JavaScript

Codeacademy

Introduction to JavaScript

Console object

The console is a panel that displays important messages, like errors, for developers. How to write in the console?

`console.log('text');`

Comments

Line comments are done using double slash: `// comment`
Block comments are done using `/* comment */`

Data Types

Number: Any number, including numbers with decimals: `4`, `8`, `1516`, `23.42`.

String: Any grouping of characters on your keyboard (letters, numbers, spaces, symbols, etc.) surrounded by single quotes: `' ... '` or double quotes `" ... "`. Though we prefer single quotes. Some people like to think of string as a fancy word for text.

Boolean: This data type only has two possible values— either `true` or `false` (without quotes). It's helpful to think of booleans as on and off switches or as the answers to a "yes" or "no" question.

Null: This data type represents the intentional absence of a value, and is represented by the keyword `null` (without quotes).

Undefined: This data type is denoted by the keyword `undefined` (without quotes). It also represents the absence of a value though it has a different use than `null`.

Symbol: A newer feature to the language, symbols are unique identifiers, useful in more complex coding. No need to worry about these for now.

Object: Collections of related data.

Arithmetic Operators

Add: `+`
Subtract: `-`

Multiply: `*`
Divide: `/`
Remainder: `%`

String Concatenation

Use `+`

String Properties

`length`

String Methods

`toUpperCase`: returns the string in capital letters
`trim`: Remove white spaces before and after the string
`startsWith`: return true if the argument is found at the beginning of the string

Built-in Math Object

`Math`: mathematical functions in an object

Methods

`random`: Prints a random number between 0 and 1
`floor`: makes a decimal number, and rounds down to the nearest whole number
`ceil`: rounds a number up to the next largest integer

Built-in Number Object

Methods

`isInteger`: a

Variables

Variables can be declared using one of the three keywords: var, let, and const.

`var` allows the type of variable to be changed. `let` does not allow that to happen. `const` declares a constant.

Mathematical Assignment Operators

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

String Interpolation

To concatenate text with variables we use: `console.log(`text ${variable}.`);` Note that the single quote used is different.

typeof operator

The typeof operator checks the value to its right and returns, or passes back, a string of the data type: `typeof unknown1`

Conditional Statements

If Statement

```
If (conditional) {
   executionBlock;
} else {
   executionBlock;
}
```

Comparison Operators

Less than: `<`
Greater than: `>`
Less than or equal to: `<=`
Greater than or equal to: `>=`
Is equal to: `===`
Is not equal to: `!==`

Logical Operators

The and operator (`&&`)
The or operator (`||`)
The not operator, otherwise known as the bang operator (`!`)

Truthy and Falsy

Falsy values:
`0`
Empty strings like `""` or `''`
`null` which represent when there is no value at all
`undefined` which represent when a declared variable lacks a value
`NaN`, or Not a Number

Everything that is not falsy is truthy.

Short circuit evaluation

```
result = testedVar || 'defaultValue';
```

If the `testedVar` is empty, then the `defaultValue` is used instead.

Ternary Operator

It is a substitute for the `if else` statement. The format is:
`testedCondition ? positiveExecution : negativeExecution`

Else If Statement

```
If (conditional) {
   executionBlock;
} else if {
   executionBlock;
} else {
   executionBlock;
}
```

The switch keyword

```
switch (test) {
```

```
  case 'first':
    blockExecutionCase1;
    break;
  case 'second':
    blockExecutionCase2;
    break;
  case third':
        blockExecutionCase3;
    break;
  default:
        blockExecutionDefault;;
    break;
}
```

Functions

Traditional

```
function functionName(parameter1, parameter2) {
  codeBlock;
  return value;
}
```

We can have a default value for the parameter:

```
function functionName(parameter1 = 'value', parameter2) {
  body;
}
```

Function Expression

```
const functionName = function(parameter1, parameter2) {
  codeBlock;
  return value;
}
```

Arrow Functions

No parameter:

```
const functionName = () => {};
```

One parameter:

```
const functionName = parameter => {};
```

Two parameters:

```
const functionName = (parameter1, parameter2) => {
  codeBlock;
  return value;
};
```

One block:

```
const functionName = parameter => expression; // the return is expression
```

Multiline:

```
const functionName = parameter => {
  codeBlock;
  return value;
}
```

Scope

Here we saw the concept of scope where a variable exists, and also blocks, which limit scope.

Arrays

Arrays in Javascript is declared in []: `let arr = [];`
Accessing items that are not within the array scope returns `undefined`.\
Elements within arrays declared as `const` are still mutable.

Methods & Properties

The `length` property returns the length of an array.
`push` adds elements to an array.
`pop` removes the last item of an array.
join
`slice(start, finish)` returns items based on an array from `start` to `finish - 1`. It is a shallow copy.
splice

`shift` returns the first element of an array and modifies the array
`unshift` add elements to the beginning of an array modifying the array
concat
`indexOf(element)` returns the index of the element

Loops

For

```
for(let counter = 5; counter < 11; counter++){
    code;
}
```

While

```
let counterTwo = 1;
while (counterTwo < 4) {
  console.log(counterTwo);
  counterTwo++;
}
```

Do While

```
let countString = '';
let i = 0;

do {
  countString = countString + i;
  i++;
} while (i < 5);
```

High-Order function

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

Functions as Data

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

```
const function1 = () => {
    code;
};

const function2 = function1;
function2(); // This function points to the same space in memory where function 1 is located
```

In JavaScript, functions are first class objects. This means that, like other objects you've encountered, JavaScript functions can have properties and methods.

Since functions are a type of object, they have properties such as .length and .name and methods such as .toString(). You can see more about the methods and properties of functions in the documentation.

Function as Parameter

Since functions can behave like any other type of data in JavaScript, it might not surprise you to learn that we can also pass functions (into other functions) as parameters. A higher-order function is a function that either accepts functions as parameters, returns a function, or both! We call the functions that get passed in as parameters and invoked callback functions because they get called during the execution of the higher-order function.

```
const timeFuncRuntime = funcParameter => {
   let t1 = Date.now();
   funcParameter();
   let t2 = Date.now();
   return t2 - t1;
}

const addOneToOne = () => 1 + 1;

timeFuncRuntime(addOneToOne);
```

Iterators

Iterators are built-in JavaScript array methods that help us iterate. Iterators are methods called on arrays to manipulate elements and return values.

Methods

forEach

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

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

map

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

const squareNumbers = numbers.map(number => {
  return number * number;
});
```

Or (that works for all of the three methods in this section)

```
Function p2(number){
   return number * number;
}
const squareNumbers = numbers.map(p2);
```

filter

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

findIndex

return the index of the first element that evaluates to true in the callback function.

```
const jumbledNums = [123, 25, 78, 5, 9];
const lessThanTen = jumbledNums.findIndex(num => {
  return num < 10;
});
```

reduce

returns a single value after iterating through the elements of an array, thereby reducing the array

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

const summedNums = numbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue
})
```

some

tests whether at least one element in the array passes the test implemented by the provided function.

```
const array = [1, 2, 3, 4, 5];
const even = (element) => element % 2 === 0;
```

Objects

Objects are passed by reference.

Creating Object Literals

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

Or

```
// An object literal with two key-value pairs
let spaceship = {
  'Fuel Type': 'diesel',
  color: 'silver'
};
```

Bracket Notation

```
spaceship['Fuel Type'] // returns 'diesel'
```

Property Assignment

```
spaceship['Fuel Type'] = 'hydrogen';
spaceship.color = 'gold';
delete spaceship.mission;  // Removes the mission property
```

Methods

```
const spaceship = {
  forward: function () {
    console.log('One step forward')
  }
};
```

Or

```
const spaceship = {
```

```
  forward () {
    console.log('One step forward')
  }
};
```

Pass By Reference

This means when we pass a variable assigned to an object into a function as an argument, the computer interprets the parameter name as pointing to the space in memory holding that object. As a result, functions which change object properties actually mutate the object permanently (even when the object is assigned to a const variable).

However, reassignment of the spaceship variable wouldn't work in the same way.

Looping Through Objects

for...in

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

The this Keyword

Reference to the property of an object from within the same object

Arrow Functions and this

If we use the code:

```
const goat = {
  dietType: 'herbivore',
  makeSound() {
    console.log('baaa');
  },
  diet: () => {
    console.log(this.dietType);
  }
};
```

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

In the code snippet above, the value of this is the global object, or an object that exists in the global scope, which doesn't have a dietType property and therefore returns undefined.

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

Privacy

JavaScript developers follow naming conventions that signal to other developers how to interact with a property.

Getters

To access a property you can use a getter method:
`get fullName() {}`

Setter

Same for setting a property:
`set fullName() {}`
`robot.fullName =  'Full Name';`

Factory Functions

It is a function that returns an object and can be reused to make multiple object instances.

```
const monsterFactory = (name, age, energySource, catchPhrase) => {
  return {
    name: name,
    age: age,
    energySource: energySource,
    scare() {
      console.log(catchPhrase);
    }
  }
};
```

And a shortened version:

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

```
};
```

Destructured Assignment

```
const { functionality } = robot;
```

Built-in Object Methods

`Object.keys(object)`. Return the properties of the object.
`Object.entries(object)`. Return key and value of the properties of the object.
`Object.assign(target, source)`. Creates a new object that has all the properties of source and the new properties from target.