codecademy

Introduction

`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!
```

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.

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();
```

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
```

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;
```

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
```

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.

Booleans

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

```
let lateToWork = true;
```

`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
```

Null

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

```
let x = null;
```

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?";
```

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
```

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.

```
const weeksInYear = Math.floor(365/7) // calculates # of
weeks in a year, rounds down to nearest integer
const daysLeftOver = 367 % 7 // calcuates the number of days
left over after 365 is divded by 7

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

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 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.`;
```

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.
```

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
```

Learn Javascript: Variables

codecademy

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
```

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;
```

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.
```

`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
```

`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.
```

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.
```

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
```

`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');
}
```

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
```

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.

Functions

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
```

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
```

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!';
}
```

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!';
}
```

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;
}
```

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
```

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
```

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];
```

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
```

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.

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;
});
```

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);
```

Objects

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.
}
```

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'
```

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.
```

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...
*/
```

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' }
```

`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'
```

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 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';
```

Browser Compatibility and Transpilation

Running scripts with npm

Command line shortcuts can be defined in a **package.json** file in a *"scripts"* object. This is used to combine multiple command line instructions in one command. These shortcuts can be executed from the terminal with the `npm run` command. For example, a script command named `"build"` can be executed with `npm run build`.

Babel Package Installation

The `babel-cli` JavaScript package contains Babel command line tools. The `babel-preset-env` JavaScript package is used to transpile *ES6* JavaScript code to *ES5*. They should be installed with the flag `-D` in the command line instruction like `npm install -D babel-preset-env` to be installed as development dependency.

ES5 & ES6 Compatibility

ECMAScript (ES) is a scripting language specification standardized by Ecma International for Javascript. ES5 is the older JavaScript version which is supported by all web browsers while the ES6 version, released in 2015, is not supported by all web browsers.

Installing Development JavaScript Packages

When a `-D` flag is used to install a JavaScript package using the command line, this adds the package under the property called *devDependencies* in a **package.json** file for the project. Whenever the developer runs `npm install`, all the listed packages and their dependencies will be installed.

ES6 JavaScript backwards compatibility

ES6 is a version of JavaScript that was released in 2015 and is backward compatible. One example of how to do this is the JavaScript *Babel* library which *transpiles ES6* JavaScript to *ES5*. Transpilation is the process of converting one programming language to another.

Babel configuration file

Babel uses the **.babelrc** file as a configuration file to determine the JavaScript file's presets, plugins and more. It can be created with the command `touch .babelrc`.

Babel build process

Babel uses the `npm run build` commend to covert all the JavaScript *ES6* files in the **src** folder of the project to *ES5*. This conversion makes the JavaScript code compatible on all modern browsers. The *ES5* code is written in a new file named **main.js** within the folder called **lib**

Installing JavaScript Packages

The *npm install* command installs JavaScript packages to the project directory. It creates a folder called **node_modules** and copies the JavaScript package files to it. This command also installs all the dependencies for the given package.

Node Package Manager

The *node package manager*, or *npm*, is a package manager for JavaScript which is used to access and manage Node packages - modules that contain JavaScript code written by other developers that are meant to provide helpful tools, reduce duplication of work, and reduce bugs.

Initiate JavaScript project

The command line instruction *npm init* is used to set up a new JavaScript project. A **package.json** file is created by the *npm init* command and contains a list of key-value pairs that provides information about the JavaScript project, such as the project name, version, description, a list of node packages required for the project, command line scripts, and more.

Caniuse.com for checking browser support

The website `caniuse.com` is useful for looking up the percent of browsers that support certain features in HTML, CSS, JavaScript, and their libraries.
For instance, you can find out what percent of browsers support specific features in ES6, as covered in the “Browser compatibility and transpilition” lesson.

Reasons to update JavaScript

Ecma international updated JavaScript from *ES5* to *ES6* in 2015 to make JavaScript syntax more similar to other languages, make JavaScript syntax more efficient and easier to read, and address *ES5* bugs.

Modules

Import Javascript modules with the require function

In Node.js, the `require` function can used to import code from another file into the current script.

```
var moduleA = require( "./module-a.js" );

// The .js extension is optional
var moduleA = require( "./module-a" );
// Both ways will produce the same result.

// Now the functionality of moduleA can be used
console.log(moduleA.someFunctionality)
```

Javascript export default

As of ES6, the *export default* keywords allow for a single variable or function to be exported, then, in another script, it will be straightforward to import the default export.
After using the *export default* it is possible to import a variable or function without using the `require()` function.

```
// module "moduleA.js"
export default function cube(x) {
  return x * x * x;
}

// In main.js
import cube from './moduleA.js';
// Now the `cube` function can be used straightforwardly.
console.log(cube(3)); // 27
```

Intermediate Javascript: Export Module

To make an object in our Javascript file exportable as a module in Node.js, we assign the object to the `exports` property of `module`.

```
let Course = {};
Course.name = "Javascript Node.js"
module.exports = Course;
```

Using the import keyword in Javascript

As of ES6, the `import` keyword can be used to import functions, objects or primitives previously exported into the current script.

There are many ways to use the `import` keyword, for example, you can import all the exports from a script by using the `*` selector as follows: `import * from 'module_name';` .

A single function can be imported with curly brackets as follows: `import {funcA} as name from 'module_name';`

Or many functions by name: `import {funcA, funcB} as name from 'module_name';`

```
// add.js
export const add = (x, y) => {
    return x + y
}


// main.js
import { add } from './add';
console.log(add(2, 3)); // 5
```

Promises

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));
```

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);
});
```

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]);
});
```

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);
```

.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);
});
```

`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);
```

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);
```

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);
```

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.

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)
});
```

Async-Await

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"]
```

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
```

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
```

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.

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.

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!" );
}
```

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
```

Requests

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.

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
};
```

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');
```

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';
```

`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(xhr.response);
};

req.send();
```

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 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)
  }
}
```