properties

`availHeight`

Returns the height of the screen (excluding the Windows Taskbar)

`availWidth`

Returns the width of the screen (excluding the Windows Taskbar)

`colorDepth`

Returns the bit depth of the color palette for displaying images

`height`

The total height of the screen

`pixelDepth`

The color resolution of the screen in bits per pixel

`width`

The total width of the screen

Events

Mouse

`onclick`

The event occurs when the user clicks on an element

`oncontextmenu`

User right-clicks on an element to open a context menu

`ondblclick`

The user double-clicks on an element

`onmousedown`

User presses a mouse button over an element

`onmouseenter`

The pointer moves onto an element

`onmouseleave`

Pointer moves out of an element

`onmousemove`

The pointer is moving while it is over an element

`onmouseover`

When the pointer is moved onto an element or one of its children

`onmouseout`

User moves the mouse pointer out of an element or one of its children

`onmouseup`

The user releases a mouse button while over an element

Keyboard

`onkeydown`

When the user is pressing a key down

`onkeypress`

The moment the user starts pressing a key

`onkeyup`

The user releases a key

Frame

`onabort`

The loading of a media is aborted

`onbeforeunload`

Event occurs before the document is about to be unloaded

`onerror`

An error occurs while loading an external file

onhashchange

There have been changes to the anchor part of a URL

onload

When an object has loaded

onpagehide

The user navigates away from a webpage

onpageshow

When the user navigates to a webpage

onresize

The document view is resized

onscroll

An element's scrollbar is being scrolled

onunload

Event occurs when a page has unloaded

Form

onblur

When an element loses focus

onchange

The content of a form element changes (for <input>, <select>and <textarea>)

onfocus

An element gets focus

onfocusin

When an element is about to get focus

onfocusout

The element is about to lose focus

oninput

User input on an element

oninvalid

An element is invalid

onreset

A form is reset

onsearch

The user writes something in a search field (for <input="search">)

onselect

The user selects some text (for <input> and <textarea>)

onsubmit

A form is submitted

Drag

ondrag

An element is dragged

ondragend

The user has finished dragging the element

ondragenter

The dragged element enters a drop target

ondragleave

A dragged element leaves the drop target

ondragover

The dragged element is on top of the drop target

ondragstart

User starts to drag an element

ondrop

Dragged element is dropped on the drop target

Clipboard

oncopy

User copies the content of an element

oncut

The user cuts an element's content

`onpaste`

A user pastes content in an element

Media

`onabort`

Media loading is aborted

`oncanplay`

The browser can start playing media (e.g. a file has buffered enough)

`oncanplaythrough`

When browser can play through media without stopping

`ondurationchange`

The duration of the media changes

`onended`

The media has reached its end

`onerror`

Happens when an error occurs while loading an external file

`onloadeddata`

Media data is loaded

`onloadedmetadata`

Meta Metadata (like dimensions and duration) are loaded

`onloadstart`

Browser starts looking for specified media

`onpause`

Media is paused either by the user or automatically

`onplay`

The media has been started or is no longer paused

`onplaying`

Media is playing after having been paused or stopped for buffering

`onprogress`

Browser is in the process of downloading the media

onratechange

The playing speed of the media changes

onseeked

User is finished moving/skipping to a new position in the media

onseeking

The user starts moving/skipping

onstalled

The browser is trying to load the media but it is not available

onsuspend

Browser is intentionally not loading media

ontimeupdate

The playing position has changed (e.g. because of fast forward)

onvolumechange

Media volume has changed (including mute)

onwaiting

Media paused but expected to resume (for example, buffering)

Animation

animationend

A CSS animation is complete

animationiteration

CSS animation is repeated

animationstart

CSS animation has started

Other

transitionend

Fired when a CSS transition has completed

onmessage

A message is received through the event source

onoffline

Browser starts to work offline

ononline

The browser starts to work online

onpopstate

When the window's history changes

onshow

A <menu> element is shown as a context menu

onstorage

A Web Storage area is updated

ontoggle

The user opens or closes the <details> element

onwheel

Mouse wheel rolls up or down over an element

ontouchcancel

Screen touch is interrupted

ontouchend

User finger is removed from a touch screen

ontouchmove

A finger is dragged across the screen

ontouchstart

Finger is placed on touch screen

Errors

try

Lets you define a block of code to test for errors

catch

Set up a block of code to execute in case of an error

throw

Create custom error messages instead of the standard JavaScript errors

finally

Lets you execute code, after try and catch, regardless of the result

Error Name Values

name

Sets or returns the error name

message

Sets or returns an error message in string from

EvalError

An error has occurred in the eval() function

RangeError

A number is “out of range”

ReferenceError

An illegal reference has occurred

SyntaxError

A syntax error has occurred

TypeError

A type error has occurred

URIError

An encodeURI() error has occurred

codecademy

Async-Await

Async Await Promises

The `async...await` syntax in ES6 offers a new way write more readable and scalable code to handle promises. It uses the same features that were already built into JavaScript.

```
function helloWorld() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Hello World!');
    }, 2000);
  });
}

async function msg() {
  const msg = await helloWorld();
  console.log('Message:', msg);
}

msg(); // Message: Hello World! <-- after 2 seconds
```

Asynchronous JavaScript function

An asynchronous JavaScript function can be created with the `async` keyword before the `function` name, or before `()` when using the async arrow function. An `async` function always returns a promise.

```
function helloWorld() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Hello World!');
    }, 2000);
  });
}

const msg = async function() { //Async Function Expression
  const msg = await helloWorld();
  console.log('Message:', msg);
}

const msg1 = async () => { //Async Arrow Function
  const msg = await helloWorld();
  console.log('Message:', msg);
}

msg(); // Message: Hello World! <-- after 2 seconds
msg1(); // Message: Hello World! <-- after 2 seconds
```

JavaScript aysnc await operator

The JavaScript `async...await` syntax in *ES6* offers a new way write more readable and scablable code to handle promises. A JavaScript `async` function can contain statements preceded by an `await` operator. The operand of `await` is a promise. At an `await` expression, the execution of the `async` function is paused and waits for the operand promise to resolve. The `await` operator returns the promise's resolved value. An `await` operand can only be used inside an `async` function.

```
function helloWorld() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Hello World!');
    }, 2000);
  });
}

async function msg() {
  const msg = await helloWorld();
  console.log('Message:', msg);
}

msg(); // Message: Hello World! <-- after 2 seconds
```

JavaScript async...await advantage

The JavaScript `async...await` syntax allows multiple promises to be initiated and then resolved for values when required during execution of the program. As an alternate to chaining `.then()` functions, it offers better maintainablity of the code and a close resemblance synchronous code.

Classes

Class

JavaScript supports the concept of *classes* as a syntax for creating objects. Classes specify the shared properties and methods that objects produced from the class will have.

When an object is created based on the class, the new object is referred to as an *instance* of the class. New instances are created using the `new` keyword.

The code sample shows a class that represents a `Song`. A new object called `mySong` is created underneath and the `.play()` method on the class is called. The result would be the text `Song playing!` printed in the console.

```
class Song {
  constructor() {
    this.title;
    this.author;
  }

  play() {
    console.log('Song playing!');
  }
}

const mySong = new Song();
mySong.play();
```

Class Constructor

Classes can have a `constructor` method. This is a special method that is called when the object is created (instantiated). Constructor methods are usually used to set initial values for the object.

```
class Song {
  constructor(title, artist) {
    this.title = title;
    this.artist = artist;
  }
}

const mySong = new Song('Bohemian Rhapsody', 'Queen');
console.log(mySong.title);
```

Class Methods

Properties in objects are separated using commas. This is not the case when using the `class` syntax. Methods in classes do not have any separators between them.

```
class Song {
  play() {
    console.log('Playing!');
  }

  stop() {
    console.log('Stopping!');
  }
}
```

`extends`

JavaScript classes support the concept of inheritance — a child class can *extend* a parent class. This is accomplished by using the `extends` keyword as part of the class definition.

Child classes have access to all of the instance properties and methods of the parent class. They can add their own properties and methods in addition to those. A child class constructor calls the parent class constructor using the `super()` method.

```
// Parent class
class Media {
  constructor(info) {
    this.publishDate = info.publishDate;
    this.name = info.name;
  }
}

// Child class
class Song extends Media {
  constructor(songData) {
    super(songData);
    this.artist = songData.artist;
  }
}

const mySong = new Song({
  artist: 'Queen',
  name: 'Bohemian Rhapsody',
  publishDate: 1975
});
```

Conditionals

Control Flow

Control flow is the order in which statements are executed in a program. The default control flow is for statements to be read and executed in order from left-to-right, top-to-bottom in a program file.
Control structures such as conditionals (`if` statements and the like) alter control flow by only executing blocks of code if certain conditions are met. These structures essentially allow a program to make decisions about which code is executed as the program runs.

Logical Operator ||

The logical OR operator `||` checks two values and returns a boolean. If one or both values are truthy, it returns `true` . If both values are falsy, it returns `false` .

A	B	A || B
false	false	false
false	true	true
true	false	true
true	true	true

```
true || false;         // true
10 > 5 || 10 > 20;     // true
false || false;        // false
10 > 100 || 10 > 20;   // false
```

Ternary Operator

The ternary operator allows for a compact syntax in the case of binary (choosing between two choices) decisions. It accepts a condition followed by a `?` operator, and then two expressions separated by a `:` . If the condition evaluates to truthy, the first expression is executed, otherwise, the second expression is executed.

```
let price = 10.5;
let day = "Monday";

day === "Monday" ? price -= 1.5 : price += 1.5;
```

`else` Statement

An `else` block can be added to an `if` block or series of `if` - `else if` blocks. The `else` block will be executed only if the `if` condition fails.

```
const isTaskCompleted = false;

if (isTaskCompleted) {
  console.log('Task completed');
} else {
  console.log('Task incomplete');
}
```

Logical Operator &&

The logical AND operator `&&` checks two values and returns a boolean. If *both* values are truthy, then it returns `true` . If one, or both, of the values is falsy, then it returns `false` .

```
true && true;      // true
1 > 2 && 2 > 1;    // false
true && false;     // false
4 === 4 && 3 > 1;  // true
```

`switch` Statement

The `switch` statements provide a means of checking an expression against multiple `case` clauses. If a case matches, the code inside that clause is executed.
The `case` clause should finish with a `break` keyword. If no case matches but a `default` clause is included, the code inside `default` will be executed.
Note: If `break` is omitted from the block of a `case` , the `switch` statement will continue to check against `case` values until a break is encountered or the flow is broken.

```
const food = 'salad';

switch (food) {
  case 'oyster':
    console.log('The taste of the sea 🦪');
    break;
  case 'pizza':
    console.log('A delicious pie 🍕');
    break;
  default:
    console.log('Enjoy your meal');
}

// Prints: Enjoy your meal
```

Functions

Functions

Functions are one of the fundamental building blocks in JavaScript. A *function* is a reusable set of statements to perform a task or calculate a value. Functions can be passed one or more values and can return a value at the end of their execution. In order to use a function, you must define it somewhere in the scope where you wish to call it.
The example code provided contains a function that takes in 2 values and returns the sum of those numbers.

```
// Defining the function:
function sum(num1, num2) {
  return num1 + num2;
}

// Calling the function:
sum(3, 6); // 9
```

Calling Functions

Functions can be *called*, or executed, elsewhere in code using parentheses following the function name. When a function is called, the code inside its function body runs. *Arguments* are values passed into a function when it is called.

```
// Defining the function
function sum(num1, num2) {
  return num1 + num2;
}

// Calling the function
sum(2, 4); // 6
```

Function Parameters

Inputs to functions are known as *parameters* when a function is declared or defined. Parameters are used as variables inside the function body. When the function is called, these parameters will have the value of whatever is *passed* in as arguments. It is possible to define a function without parameters.

```
// The parameter is name
function sayHello(name) {
  return `Hello, ${name}!`;
}
```

`return` Keyword

Functions return (pass back) values using the `return` keyword. `return` ends function execution and returns the specified value to the location where it was called. A common mistake is to forget the `return` keyword, in which case the function will return `undefined` by default.

```
// With return
function sum(num1, num2) {
  return num1 + num2;
}

// Without return, so the function doesn't output the sum
function sum(num1, num2) {
  num1 + num2;
}
```

Function Declaration

Function *declarations* are used to create named functions. These functions can be called using their declared name. Function declarations are built from:

- The `function` keyword.
- The function name.
- An optional list of parameters separated by commas enclosed by a set of parentheses `()`.
- A function body enclosed in a set of curly braces `{}`.

```
function add(num1, num2) {
  return num1 + num2;
}
```

Anonymous Functions

Anonymous functions in JavaScript do not have a name property. They can be defined using the `function` keyword, or as an arrow function. See the code example for the difference between a named function and an anonymous function.

```
// Named function
function rocketToMars() {
  return 'BOOM!';
}

// Anonymous function
const rocketToMars = function() {
  return 'BOOM!';
}
```

Introduction

JavaScript

JavaScript is a programming language that powers the dynamic behavior on most websites. Alongside HTML and CSS, it is a core technology that makes the web run.

`console.log()`

The `console.log()` method is used to log or print messages to the console. It can also be used to print objects and other info.

```
console.log('Hi there!');
// Prints: Hi there!
```

Strings

Strings are a primitive data type. They are any grouping of characters (letters, spaces, numbers, or symbols) surrounded by single quotes `'` or double quotes `"`.

```
let single = 'Wheres my bandit hat?';
let double = "Wheres my bandit hat?";
```

Numbers

Numbers are a primitive data type. They include the set of all integers and floating point numbers.

```
let amount = 6;
let price = 4.99;
```

Booleans

Booleans are a primitive data type. They can be either `true` or `false`.

```
let lateToWork = true;
```

Null

Null is a primitive data type. It represents the intentional absence of value. In code, it is represented as `null`.

```
let x = null;
```

Arithmetic Operators

JavaScript supports arithmetic operators for:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `%` modulo

```
// Addition
5 + 5
// Subtraction
10 - 5
// Multiplication
5 * 10
// Division
10 / 5
// Modulo
10 % 5
```

String `.length`

The `.length` property of a string returns the number of characters that make up the string.

```
let message = 'good nite';
console.log(message.length);
// Prints: 9

console.log('howdy'.length);
// Prints: 5
```

Methods

Methods return information about an object, and are called by appending an instance with a period `.`, the method name, and parentheses.

```
// Returns a number between 0 and 1
Math.random();
```

Data Instances

When a new piece of data is introduced into a JavaScript program, the program keeps track of it in an instance of that data type. An instance is an individual case of a data type.

Iterators

Functions Assigned to Variables

In JavaScript, functions are a data type just as strings, numbers, and arrays are data types. Therefore, functions can be assigned as values to variables, but are different from all other data types because they can be invoked.

```
let plusFive = (number) => {
  return number + 5;
};
// f is assigned the value of plusFive
let f = plusFive;

plusFive(3); // 8
// Since f has a function value, it can be invoked.
f(9); // 14
```

Callback Functions

In JavaScript, a callback function is a function that is passed into another function as an argument. This function can then be invoked during the execution of that higher order function (that it is an argument of).
Since, in JavaScript, functions are objects, functions can be passed as arguments.

```
const isEven = (n) => {
  return n % 2 == 0;
}

let printMsg = (evenFunc, num) => {
  const isNumEven = evenFunc(num);
  console.log(`The number ${num} is an even number:
${isNumEven}.`)
}

// Pass in isEven as the callback function
printMsg(isEven, 4);
// Prints: The number 4 is an even number: True.
```

Higher-Order Functions

In Javascript, functions can be assigned to variables in the same way that strings or arrays can. They can be passed into other functions as parameters or returned from them as well.
A “higher-order function” is a function that accepts functions as parameters and/or returns a function.

Array Method `.reduce()`

The `.reduce()` method iterates through an array and returns a single value.
It takes a callback function with two parameters `(accumulator, currentValue)` as arguments. On each iteration, `accumulator` is the value returned by the last iteration, and the `currentValue` is the current element. Optionally, a second argument can be passed which acts as the initial value of the accumulator.
Here, the `.reduce()` method will sum all the elements of the array.

```
const arrayOfNumbers = [1, 2, 3, 4];

const sum = arrayOfNumbers.reduce((accumulator, currentValue)
=> {
  return accumulator + currentValue;
});

console.log(sum); // 10
```

Array Method `.forEach()`

The `.forEach()` method executes a callback function on each of the elements in an array in order.
Here, the callback function containing a `console.log()` method will be executed `5` times, once for each element.

```
const numbers = [28, 77, 45, 99, 27];

numbers.forEach(number => {
  console.log(number);
});
```

Array Method `.filter()`

The `.filter()` method executes a callback function on each element in an array. The callback function for each of the elements must return either `true` or `false`. The returned array is a new array with any elements for which the callback function returns `true`.
Here, the array `filteredArray` will contain all the elements of `randomNumbers` but `4`.

```
const randomNumbers = [4, 11, 42, 14, 39];
const filteredArray = randomNumbers.filter(n => {
  return n > 5;
});
```

Loops

While Loop

The `while` loop creates a loop that is executed as long as a specified condition evaluates to `true`. The loop will continue to run until the condition evaluates to `false`. The condition is specified before the loop, and usually, some variable is incremented or altered in the `while` loop body to determine when the loop should stop.

```
while (condition) {
  // code block to be executed
}
```

```
let i = 0;

while (i < 5) {
  console.log(i);
  i++;
}
```

Reverse Loop

A `for` loop can iterate “in reverse” by initializing the loop variable to the starting value, testing for when the variable hits the ending value, and decrementing (subtracting from) the loop variable at each iteration.

```
const items = ['apricot', 'banana', 'cherry'];

for (let i = items.length - 1; i >= 0; i -= 1) {
  console.log(`${i}. ${items[i]}`);
}

// Prints: 2. cherry
// Prints: 1. banana
// Prints: 0. apricot
```

Do...While Statement

A `do...while` statement creates a loop that executes a block of code once, checks if a condition is true, and then repeats the loop as long as the condition is true. They are used when you want the code to always execute at least once. The loop ends when the condition evaluates to false.

```
x = 0
i = 0

do {
  x = x + i;
  console.log(x)
  i++;
} while (i < 5);

// Prints: 0 1 3 6 10
```

For Loop

A `for` loop declares looping instructions, with three important pieces of information separated by semicolons `;`:

- The *initialization* defines where to begin the loop by declaring (or referencing) the iterator variable
- The *stopping condition* determines when to stop looping (when the expression evaluates to `false`)
- The *iteration statement* updates the iterator each time the loop is completed

```
for (let i = 0; i < 4; i += 1) {
  console.log(i);
};

// Output: 0, 1, 2, 3
```

Looping Through Arrays

An array’s length can be evaluated with the `.length` property. This is extremely helpful for looping through arrays, as the `.length` of the array can be used as the stopping condition in the loop.

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

// Output: Every item in the array
```

codecademy

Objects

Dot Notation for Accessing Object Properties

Properties of a JavaScript object can be accessed using the dot notation in this manner: `object.propertyName` . Nested properties of an object can be accessed by chaining key names in the correct order.

```
const apple = {
  color: 'Green',
  price: {
    bulk: '$3/kg',
    smallQty: '$4/kg'
  }
};
console.log(apple.color); // 'Green'
console.log(apple.price.bulk); // '$3/kg'
```

Restrictions in Naming Properties

JavaScript object key names must adhere to some restrictions to be valid. Key names must either be strings or valid identifier or variable names (i.e. special characters such as `-` are not allowed in key names that are not strings).

```
// Example of invalid key names
const trainSchedule = {
  platform num: 10, // Invalid because of the space between
words.
  40 - 10 + 2: 30, // Expressions cannot be keys.
  +compartment: 'C' // The use of a + sign is invalid unless
it is enclosed in quotations.
}
```

Objects

An *object* is a built-in data type for storing key-value pairs. Data inside objects are unordered, and the values can be of any type.

Accessing non-existent JavaScript properties

When trying to access a JavaScript object property that has not been defined yet, the value of `undefined` will be returned by default.

```
const classElection = {
  date: 'January 12'
};

console.log(classElection.place); // undefined
```

JavaScript Objects are Mutable

JavaScript objects are *mutable*, meaning their contents can be changed, even when they are declared as `const` . New properties can be added, and existing property values can be changed or deleted.
It is the *reference* to the object, bound to the variable, that cannot be changed.

```
const student = {
  name: 'Sheldon',
  score: 100,
  grade: 'A',
}

console.log(student)
// { name: 'Sheldon', score: 100, grade: 'A' }

delete student.score
student.grade = 'F'
console.log(student)
// { name: 'Sheldon', grade: 'F' }

student = {}
// TypeError: Assignment to constant variable.
```

Promises

JavaScript Promise Object

A JavaScript `Promise` is an object that can be used to get the outcome of an asynchronous operation when that result is not instantly available.
Since JavaScript code runs in a non-blocking manner, promises become essential when we have to wait for some asynchronous operation without holding back the execution of the rest of the code.

States of a JavaScript Promise

A JavaScript `Promise` object can be in one of three states: `pending`, `resolved`, or `rejected`.
While the value is not yet available, the `Promise` stays in the `pending` state.
Afterwards, it transitions to one of the two states: `resolved` or `rejected`.
A resolved promise stands for a successful completion. Due to errors, the promise may go in the `rejected` state.
In the given code block, if the `Promise` is on `resolved` state, the first parameter holding a callback function of the `then()` method will print the resolved value. Otherwise, an alert will be shown.

```
const promise = new Promise((resolve, reject) => {
  const res = true;
  // An asynchronous operation.
  if (res) {
    resolve('Resolved!');
  }
  else {
    reject(Error('Error'));
  }
});

promise.then((res) => console.log(res), (err) => alert(err));
```

Creating a Javascript Promise object

An instance of a JavaScript `Promise` object is created using the `new` keyword.
The constructor of the `Promise` object takes a function, known as the *executor function*, as the argument. This function is responsible for resolving or rejecting the promise.

```
const executorFn = (resolve, reject) => {
  console.log('The executor function of the promise!');
};

const promise = new Promise(executorFn);
```

Executor function of JavaScript Promise object

A JavaScript promise's executor function takes two functions as its arguments. The first parameter represents the function that should be called to resolve the promise and the other one is used when the promise should be rejected. A `Promise` object may use any one or both of them inside its executor function.
In the given example, the promise is always resolved unconditionally by the `resolve` function. The `reject` function could be used for a rejection.

```
const executorFn = (resolve, reject) => {
  resolve('Resolved!');
};

const promise = new Promise(executorFn);
```

`setTimeout()`

`setTimeout()` is an asynchronous JavaScript function that executes a code block or evaluates an expression through a callback function after a delay set in milliseconds.

```
const loginAlert = () =>{
  alert('Login');
};

setTimeout(loginAlert, 6000);
```

.then() method of a JavaScript Promise object

The `.then()` method of a JavaScript `Promise` object can be used to get the eventual result (or error) of the asynchronous operation.
`.then()` accepts two function arguments. The first handler supplied to it will be called if the promise is resolved. The second one will be called if the promise is rejected.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Result');
  }, 200);
});

promise.then((res) => {
  console.log(res);
}, (err) => {
  alert(err);
});
```

Chaining multiple .then() methods

The `.then()` method returns a Promise, even if one or both of the handler functions are absent. Because of this, multiple `.then()` methods can be chained together. This is known as composition.
In the code block, a couple of `.then()` methods are chained together. Each method deals with the resolved value of their respective promises.

```
const promise = new Promise(resolve => setTimeout(() =>
resolve('dAlan'), 100));

promise.then(res => {
  return res === 'Alan' ? Promise.resolve('Hey Alan!')
: Promise.reject('Who are you?')
}).then((res) => {
  console.log(res)
}, (err) => {
  alert(err)
});
```

The `.catch()` method for handling rejection

The function passed as the second argument to a `.then()` method of a promise object is used when the promise is rejected. An alternative to this approach is to use the JavaScript `.catch()` method of the promise object. The information for the rejection is available to the handler supplied in the `.catch()` method.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject(Error('Promise Rejected Unconditionally.'));
  }, 1000);
});

promise.then((res) => {
  console.log(value);
});

promise.catch((err) => {
  alert(err);
});
```

Avoiding nested `Promise` and `.then()`

In JavaScript, when performing multiple asynchronous operations in a sequence, promises should be composed by chaining multiple `.then()` methods. This is better practice than nesting.
Chaining helps streamline the development process because it makes the code more readable and easier to debug.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('*');
  }, 1000);
});

const twoStars = (star) => {
  return (star + star);
};

const oneDot = (star) => {
  return (star + '.');
};

const print = (val) => {
  console.log(val);
};

// Chaining them all together
promise.then(twoStars).then(oneDot).then(print);
```

JavaScript Promise.all()

The JavaScript `Promise.all()` method can be used to execute multiple promises in parallel. The function accepts an array of promises as an argument. If all of the promises in the argument are resolved, the promise returned from `Promise.all()` will resolve to an array containing the resolved values of all the promises in the order of the initial array. Any rejection from the list of promises will cause the greater promise to be rejected.

In the code block, `3` and `2` will be printed respectively even though `promise1` will be resolved after `promise2`.

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(3);
  }, 300);
});
const promise2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(2);
  }, 200);
});

Promise.all([promise1, promise2]).then((res) => {
  console.log(res[0]);
  console.log(res[1]);
});
```

JavaScript `for...in` loop

The JavaScript `for...in` loop can be used to iterate over the keys of an object. In each iteration, one of the properties from the object is assigned to the variable of that loop.

```
let mobile = {
  brand: 'Samsung',
  model: 'Galaxy Note 9'
};

for (let key in mobile) {
  console.log(`${key}: ${mobile[key]}`);
}
```

Properties and values of a JavaScript object

A JavaScript object literal is enclosed with curly braces `{}`. Values are mapped to keys in the object with a colon (`:`), and the key-value pairs are separated by commas. All the keys are unique, but values are not.
Key-value pairs of an object are also referred to as *properties*.

```
const classOf2018 = {
  students: 38,
  year: 2018
}
```

Delete operator

Once an object is created in JavaScript, it is possible to remove properties from the object using the `delete` operator. The `delete` keyword deletes both the value of the property and the property itself from the object. The `delete` operator only works on properties, not on variables or functions.

```
const person = {
  firstName: "Matilda",
  age: 27,
  hobby: "knitting",
  goal: "learning JavaScript"
};

delete person.hobby; // or delete person[hobby];

console.log(person);
/*
{
  firstName: "Matilda"
  age: 27
  goal: "learning JavaScript"
}
*/
```

javascript passing objects as arguments

When JavaScript objects are passed as arguments to functions or methods, they are passed by *reference*, not by value. This means that the object itself (not a copy) is accessible and mutable (can be changed) inside that function.

```
const origNum = 8;
const origObj = {color: 'blue'};

const changeItUp = (num, obj) => {
  num = 7;
  obj.color = 'red';
};

changeItUp(origNum, origObj);

// Will output 8 since integers are passed by value.
console.log(origNum);

// Will output 'red' since objects are passed
// by reference and are therefore mutable.
console.log(origObj.color);
```

JavaScript Object Methods

JavaScript objects may have property values that are *functions*. These are referred to as object *methods*.

Methods may be defined using anonymous *arrow function expressions*, or with *shorthand method syntax*.

Object methods are invoked with the syntax: `objectName.methodName(arguments)` .

```
const engine = {
  // method shorthand, with one argument
  start(adverb) {
    console.log(`The engine starts up ${adverb}...`);
  },
  // anonymous arrow function expression with no arguments
  sputter: () => {
    console.log('The engine sputters...');
  },
};

engine.start('noisily');
engine.sputter();

/* Console output:
The engine starts up noisily...
The engine sputters...
*/
```

`this` Keyword

The reserved keyword `this` refers to a method's calling object, and it can be used to access properties belonging to that object.

Here, using the `this` keyword inside the object function to refer to the `cat` object and access its `name` property.

```
const cat = {
  name: 'Pipey',
  age: 8,
  whatName() {
    return this.name
  }
};

console.log(cat.whatName());
// Output: Pipey
```

javascript function this

Every JavaScript function or method has a `this` context. For a function defined inside of an object, `this` will refer to that object itself. For a function defined outside of an object, `this` will refer to the global object (`window` in a browser, `global` in Node.js).

```
const restaurant = {
  numCustomers: 45,
  seatCapacity: 100,
  availableSeats() {
    // this refers to the restaurant object
    // and it's used to access its properties
    return this.seatCapacity - this.numCustomers;
  }
}
```

JavaScript Arrow Function this Scope

JavaScript arrow functions do not have their own `this` context, but use the `this` of the surrounding lexical context. Thus, they are generally a poor choice for writing object methods.

Consider the example code:

`loggerA` is a property that uses arrow notation to define the function. Since `data` does not exist in the global context, accessing `this.data` returns `undefined` .

`loggerB` uses method syntax. Since `this` refers to the enclosing object, the value of the `data` property is accessed as expected, returning `"abc"` .

```
const myObj = {
    data: 'abc',
    loggerA: () => { console.log(this.data); },
    loggerB() { console.log(this.data); },
};

myObj.loggerA();    // undefined
myObj.loggerB();    // 'abc'
```

javascript getters and setters restricted

JavaScript object properties are not private or protected. Since JavaScript objects are passed by reference, there is no way to fully prevent incorrect interactions with object properties.
One way to implement more restricted interactions with object properties is to use *getter* and *setter* methods.
Typically, the internal value is stored as a property with an identifier that matches the *getter* and *setter* method names, but begins with an underscore (_).

```
const myCat = {
  _name: 'Dottie',
  get name() {
    return this._name;
  },
  set name(newName) {
    this._name = newName;
  }
};

// Reference invokes the getter
console.log(myCat.name);

// Assignment invokes the setter
myCat.name = 'Yankee';
```

getters and setters intercept property access

JavaScript getter and setter methods are helpful in part because they offer a way to intercept property access and assignment, and allow for additional actions to be performed before these changes go into effect.

```
const myCat = {
  _name: 'Snickers',
  get name(){
    return this._name
  },
  set name(newName){
    //Verify that newName is a non-empty string before setting
as name property
    if (typeof newName === 'string' && newName.length > 0){
      this._name = newName;
    } else {
      console.log("ERROR: name must be a non-empty string");
    }
  }
}
```

javascript factory functions

A JavaScript function that returns an object is known as a *factory function*. Factory functions often accept parameters in order to customize the returned object.

```
// A factory function that accepts 'name',
// 'age', and 'breed' parameters to return
// a customized dog object.
const dogFactory = (name, age, breed) => {
  return {
    name: name,
    age: age,
    breed: breed,
    bark() {
      console.log('Woof!');
    }
  };
};
```

JavaScript destructuring assignment shorthand syntax

The JavaScript *destructuring assignment* is a shorthand syntax that allows object properties to be extracted into specific variable values.
It uses a pair of curly braces (`{}`) with property names on the left-hand side of an assignment to extract values from objects. The number of variables can be less than the total properties of an object.

```
const rubiksCubeFacts = {
  possiblePermutations: '43,252,003,274,489,856,000',
  invented: '1974',
  largestCube: '17x17x17'
};
const {possiblePermutations, invented, largestCube}
= rubiksCubeFacts;
console.log(possiblePermutations); //
'43,252,003,274,489,856,000'
console.log(invented); // '1974'
console.log(largestCube); // '17x17x17'
```

shorthand property name syntax for object creation

The *shorthand property name* syntax in JavaScript allows creating objects without explicitly specifying the property names (ie. explicitly declaring the value after the key). In this process, an object is created where the property names of that object match variables which already exist in that context. Shorthand property names populate an object with a key matching the identifier and a value matching the identifier's value.

```
const activity = 'Surfing';
const beach = { activity };
console.log(beach); // { activity: 'Surfing' }
```

Break Keyword

Within a loop, the `break` keyword may be used to exit the loop immediately, continuing execution after the loop body.
Here, the `break` keyword is used to exit the loop when `i` is greater than 5.

```
for (let i = 0; i < 99; i += 1) {
  if (i > 5) {
     break;
  }
  console.log(i)
}

// Output: 0 1 2 3 4 5
```

Nested For Loop

A nested `for` loop is when a `for` loop runs inside another `for` loop.
The inner loop will run all its iterations for *each* iteration of the outer loop.

```
for (let outer = 0; outer < 2; outer += 1) {
  for (let inner = 0; inner < 3; inner += 1) {
    console.log(`${outer}-${inner}`);
  }
}

/*
Output:
0-0
0-1
0-2
1-0
1-1
1-2
*/
```

Loops

A *loop* is a programming tool that is used to repeat a set of instructions. *Iterate* is a generic term that means “to repeat” in the context of *loops*. A *loop* will continue to *iterate* until a specified condition, commonly known as a *stopping condition*, is met.

Array Method `.map()`

The `.map()` method executes a callback function on each element in an array. It returns a new array made up of the return values from the callback function.
The original array does not get altered, and the returned array may contain different elements than the original array.

```
const finalParticipants = ['Taylor', 'Donald', 'Don',
'Natasha', 'Bobby'];

const announcements = finalParticipants.map(member => {
  return member + ' joined the contest.';
})

console.log(announcements);
```

Libraries

Libraries contain methods that can be called by appending the library name with a period `.`, the method name, and a set of parentheses.

```
Math.random();
// ☝ Math is the library
```

`Math.random()`

The `Math.random()` function returns a floating-point, random number in the range from 0 (inclusive) up to but not including 1.

```
console.log(Math.random());
// Prints: 0 - 0.9
```

`Math.floor()`

The `Math.floor()` function returns the largest integer less than or equal to the given number.

```
console.log(Math.floor(5.95));
// Prints: 5
```

Single Line Comments

In JavaScript, single-line comments are created with two consecutive forward slashes `//`.

```
// This line will denote a comment
```

Multi-line Comments

In JavaScript, multi-line comments are created by surrounding the lines with `/*` at the beginning and `*/` at the end. Comments are good ways for a variety of reasons like explaining a code block or indicating some hints, etc.

```
/*
The below configuration must be
changed before deployment.
*/

let baseUrl = 'localhost/taxwebapp/country';
```

Remainder / Modulo Operator

The remainder operator, sometimes called modulo, returns the number that remains after the right-hand number divides into the left-hand number as many times as it evenly can.

```
// calculates # of weeks in a year, rounds down to nearest
integer
const weeksInYear = Math.floor(365/7);

// calcuates the number of days left over after 365 is divded
by 7
const daysLeftOver = 365 % 7 ;

console.log("A year has " + weeksInYear + " weeks and "
+ daysLeftOver + " days");
```

Learn Javascript: Variables

A variable is a container for data that is stored in computer memory. It is referenced by a descriptive name that a programmer can call to assign a specific value and retrieve it.

```
// examples of variables
let name = "Tammy";
const found = false;
var age = 3;
console.log(name, found, age);
// Tammy, false, 3
```

`const` Keyword

A constant variable can be declared using the keyword `const`. It must have an assignment. Any attempt of re-assigning a `const` variable will result in JavaScript runtime error.

```
const numberOfColumns = 4;
numberOfColumns = 8;
// TypeError: Assignment to constant variable.
```

`let` Keyword

`let` creates a local variable in JavaScript & can be re-assigned. Initialization during the declaration of a `let` variable is optional. A `let` variable will contain `undefined` if nothing is assigned to it.

```
let count;
console.log(count); // Prints: undefined
count = 10;
console.log(count); // Prints: 10
```

Undefined

`undefined` is a primitive JavaScript value that represents lack of defined value.
Variables that are declared but not initialized to a value will have the value `undefined`.

```
var a;

console.log(a);
// Prints: undefined
```

Assignment Operators

An assignment operator assigns a value to its left operand based on the value of its right operand. Here are some of them:

- `+=` addition assignment
- `-=` subtraction assignment
- `*=` multiplication assignment
- `/=` division assignment

```
let number = 100;

// Both statements will add 10
number = number + 10;
number += 10;

console.log(number);
// Prints: 120
```

String Concatenation

In JavaScript, multiple strings can be concatenated together using the `+` operator. In the example, multiple strings and variables containing string values have been concatenated. After execution of the code block, the `displayText` variable will contain the concatenated string.

```
let service = 'credit card';
let month = 'May 30th';
let displayText = 'Your ' + service  + ' bill is due on '
+  month + '.';

console.log(displayText);
// Prints: Your credit card bill is due on May 30th.
```

String Interpolation

String interpolation is the process of evaluating string literals containing one or more placeholders (expressions, variables, etc).
It can be performed using template literals: `text ${expression} text`.

```
let age = 7;

// String concatenation
'Tommy is ' + age + ' years old.';

// String interpolation
`Tommy is ${age} years old.`;
```

Template Literals

Template literals are strings that allow embedded expressions, `${expression}`. While regular strings use single `'` or double `"` quotes, template literals use backticks instead.

```
let name = "Codecademy";
console.log(`Hello, ${name}`);
// Prints: Hello, Codecademy

console.log(`Billy is ${6+8} years old.`);
// Prints: Billy is 14 years old.
```

Variables

Variables are used whenever there's a need to store a piece of data. A variable contains data that can be used in the program elsewhere. Using variables also ensures code re-usability since it can be used to replace the same value in multiple places.

```
const currency = '$';
let userIncome = 85000;

console.log(currency + userIncome + ' is more than the average
income.');
// Prints: $85000 is more than the average income.
```

Declaring Variables

To declare a variable in JavaScript, any of these three keywords can be used along with a variable name:

- `var` is used in pre-ES6 versions of JavaScript.
- `let` is the preferred way to declare a variable when it can be reassigned.
- `const` is the preferred way to declare a variable with a constant value.

```
var age;
let weight;
const numberOfFingers = 20;
```

Function Expressions

Function *expressions* create functions inside an expression instead of as a function declaration. They can be anonymous and/or assigned to a variable.

```
const dog = function() {
  return 'Woof!';
}
```

Arrow Functions (ES6)

Arrow function expressions were introduced in ES6. These expressions are clean and concise. The syntax for an arrow function expression does not require the `function` keyword and uses a fat arrow `=>` to separate the parameter(s) from the body. There are several variations of arrow functions:

- Arrow functions with a single parameter do not require `()` around the parameter list.
- Arrow functions with a single expression can use the concise function body which returns the result of the expression without the `return` keyword.

```
// Arrow function with two arguments
const sum = (firstParam, secondParam) => {
  return firstParam + secondParam;
};
console.log(sum(2,5)); // Prints: 7

// Arrow function with no arguments
const printHello = () => {
  console.log('hello');
};
printHello(); // Prints: hello

// Arrow functions with a single argument
const checkWeight = weight => {
  console.log(`Baggage weight : ${weight} kilograms.`);
};
checkWeight(25); // Prints: Baggage weight : 25 kilograms.


// Concise arrow functions
const multiply = (a, b) => a * b;
console.log(multiply(2, 30)); // Prints: 60
```

Requests

Asynchronous calls with XMLHttpRequest

AJAX enables HTTP requests to be made not only during the load time of a web page but also anytime after a page initially loads. This allows adding dynamic behavior to a webpage. This is essential for giving a good user experience without reloading the webpage for transferring data to and from the web server.
The XMLHttpRequest (XHR) web API provides the ability to make the actual asynchronous request and uses AJAX to handle the data from the request.
The given code block is a basic example of how an HTTP GET request is made to the specified URL.

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'mysite.com/api/getjson');
```

HTTP POST request

HTTP `POST` requests are made with the intention of sending new information to the source (server) that will receive it.
For a `POST` request, the new information is stored in the *body* of the request.

HTTP GET request

HTTP `GET` requests are made with the intention of retrieving information or data from a source (server) over the web.
`GET` requests have no *body*, so the information that the source requires, in order to return the proper response, must be included in the request URL path or query string.

The query string in a URL

Query strings are used to send additional information to the server during an HTTP GET request.
The query string is separated from the original URL using the question mark character `?`.
In a query string, there can be one or more key-value pairs joined by the equal character `=`.
For separating multiple key-value pairs, an ampersand character `&` is used.
Query strings should be url-encoded in case of the presence of URL unsafe characters.

```
const requestUrl = 'http://mysite.com/api/vendor?
name=kavin&id=35412';
```

JSON: JavaScript Object Notation

JSON or *JavaScript Object Notation* is a data format suitable for transporting data to and from a server.
It is essentially a slightly stricter version of a Javascript object. A JSON object should be enclosed in curly braces and may contain one or more property-value pairs. JSON names require double quotes, while standard Javascript objects do not.

```
const jsonObj = {
  "name": "Rick",
  "id": "11A",
  "level": 4
};
```

`XMLHttpRequest` GET Request Requirements

The request type, response type, request URL, and handler for the response data must be provided in order to make an HTTP GET request with the JavaScript `XMLHttpRequest` API.
The URL may contain additional data in the query string. For an HTTP GET request, the request type must be `GET`.

```
const req = new XMLHttpRequest();
req.responseType = 'json';
req.open('GET', '/myendpoint/getdata?id=65');
req.onload = () => {
  console.log(req.response);
};

req.send();
```

codecademy

Scope

Scope

Scope is a concept that refers to where values and functions can be accessed.
Various scopes include:

- *Global* scope (a value/function in the global scope can be used anywhere in the entire program)
- *File* or *module* scope (the value/function can only be accessed from within the file)
- *Function* scope (only visible within the function),
- *Code block* scope (only visible within a `{ ... }` codeblock)

```
function myFunction() {

  var pizzaName = "Volvo";
  // Code here can use pizzaName

}

// Code here can't use pizzaName
```

Block Scoped Variables

`const` and `let` are *block scoped* variables, meaning they are only accessible in their block or nested blocks. In the given code block, trying to print the `statusMessage` using the `console.log()` method will result in a `ReferenceError`. It is accessible only inside that `if` block.

```
const isLoggedIn = true;

if (isLoggedIn == true) {
  const statusMessage = 'User is logged in.';
}

console.log(statusMessage);

// Uncaught ReferenceError: statusMessage is not defined
```

Global Variables

JavaScript variables that are declared outside of blocks or functions can exist in the *global scope*, which means they are accessible throughout a program. Variables declared outside of smaller block or function scopes are accessible inside those smaller scopes.
Note: It is best practice to keep global variables to a minimum.

```
// Variable declared globally
const color = 'blue';

function printColor() {
  console.log(color);
}

printColor(); // Prints: blue
```

HTTP POST request with the `XMLHttpRequest` API

To make an HTTP POST request with the JavaScript `XMLHttpRequest` API, a request type, response type, request URL, request body, and handler for the response data must be provided. The request body is essential because the information sent via the POST method is not visible in the URL. The request type must be `POST` for this case. The response type can be a variety of types including array buffer, json, etc.

```
const data = {
  fish: 'Salmon',
  weight: '1.5 KG',
  units: 5
};
const xhr = new XMLHttpRequest();
xhr.open('POST', '/inventory/add');
xhr.responseType = 'json';
xhr.send(JSON.stringify(data));

xhr.onload = () => {
  console.log(xhr.response);
};
```

ok property fetch api

In a Fetch API function `fetch()` the `ok` property of a response checks to see if it evaluates to `true` or `false` . In the code example the `.ok` property will be `true` when the HTTP request is successful. The `.ok` property will be `false` when the HTTP request is unsuccessful.

```
    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)
  })
}
```

JSON Formatted response body

The `.json()` method will resolve a returned promise to a JSON object, parsing the body text as JSON.
The example block of code shows `.json()` method that returns a promise that resolves to a JSON-formatted response body as a JavaScript object.

```
fetch('url-that-returns-JSON')
.then(response => response.json())
.then(jsonResponse => {
  console.log(jsonResponse);
});
```

promise url parameter fetch api

A JavaScript Fetch API is used to access and manipulate requests and responses within the HTTP pipeline, fetching resources asynchronously across a network.
A basic `fetch()` request will accept a URL parameter, send a request and contain a success and failure promise handler function.
In the example, the block of code begins by calling the `fetch()` function. Then a `then()` method is chained to the end of the `fetch()` . It ends with the response callback to handle success and the rejection callback to handle failure.

```
fetch('url')
.then(
  response  => {
    console.log(response);
  },
 rejection => {
    console.error(rejection.message);
);
```

Fetch API Function

The Fetch API function `fetch()` can be used to create requests. Though accepting additional arguments, the request can be customized. This can be used to change the request type, headers, specify a request body, and much more as shown in the example block of code.

```
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 => {
  console.log(jsonResponse);
})
```

async await syntax

The **async...await** syntax is used with the JS Fetch API `fetch()` to work with promises. In the code block example we see the keyword `async` placed the function. This means that the function will return a promise. The keyword `await` makes the JavaScript wait until the problem is resolved.

```
const getSuggestions = async () => {
  const wordQuery = inputField.value;
  const endpoint = `${url}${queryParams}${wordQuery}`;
  try{
const response = __~await~__ __~fetch(endpoint, {cache: 'no-
cache'});
    if(response.ok){
      const jsonResponse = await response.json()
    }
  }
  catch(error){
    console.log(error)
  }
}
```

`if` Statement

An `if` statement accepts an expression with a set of parentheses:

- If the expression evaluates to a truthy value, then the code within its code body executes.
- If the expression evaluates to a falsy value, its code body will not execute.

```
const isMailSent = true;

if (isMailSent) {
  console.log('Mail sent to recipient');
}
```

Truthy and Falsy

In JavaScript, values evaluate to `true` or `false` when evaluated as Booleans.

- Values that evaluate to `true` are known as *truthy*
- Values that evaluate to `false` are known as *falsy*

Falsy values include `false`, `0`, empty strings, `null` `undefined`, and `NaN`. All other values are truthy.

Logical Operator `!`

The logical NOT operator `!` can be used to do one of the following:

- Invert a Boolean value.
- Invert the truthiness of non-Boolean values.

```
let lateToWork = true;
let oppositeValue = !lateToWork;
console.log(oppositeValue);
// Prints: false
```

Comparison Operators

Comparison operators are used to comparing two values and return `true` or `false` depending on the validity of the comparison:

- `===` strict equal
- `!==` strict not equal
- `>` greater than
- `>=` greater than or equal
- `<` less than
- `<=` less than or equal

```
1 > 3       // false
3 > 1       // true
250 >= 250  // true
1 === 1     // true
1 === 2     // false
1 === '1'   // false
```

`else if` Clause

After an initial `if` block, `else if` blocks can each check an additional condition. An optional `else` block can be added after the `else if` block(s) to run by default if none of the conditionals evaluated to truthy.

```
const size = 10;

if (size > 100) {
  console.log('Big');
} else if (size > 20) {
  console.log('Medium');
} else if (size > 4) {
  console.log('Small');
} else {
  console.log('Tiny');
}
// Print: Small
```

Static Methods

Within a JavaScript class, the `static` keyword defines a static method for a class. Static methods are not called on individual instances of the class, but are called on the class itself. Therefore, they tend to be general (utility) methods.

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

  introduce() {
    console.log('This is ' + this._name + ' !');
  }

  // A static method
  static bark() {
    console.log('Woof!');
  }
}

const myDog = new Dog('Buster');
myDog.introduce();

// Calling the static method
Dog.bark();
```

Async Function Error Handling

JavaScript `async` functions uses `try...catch` statements for error handling. This method allows shared error handling for synchronous and asynchronous code.

```
let json = '{ "age": 30 }'; // incomplete data

try {
  let user = JSON.parse(json); // <-- no errors
  alert( user.name ); // no name!
} catch (e) {
  alert( "Invalid JSON data!" );
}
```

Using async await syntax

Constructing one or more promises or calls without `await` can allow multiple `async` functions to execute simultaneously. Through this approach, a program can take advantage of *concurrency*, and asynchronous actions can be initiated within an `async` function. Since using the `await` keyword halts the execution of an `async` function, each async function can be awaited once its value is required by program logic.

Resolving JavaScript Promises

When using JavaScript `async...await`, multiple asynchronous operations can run concurrently. If the resolved value is required for each promise initiated, `Promise.all()` can be used to retrieve the resolved value, avoiding unnecessary blocking.

```
let promise1 = Promise.resolve(5);
let promise2 = 44;
let promise3 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2,
promise3]).then(function(values) {
  console.log(values);
});
// expected output: Array [5, 44, "foo"]
```