Comments

Examples

```
// This is an in-line comment.

/* This is a
multi-line comment */
```

Data types

Basics

JavaScript provides seven different data types:

Data Types	Examples
`undefined`	A variable that has not been assigned a value is of type `undefined`.
`null`	no value.
`string`	`'a', 'aa', 'aaa', 'Hello!', '11 cats'`
`number`	`12, -1, 0.4`
`boolean`	`true, false`
`object`	A collection of properties.
`symbol`	Represents a unique identifier.

Variables

Examples

```
// declare a variable
let ourName

// store values
myNumber = 5
myString = 'myLet'

// declare variables with the assignment operator
let myNum = 0

// add, subtract, multiply and divide numbers
myLet = 5 + 10 // 15
myLet = 12 - 6 // 6
myLet = 13 * 13 // 169
myLet = 16 / 2 // 8

// increment and decrement numbers
i++ // the equivalent of i = i + 1
i-- // the equivalent of i = i - 1;

// decimals
let ourDecimal = 5.7 // float
```

ES6 var, let and const

- Unlike `var`, `let` throws an error if you declare the same variable twice.
- Variables declared with `let` inside a block, statement, or expression, its scope is limited to that block, statement, or expression.
- Variables declared with `const` are read-only and cannot be reassigned.
- Objects (including arrays and functions) assigned to a variable using `const` are still mutable and only prevents the reassignment of the variable identifier.
  To ensure your data doesn't change, JavaScript provides a function Object.freeze to prevent data mutation.

```
let obj = {
  name: 'FreeCodeCamp',
  review: 'Awesome',
}

Object.freeze(obj)
obj.review = 'bad' //will be ignored. Mutation not allowed
obj.newProp = 'Test' // will be ignored. Mutation not allowed
console.log(obj)
// { name: "FreeCodeCamp", review:"Awesome"}
```

Strings

Examples

```
// escape literal quotes
const sampleStr = 'Alan said, "Peter is learning JavaScript".'
// this prints: Alan said, "Peter is learning JavaScript".

// concatenating strings
const ourStr = 'I come first. ' + 'I come second.'

// concatenating strings with +=
const ourStr = 'I come first. '
ourStr += 'I come second.'

// constructing strings with variables
const ourName = 'freeCodeCamp'
const ourStr = 'Hello, our name is ' + ourName + ', how are you?'

// appending variables to strings
const anAdjective = 'awesome!'
const ourStr = 'freeCodeCamp is '
ourStr += anAdjective
```

Escape sequences

Code	Output
`\'`	single quote (')
`\"`	double quote (")
`\\`	backslash (\)
`\n`	newline
`\r`	carriage return
`\t`	Tab
`\b`	backspace
`\f`	form feed

The length of a string

```
'Alan Peter'.length // 10
```

Split and Join

```
let str = 'a string'
let splittedStr = str.split('')
// [ 'a', ' ', 's', 't', 'r', 'i', 'n', 'g' ]

let joinedStr = splittedStr.join('')
// a string
```

Index of a String

```
//first element has an index of 0
const firstLetterOfFirstName = ''
const firstName = 'Ada'
firstLetterOfFirstName = firstName[0] // A

// find the las character of a string
const firstName = 'Ada'
const lastLetterOfFirstName = firstName[firstName.length - 1] // a
```

ES6 Template Literals

```
const person = {
  name: 'Zodiac Hasbro',
  age: 56,
}

// Template literal with multi-line and string interpolation
const greeting = `Hello, my name is ${person.name}!
I am ${person.age} years old.`

console.log(greeting)
// Hello, my name is Zodiac Hasbro!
// I am 56 years old.
```

Arrays

Example

```
const sandwich = ['peanut butter', 'jelly', 'bread'][
  // nested arrays
  (['Bulls', 23], ['White Sox', 45])
]
```

Index of an array

```
const ourArray = [50, 60, 70]
consy ourData = ourArray[0] // equals 50

// modify an array with indexes
consy ourArray = [50, 40, 30]
ourArray[0] = 15 // equals [15,40,30]

// access multi-dimensional arrays with indexes
const arr = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
  [[10, 11, 12], 13, 14],
]
arr[3] // [[10,11,12], 13, 14]
arr[3][0] // [10,11,12]
arr[3][0][1] // 11
```

Manipulate arrays with reverse, push, pop, shift and unshift

```
// reverse an array
;[1, 'two', 3].reverse() // [ 3, 'two', 1 ]

// push() to append data to the end of an array
const arr = [1, 2, 3]
arr.push(4) // arr is now [1,2,3,4]

// pop() to "pop" a value off of the end of an array
const threeArr = [1, 4, 6]
const oneDown = threeArr.pop()
console.log(oneDown) // Returns 6
console.log(threeArr) // Returns [1, 4]

// shift() removes the first element of an array
const ourArray = [1, 2, [3]]
const removedFromOurArray = ourArray.shift()
// removedFromOurArray now equals 1 and ourArray now equals [2, [3]].

// unshift() adds the element at the beginning of the array
const ourArray = ['Stimpson', 'J', 'cat']
ourArray.shift() // ourArray now equals ["J", "cat"]
ourArray.unshift('Happy') // ourArray now equals ["Happy", "J", "cat"]
```

Remove any element with splice

```
// first parameter is the index, the second indicates the number of
elements to delete.
let array = ['today', 'was', 'not', 'so', 'great']
array.splice(2, 2)
// remove 2 elements beginning with the 3rd element
// array now equals ['today', 'was', 'great']

// also returns a new array containing the value of the removed elements
let array = ['I', 'am', 'feeling', 'really', 'happy']
let newArray = array.splice(3, 2)
// newArray equals ['really', 'happy']

// the third parameter, represents one or more elements, let us add them
function colorChange(arr, index, newColor) {
  arr.splice(index, 1, newColor)
  return arr
}
let colorScheme = ['#878787', '#a08794', '#bb7e8c', '#c9b6be', '#d1becf']
colorScheme = colorChange(colorScheme, 2, '#332327')
// we have removed '#bb7e8c' and added '#332327' in its place
// colorScheme now equals ['#878787', '#a08794', '#332327', '#c9b6be',
'#d1becf']
```

Copy an array with slice

```
// Copies a given number of elements to a new array and leaves the original
array untouched
let weatherConditions = ['rain', 'snow', 'sleet', 'hail', 'clear']
let todaysWeather = weatherConditions.slice(1, 3)
// todaysWeather equals ['snow', 'sleet'];
// weatherConditions still equals ['rain', 'snow', 'sleet', 'hail',
'clear']
```

indexOf

```
let fruits = ['apples', 'pears', 'oranges', 'peaches', 'pears']
fruits.indexOf('dates') // -1
fruits.indexOf('oranges') // 2
fruits.indexOf('pears') // 1, the first index at which the element exists
```

Accessing Nested Arrays

```
const ourPets = [
  {
    animalType: 'cat',
    names: ['Meowzer', 'Fluffy', 'Kit-Cat'],
  },
  {
    animalType: 'dog',
    names: ['Spot', 'Bowser', 'Frankie'],
  },
]
ourPets[0].names[1] // "Fluffy"
ourPets[1].names[0] // "Spot"
```

ES6 Includes to Determine if an Array Contains an Element

```
let fruits = ['Banana', 'Orange', 'Apple', 'Mango']
fruits.includes('Mango') // true
```

ES6 The Spread Operator

```
// ...arr returns an unpacked array. In other words, it spreads the array.
const arr = [6, 89, 3, 45]
const maximus = Math.max(...arr) // 89

// [...new Set(arr)] = unique value array
const arr = [1, 2, 2, 3, 3, 4, 5, 5]
const uniq = [...new Set(arr)] // [1,2,3,4,5]

// copy an array
let thisArray = [true, true, undefined, false, null]
let thatArray = [...thisArray]
// thatArray equals [true, true, undefined, false, null]
// thisArray remains unchanged, and is identical to thatArray

// combine arrays
let thisArray = ['sage', 'rosemary', 'parsley', 'thyme']
let thatArray = ['basil', 'cilantro', ...thisArray, 'coriander']
// thatArray now equals ['basil', 'cilantro', 'sage', 'rosemary',
'parsley', 'thyme', 'coriander']
```

ES6 Destructuring Arrays to Assign Variables

```
const [a, b] = [1, 2, 3, 4, 5, 6]
console.log(a, b) // 1, 2

// it can access any value by using commas to reach the desired index
const [a, b, , , c] = [1, 2, 3, 4, 5, 6]
console.log(a, b, c) // 1, 2, 5

// to collect the rest of the elements into a separate array.
const [a, b, ...arr] = [1, 2, 3, 4, 5, 7]
console.log(a, b) // 1, 2
console.log(arr) // [3, 4, 5, 7]
```

Array Methods

Map

```
let watchList = [
  {
    Title: 'Inception',
    imdbRating: '8.8',
    Type: 'movie',
  },
  {
    Title: 'Interstellar',
    imdbRating: '8.6',
    Type: 'movie',
  },
  {
```

```
    Title: 'The Dark Knight',
    imdbRating: '9.0',
    Type: 'movie',
  },
  {
    Title: 'Batman Begins',
    imdbRating: '7.9',
    Type: 'movie',
  },
]

const rating = watchList.map(function (movie) {
  return { title: movie.Title, rating: movie.imdbRating }
})
/* [ { title: 'Inception', rating: '8.8' },
  { title: 'Interstellar', rating: '8.6' },
  { title: 'The Dark Knight', rating: '9.0' },
  { title: 'Batman Begins', rating: '7.9' } ] */

// or...
const rating = watchList.map((movie) => ({
  title: movie.Title,
  rating: movie.imdbRating,
}))
/* [ { title: 'Inception', rating: '8.8' },
  { title: 'Interstellar', rating: '8.6' },
  { title: 'The Dark Knight', rating: '9.0' },
  { title: 'Batman Begins', rating: '7.9' } ] */
```

JavaScript Objects

Example

```
let cat = {
  name: 'Whiskers',
  legs: 4,
  tails: 1,
  enemies: ['Water', 'Dogs'],
}
```

Accessing Objects Properties

Accessing with dot (.) notation

```
let myObj = {
  prop1: 'val1',
  prop2: 'val2',
}

const prop1val = myObj.prop1 // val1
const prop2val = myObj.prop2 // val2

or

myObj.prop1
myObj.prop2
```

Accessing with bracket (`[]`) notation

```
let myObj = {
  'Space Name': 'Kirk',
  'More Space': 'Spock',
  NoSpace: 'USS Enterprise',
}

myObj['Space Name'] // Kirk
myObj['More Space'] // Spock
myObj['NoSpace'] // USS Enterprise
```

Accessing with variables

```
let dogs = {
  Fido: 'Mutt',
  Hunter: 'Doberman',
  Snoopie: 'Beagle',
}

const myDog = 'Hunter'
const myBreed = dogs[myDog]
console.log(myBreed)  // "Doberman"
```

Accessing and modifying Nested Objects

```
let userActivity = {
  id: 23894201352,
  date: 'January 1, 2017',
  data: {
    totalUsers: 51,
    online: 42,
  },
}

userActivity.data.online = 45 // or
userActivity['data'].online = 45 // or
userActivity['data']['online'] = 45
```

Creating an array from the keys of an object

```
let users = {
  Alan: {
    age: 27,
    online: false,
  },
  Jeff: {
    age: 32,
    online: true,
  },
  Sarah: {
    age: 48,
    online: false,
  },
  Ryan: {
    age: 19,
    online: true,
  },
}

function getArrayOfUsers(obj) {
  let arr = []
```

```
  for (let key in obj) {
    arr.push(key)
  }
  return arr
}
```

Modifying Objects Properties

```
// Updating object properties
let ourDog = {
  name: 'Camper',
  legs: 4,
  tails: 1,
  friends: ['everything!'],
}

ourDog.name = 'Happy Camper' // or
ourDog['name'] = 'Happy Camper'

// add new properties
ourDog.bark = 'bow-wow' // or
ourDog['bark'] = 'bow-wow'

// delete properties
delete ourDog.bark
```

Objects for Lookups

```
let alpha = {
  1:"Z",
  2:"Y",
  3:"X",
  4:"W",
  ...
  24:"C",
  25:"B",
  26:"A"
};
alpha[2]; // "Y"
alpha[24]; // "C"

const value = 2;
alpha[value]; // "Y"
```

Test Object Properties

```
let myObj = {
  top: 'hat',
  bottom: 'pants',
}

myObj.hasOwnProperty('top') // true
myObj.hasOwnProperty('middle') // false
```

Accessing Nested Objects

```
let ourStorage = {
  desk: {
    drawer: 'stapler',
  },
  cabinet: {
    'top drawer': {
      folder1: 'a file',
      folder2: 'secrets',
    },
    'bottom drawer': 'soda',
  },
}

ourStorage.cabinet['top drawer'].folder2 // "secrets"
ourStorage.desk.drawer // "stapler"
```

ES6 Destructuring Variables from Objects

```
// Consider the following ES5 code
var voxel = { x: 3.6, y: 7.4, z: 6.54 }
var x = voxel.x // x = 3.6
var y = voxel.y // y = 7.4
var z = voxel.z // z = 6.54

// the same assignment statement with ES6 destructuring syntax
const { x, y, z } = voxel // x = 3.6, y = 7.4, z = 6.54

// to store the values of voxel.x into a, voxel.y into b, and voxel.z into
c, you have that freedom as well
const { x: a, y: b, z: c } = voxel // a = 3.6, b = 7.4, c = 6.54

// Destructuring Variables from Nested Objects
const a = {
  start: { x: 5, y: 6 },
  end: { x: 6, y: -9 },
}

const {
  start: { x: startX, y: startY },
} = a

console.log(startX, startY) // 5, 6
```

ES6 Destructuring to Pass an Object as a Function's Parameters

```
// destructure the object in a function argument itself.
const profileUpdate = (profileData) => {
  const { name, age, nationality, location } = profileData
  // do something with these variables
}

// this can also be done in-place:
const profileUpdate = ({ name, age, nationality, location }) => {
  /* do something with these fields */
}
```

ES6 Object Literal Declarations Using Simple Fields

```
const getMousePosition = (x, y) => ({
  x: x,
  y: y,
})

// the same function rewritten to use this new syntax:
const getMousePosition = (x, y) => ({ x, y })
```

Booleans

Basics

Booleans may only be one of two values: true or false. They are basically little on-off switches, where true is "on" and false is "off". These two states are mutually exclusive.

```
true
false
```

If Else Statements

If Statement

```
if (condition is true) {
  statement is executed
}
```

Else Statement

```
if (num > 10) {
  return 'Bigger than 10'
} else {
  return '10 or Less'
}
```

Else if statement

```
if (num > 15) {
  return 'Bigger than 15'
} else if (num < 5) {
  return 'Smaller than 5'
} else {
  return 'Between 5 and 15'
}
```

Conditional (Ternary) Operator

```
// this if statement...
function findGreater(a, b) {
  if (a > b) {
    return 'a is greater'
```

```
  } else {
    return 'b is greater'
  }
}

// is equivalent to this ternary operator
function findGreater(a, b) {
  return a > b ? 'a is greater' : 'b is greater'
}
```

Multiple Conditional (Ternary) Operators

```
// this if statement...
function findGreaterOrEqual(a, b) {
  if (a === b) {
    return 'a and b are equal'
  } else if (a > b) {
    return 'a is greater'
  } else {
    return 'b is greater'
  }
}

// is equivalent to this ternary operator
function findGreaterOrEqual(a, b) {
  return a === b ? 'a and b are equal' : a > b ? 'a is greater' : 'b is
greater'
}
```

Switch Statement

Example

```
switch(num) {
  case value1:
    statement1;
    break;
  case value2:
    statement2;
    break;
...
  case valueN:
    statementN;
    break;
}
```

Default Switch Statement

```
switch (num) {
  case value1:
    statement1;
    break;
  case value2:
    statement2;
    break;
...
  default:
    defaultStatement;
```

```
    break;
}
```

Multiple Options with Switch Statement

```
switch (val) {
  case 1:
  case 2:
  case 3:
    result = '1, 2, or 3'
    break
  case 4:
    result = '4 alone'
}
```

Comparison Operators

Basics

Operator	Meaning
==	Equality
===	Strict Equality
!=	Inequality
!==	Strict Inequality
>	Greater Than
>=	Greater or Equal Than
<	Less Than
<=	Less or Equal Than
&&	And

`

While Loops

While Loop

```
let ourArray = []
let i = 0
while (i < 5) {
  ourArray.push(i)
  i++
}
```

Do...While Loops

```
let ourArray = []
let i = 0
do {
  ourArray.push(i)
  i++
} while (i < 5)
```

For Loops

For Loop

```
const ourArray = []
let i = 0
while (i < 5) {
  ourArray.push(i)
  i++
}

// Count Backwards With a For Loop
const ourArray = []
for (let i = 10; i > 0; i -= 2) {
  ourArray.push(i)
}

// Iterate Through an Array
const arr = [10, 9, 8, 7, 6]
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i])
}

// Nested for loops
let arr = [
  [1, 2],
  [3, 4],
  [5, 6],
]
for (let i = 0; i < arr.length; i++) {
  for (var j = 0; j < arr[i].length; j++) {
    console.log(arr[i][j])
  }
}
```

ES6 for-of

```
for (let value of myArray) {
  console.log(value)
}
```

Functions

Example

```
function functionName() {
  console.log('Hello World')
}

functionName() // call the function
```

Function Arguments

```
function ourFunctionWithArgs(a, b) {
  console.log(a - b)
}
ourFunctionWithArgs(10, 5) // 5
```

Return Statement

```
function plusThree(num) {
  return num + 3
}
const answer = plusThree(5) // 8
```

Immediately Invoked Function Expression or IIFE

```
;(function () {
  console.log('A cozy nest is ready')
})()
```

ES6 Arrow Functions

```
const myFunc = function () {
  const myVar = 'value'
  return myVar
}

// can be rewritten like this
const myFunc = () => {
  const myVar = 'value'
  return myVar
}

// and if there is no function body, and only a return value
const myFunc = () => 'value'

// to pass parameters to an arrow function
const doubler = (item) => item * 2
```

ES6 Higher Order Arrow Functions

```
FBPosts.filter(function (post) {
  return post.thumbnail !== null && post.shares > 100 && post.likes > 500
})

// the previous function can be rewritten like this
FBPosts.filter(
  (post) => post.thumbnail !== null && post.shares > 100 && post.likes >
500
)
```

ES6 Rest Operator with Function Parameters

With the rest operator, you can create functions that take a variable number of arguments. These arguments are stored in an array that can be accessed later from inside the function.

```
function howMany(...args) {
  return 'You have passed ' + args.length + ' arguments.'
}
console.log(howMany(0, 1, 2)) // You have passed 3 arguments
console.log(howMany('string', null, [1, 2, 3], {})) // You have passed 4
arguments.
```

ES6 Declarative Functions within Objects

```
// When defining functions within objects in ES5, we have to use the
keyword function
const person = {
  name: 'Taylor',
  sayHello: function () {
    return `Hello! My name is ${this.name}.`
  },
}

// With ES6, You can remove the function keyword and colon
const person = {
  name: 'Taylor',
  sayHello() {
    return `Hello! My name is ${this.name}.`
  },
}
```

Regular Expressions

Resume Table

Character	Description
`\`	Escapes a special character.
`` ` ``	`` ` ``
`i`	This flag is used to ignore upper and lowercase. `/ignorecase/i`.
`g`	Search or extract a pattern more than once.
`.`	The wildcard character `.` will match any character except new lines.
`[]`	Allow you to define the characters to match. `/b[au]g/` will match "bag", "bug" but not "bog".
`[a-z]`	Match all the characters between a and z.
`[1-9]`	Match all the numbers between 1 and 9.
`[a-z1-9]`	Match all the character between a and z, and the numbers between 1 and 9.
`[^]`	Match the characters not in the set. `[^a-e]` match all other characters except A, B, C, D, and E.

Character	Description
`+`	Match 1 or more occurrences of the previous character in a row.
`*`	Match 0 or more occurrences of the previous character.
`?`	Match 0 or 1 occurrence of the previous character. Useful for Lazy matching.
`^`	Search for patterns at the beginning of strings.
`$`	Search for patterns at the end of a string.
`\w`	Equal to `[A-Za-z0-9_]`. Matches upper, lowercase, numbers the and underscore character (-).
`\W`	Matches any nonword character. Equivalent to `[^a-za-z0-9_]`.
`\d`	Equal to `[0-9]`. Match one digit.
`\D`	Equal to `[^0-9]`. Match one non digit.
`\s`	Match a whitespace.
`\S`	Match everything except whitespace.
`a{2,5}`	Match the letter `a` between 3 and 5 times.
`a{2,}`	Specify only the lower number of matches.
`a{5}`	Specify the exact number of matches.
`(...)`	Specify a group that can be acceded with number (from 1)

Regex Methods

Method	Description
`test()`	Returns true or false if the pattern match a string or not.
`match()`	Extract the actual matches found.
`replace()`	Search and replace text in a string .

Examples

```
// test method returns true or false if the pattern match a string or not
const myString = 'Hello, World!'
const myRegex = /Hello/
const result = myRegex.test(myString)

// extract the matches of a regex with the match method
const extractStr = "Extract the word 'coding' from this string."
const codingRegex = /coding/
const result = extractStr.match(codingRegex)

// Search and replace
const wrongText = 'The sky is silver.'
const silverRegex = /silver/
wrongText.replace(silverRegex, 'blue') // Returns "The sky is blue."

// search for multiple patterns using the alternation or OR operator: |
const petString = 'James has a pet cat.'
const petRegex = /dog|cat|bird|fish/
const result = petRegex.test(petString)

// ignore upper or lowercase
const myString = 'freeCodeCamp'
const fccRegex = /freeCodeCamp/i // flag i
const result = fccRegex.test(myString)
```

```
// Search or extract a pattern more than once
const twinkleStar = 'Twinkle, twinkle, little star'
const starRegex = /Twinkle/gi // a regex can have multiple flags
const result = twinkleStar.match(starRegex)

// The wildcard character . will match any character except new lines.
const exampleStr = "Const's have fun with regular expressions!"
const unRegex = /.un/
const result = unRegex.test(exampleStr)

// define the characters to match, in this example all the vowels in
quoteSample
const quoteSample =
  'Beware of bugs in the above code; I have only proved it correct, not
tried it.'
const vowelRegex = /[aeiou]/gi
const result = quoteSample.match(vowelRegex)

// Match all the characters in quoteSample (between a and z)
const quoteSample = 'The quick brown fox jumps over the lazy dog.'
const alphabetRegex = /[a-z]/gi
const result = quoteSample.match(alphabetRegex)

// Match all the character between two characters and numbers
const quoteSample = 'Blueberry 3.141592653s are delicious.'
const myRegex = /[h-s2-6]/gi
const result = quoteSample.match(myRegex)

// Match all that is not a number or a vowel
const quoteSample = '3 blind mice.'
const myRegex = /[^aeiou0-9]/gi
const result = quoteSample.match(myRegex)

// Match 1 or more occurrences of the previous character (* for 0 or more)
const difficultSpelling = 'Mississippi'
const myRegex = /s+/g
const result = difficultSpelling.match(myRegex)

// ? Match 0 or 1 occurrence of the previous character. Useful for Lazy
matching
let text = 'titanic'
const myRegex = /t[a-z]*?i/
const result = text.match(myRegex)

// Search for patterns at the beginning of strings
const rickyAndCal = 'Cal and Ricky both like racing.'
const calRegex = /^Cal/
const result = calRegex.test(rickyAndCal)

// Search for patterns at the end of a string
const caboose = 'The last car on a train is the caboose'
const lastRegex = /caboose$/
const result = lastRegex.test(caboose)

// \w is equal to [A-Za-z0-9_]
const quoteSample = 'The five boxing wizards jump quickly.'
const alphabetRegexV2 = /\w/g
const result = quoteSample.match(alphabetRegexV2).length

// Match only 3 to 6 constter h's in the word "Oh no"
```

```
const ohStr = 'Ohhh no'
const ohRegex = /Oh{3,6} no/
const result = ohRegex.test(ohStr)

// Match both the American English (favorite) and the British English
(favorite) version of the word
const favWord = 'favorite'
const favRegex = /favou?rite/
const result = favRegex.test(favWord)

// Groups () const you reuse patterns
const repeatNum = '42 42 42'
const reRegex = /^(\d+)\s\1\s\1$/ // every 1 represent the group (\d+)
const result = reRegex.test(repeatNum)

// Remove all the spaces at the beginning an end of a string
const hello = '   Hello, World!  '
const wsRegex = /^\s+(.*\S)\s+$/
const result = hello.replace(wsRegex, '$1') // returns 'Hello, World!'
```

Object Oriented

Example

```
let duck = {
  name: 'Aflac',
  numLegs: 2,
  sayName: function () {
    return 'The name of this duck is ' + this.name + '.'
  },
}

duck.sayName() // Returns "The name of this duck is Aflac."
```

Constructors and New Objects

Constructors follow a few conventions:

- Constructors are defined with a capitalized name to distinguish them from other functions that are not constructors.
- Constructors use the keyword this to set properties of the object they will create. Inside the constructor, this refers to the new object it will create.
- Constructors define properties and behaviors instead of returning a value as other functions might.

```
// constructor
function Bird(name, color) {
  this.name = name
  this.color = color
}

// create a new instance of Bird
const cardinal = new Bird('Bruce', 'red')
const duck = new Bird('Donald', 'blue')
```

```
// access and modify blueBird object
cardinal.name // Bruce
cardinal.color // red
cardinal.color = green
cardinal.color // green

// check if an object is an instance of a constructor
cardinal instanceof Bird // true
crow instanceof Bird // false

// check an objects own (name, color, numLegs) properties
cardinal.hasOwnProperty('color') // true
cardinal.hasOwnProperty('age') // false

//check an objects properties with the constructor property
cardinal.constructor === Bird // true

// use constructor.prototype to add new properties to object constructors
Bird.prototype.cute = true
cardinal.cute // true
crow.cute // true

// add more than one property and method to a constructor
Bird.prototype = {
  constructor: Bird, // specify the constructor
  numLegs: 2, // new property

  eat: function () {
    // new method
    console.log('nom nom nom')
  },

  describe: function () {
    // new method
    console.log('My name is ' + this.name)
  },
}

const chicken = new Bird('Dinner', 'brown')
chicken.numLegs // 2
chicken.eat() // nom nom nom
chicken.describe() // My name is Dinner
```

Inheritance

```
function Animal() {}

Animal.prototype = {
  constructor: Animal,
  eat: function () {
    console.log('nom nom nom')
  },
}

function Cat(name) {
  this.name = name
}
```

```
// make the Cat constructor inherit the eat function from Animal
Cat.prototype = Object.create(Animal.prototype)

const myCat = new Cat('charles')
myCat.eat() // nom nom nom
```

Add methods after Inheritance and override them

```

function Animal() {}
Animal.prototype.eat = function () {
  console.log('nom nom nom')
}

// Dog constructor
function Dog() {}

// make the Gog constructor inherit the eat function from Animal
Dog.prototype = Object.create(Animal.prototype)
Dog.prototype.constructor = Dog

Dog.prototype.bark = function () {
  console.log('wof wof!')
}

// the new object will have both, the inherited eat() and its own bark()
method
const beagle = new Dog()
beagle.eat() // "nom nom nom"
beagle.bark() // "Woof!"

// override an inherited method
Dog.prototype.eat = function () {
  return 'nice meeeeat!'
}

let doberman = new Dog()
doberman.eat() // nice meeeeat!
```

Mixins

A mixin allows unrelated objects to use a collection of functions.

```

let bird = {
  name: 'Donald',
  numLegs: 2,
}

let boat = {
  name: 'Warrior',
  type: 'race-boat',
}

// this mixin contain the glide method
const glideMixin = function (obj) {
  obj.glide = function () {
    console.log('gliding...')
  }
}
```

```
// the object is passed to the mixin and the glide method is assigned
glideMixin(bird)
glideMixin(boat)

bird.glide() // "gliding..."
boat.glide() // "gliding..."
```

Closures to Protect Properties

In JavaScript, a function always has access to the context in which it was created. This is called closure. Now, the property can only be accessed and changed by methods also within the constructor function. In JavaScript, this is called closure.

```
function Bird() {
  // instead of this.hatchedEgg...
  const hatchedEgg = 10 // private property

  this.getHatchedEggCount = function () {
    // publicly available method that a bird object can use
    return hatchedEgg
  }
}

const ducky = new Bird()
ducky.hatchedEgg = 2 // nothing happens
ducky.getHatchedEggCount // 10
```

Modules

An immediately invoked function expression (IIFE) is often used to group related functionality into a single object or module.

```
let funModule = (function () {
  return {
    isCuteMixin: function (obj) {
      obj.isCute = function () {
        return true
      }
    },

    singMixin: function (obj) {
      obj.sing = function () {
        console.log('Singing to an awesome tune')
      }
    },
  }
})()

function Dog() {}
let goodBoy = new Dog()

// assign the singMixin method to the goodBoy object
funModule.singMixin(goodBoy)
goodBoy.sing() // Singing to an awesome tune
```

ES6 Object Oriented

Basics

ES6 provides a new syntax to help create objects, the keyword `class`.The class syntax is just a syntax, and not a full-fledged class based implementation of object oriented paradigm, unlike in languages like Java, or Python, or Ruby etc.

ES6 Classes

```
class Book {
  constructor(title, author, year) {
    this.title = title
    this.author = author
    this.year = year
  }

  getSummary() {
    return `${this.title} was written by ${this.author} in ${this.year}`
  }

  getAge() {
    const years = new Date().getFullYear() - this.year
    return `${this.title} is ${years} years old`
  }
}

book = new Book('Book One', 'John Doe', 2016)
book.getSummary() // Book One was written by John Doe in 2016
book.getAge() // Book One is 3 years old
```

ES6 getters and setters

```
class Book {
  constructor(author) {
    this._author = author
  }
  // getter
  get writer() {
    return this._author
  }
  // setter
  set writer(updatedAuthor) {
    this._author = updatedAuthor
  }
}
const lol = new Book('anonymous')
console.log(lol.writer) // anonymous
lol.writer = 'wut'
console.log(lol.writer) // wut
```

ES6 Statics Methods

Static methods allow using methods without instantiating an object

```
class Book {
  constructor(title, author, year) {
    this.title = title
    this.author = author
    this.year = year
  }

  static sayHi() {
    return 'Hi!'
  }
}

Book.sayHi() // Hi!
```

ES6 Inheritance

```
class Book {
  constructor(title, author, year) {
    this.title = title
    this.author = author
    this.year = year
  }

  getSummary() {
    return `${this.title} was written by ${this.author} in ${this.year}`
  }
}

class Magazine extends Book {
  constructor(title, author, year, month) {
    super(title, author, year)
    this.month = month
  }

  sayHi() {
    return 'Hi!'
  }
}

mag = new Magazine('Mag', 'People', 2019, 'jan')
mag.getSummary() // Mag was written by People in 2019
mag.sayHi() // Hi!
```

ES6 import and export

Basics

The lessons in this section handle non-browser features. import won't work on a browser directly. However, we can use various tools to create code out of this to make it work in browser.

import

```
// we can choose which parts of a module or file to load into a given file.
import { function } from 'file_path'
// We can also import variables the same way!

// Import Everything from a File
import * as name_of_your_choice from 'file_path'
```

export

In order for `import` to work, though, we must first `export` the functions or variables we need. Like import, export is a non-browser feature.

```
const capitalizeString = (string) => {
  return string.charAt(0).toUpperCase() + string.slice(1)
}
export { capitalizeString } //How to export functions.
export const foo = 'bar' //How to export variables.

// Alternatively, if you would like to compact all your export statements
into one line, you can take this approach
const capitalizeString = (string) => {
  return string.charAt(0).toUpperCase() + string.slice(1)
}
const foo = 'bar'
export { capitalizeString, foo }

// use export default if only one value is being exported from a file.
// It is also used to create a fallback value for a file or module
export default function add(x, y) {
  return x + y
}
// and to import
import add from 'math_functions'
add(5, 4) //Will return 9
```

Async/Await

JavaScript is a synchronous language!

This means only one operation can be carried out at a time. Also There are many ways JavaScript provides us with the ability to make it behave like an asynchronous language. One of them is with the Async-Await clause.

What is async-await?

Async and Await are extensions of promises. First you need to clear the concepts of `Promises(Future)` before coming to async/await.

Async

Async functions enable us to write promise based code as if it were synchronous, but without blocking the execution thread. It operates asynchronously via the event-loop. Async functions will always return a value. Using `async` simply implies that a promise will be returned, and if a `promise` is not returned, JavaScript automatically wraps it in a resolved promise with its value.

```
async function firstAsync() {
  return 23
}

firstAsync().then(alert) // 23
```

Running the above code gives the alert output as 23, it means that a `promise` was returned, otherwise the `.then()` method simply would not be possible.

Await

The await operator is used to wait for a Promise to complete. It can be used inside an `async` block only. The keyword `await` makes JavaScript wait until the promise returns a result. It has to be noted that it only makes the `async` function block wait and not the whole program execution.
The code block below shows the use of Async Await together.

```
async function firstAsync() {
    let promise = new Promise((res, rej) => {
        setTimeout(() => res("Now it's done!"), 1000)
    });

    // wait until the promise returns us a value
    let result = await promise;

    // "Now it's done!"
    alert(result);
    }
};
firstAsync();
```

Things to remember when using Async Await

We can’t use the `await` keyword inside of regular functions.

```
function firstAsync() {
  let promise = Promise.resolve(10);
  let result = await promise; // Syntax error
}
```

To make the above function work properly, we need to add `async` before the function `firstAsync();`

Async Await makes execution sequential

Not necessarily a bad thing, but having paralleled execution is much much faster.

For example:

```
async function sequence() {
  await promise1(50) // Wait 50ms…
  await promise2(50) // …then wait another 50ms.
  return 'done!'
}
```

The above takes 100ms to complete, not a huge amount of time but still slow.

This is because it is happening in sequence. Two promises are returned, both of which takes 50ms to complete. The second promise executes only after the first promise is resolved. This is not a good practice, as large requests can be very time consuming. We have to make the execution parallel.

That can be achieved by using `Promise.all()` .
According to MDN:

The `Promise.all()` method returns a single `Promise` that resolves when all of the promises passed as an iterable have resolved or when the iterable contains > no promises. It rejects with the reason of the first promise that rejects.

Promise.all()

```
async function sequence() {
  await Promise.all([promise1(), promise2()])
  return 'done!'
}
```

The `promise.all()` function resolves when all the promises inside the iterable have been resolved and then returns the result.

Another method:

```
async function parallel() {
  // Start a 500ms timer asynchronously…
  const wait1 = promise1(50)
  // …meaning this timer happens in parallel.
  const wait2 = promise2(50)

  // Wait 50ms for the first timer…
  await wait1

  // …by which time this timer has already finished.
  await wait2

  return 'done!'
}
```

Async Await is very powerful but they come with caveats. But if we use them properly, they help to make our code very readable and efficient.