Learn to Code

Course Material • Hamburg Coding School • 17.08. - 02.09.2020

Outline

Introduction

Basic Coding Logic and Syntax

- JavaScript is basically computed line by line,
  from top down, from left to right,
  what is written inside brackets is evaluated before what is outside them.
- Certain keywords are used to identify actions to be performed
  (e.g. the var keyword tells the browser to create a variable).
- When you name a variable (or array, function etc.) the first character must be a letter, an underscore (_) or a dollar sign ($). Subsequent characters may be letters, numbers, underscores or dollar signs. Also:
    - it can not be one of the keywords that are reserved for JavaScript (like *var*, *if*, *else*, *for*, *break*, etc. - about 40 words, you can look them up here: https://www.w3schools.com/js/js_reserved.asp)
    - it can not start with a number
    - JavaScript is case sensitive (the variables lastname and lastName are two different variables)
    - by convention and for readability JavaScript uses camelCase and PascalCase.
- Comments are ignored by the browser and will not be executed. They can be used to explain JavaScript code or to prevent execution, when testing alternative code.
  ```
  // double dashes to comment out single lines
  /* dashes and asterisks to comment out
  multiple lines */
  ```
- Semicolons separate JavaScript statements. Add a semicolon at the end of each executable statement.
- JavaScript statements can be grouped together in code blocks, inside curly brackets `{...}`.
  The purpose of code blocks is to define statements to be executed together.

Coding Principles

- Break the task down into small steps.
- First make it work, then make it better.
- Test if what you are doing is working (for example with `console.log()` etc.)
- Check the console to find errors.
- Make errors to learn from them.
- Your code should be right as well as easy to read by others.
- Naming your variables and functions for what they are meant will make your code more readable.
- If you can’t solve it, take a break.
- If you don’t make progress for one hour, ask someone.
- DRY: Don’t Repeat Yourself

JavaScript

Data Types

A variable can hold a value, which can be a text (string), a number or something else. This is referred to as the data type.

numbers	JavaScript has only one type of numbers. Many other programming languages distinguish between *integers* (whole numbers, e.g. 35) and *floats* (everything with a dot, e.g. 3.14, or even 1.0).
strings	Any series of characters like "John Doe". Strings are written with quotes. You can use single quotes ‘…’ or double quotes “…”.
booleans	Can only have one of two values: **true** or **false**
arrays	A structure that allows you to store a list of values in one single reference.
objects	Constructs that have properties (see section about Objects). Everything in JavaScript is an object, and can be stored in a variable.

When adding a number and a string, JavaScript will treat the number as a string.

Mathematical Operators

+	Addition (and also used for concatenating strings)
-	Subtraction
*	Multiplication
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement
**	Exponentiation (optional knowledge)

Assignment Operators

=	assigns a value
+=	example: x += y is the same as x = x + y
-=	example: x -= y is the same as x = x - y
*=	example: x += y is the same as x = x * y
/=	example: x += y is the same as x = x / y
%=	example: x += y is the same as x = x % y
**=	example: x += y is the same as x = x ** y (optional knowledge)

Comparison Operators

==	equal to
!=	not equal
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
===	equal value and equal type (optional knowledge)

!==	not equal value or not equal type (optional knowledge)

Logical Operators

&&	logical AND	(both sides need to be true)
||	logical OR	(one side needs to be true)
!	logical NOT	(turns true to false and vice versa)

Variables

JavaScript variables can be seen as containers for storing data.
With **var** you declare (create) it, with **=** you assign it a value. This can be done in one line:

```
var carBrand = “Mercedes”;
```

`carBrand` is the name of the variable, `“Mercedes”` is its value.

You can access the variable value by using the variable name.

```
console.log(carBrand);  // Will print out “Mercedes”
```

If you reassign the variable a different value later, the original value gets overwritten.

```
var favoriteDesert = “ice cream”;
var favoriteDesert = “chocolate cake”; // the variable name has
not changed, but its value is now “chocolate cake”.
```

JavaScript first calculates the value before assigning it to the variable.

```
var x = 5;
var x = x + 3;  // now the value of x is 8
```

Optional knowledge:

`let` is used in web development, an alternative to `var`, but stricter

`const` is used for constants, meaning the content can’t change

Arrays

An array is a special variable, which can hold more than one value at a time.

```
var arrayName = [ “a”, “b”, “c” ];
```

JavaScript arrays are written with square brackets.

Array items are separated by commas and their indexes are zero-based, which means the first item is [0], second is [1], and so on.

You access a value in an array by using the array name, and the index in brackets.

```
console.log(arrayName[1]);   // Will print out “b”
```

Adding Array Elements

The easiest way to add a new element to (the end of) an array is using the `push()` method:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Lemon");    // adds a new element (Lemon) to fruits
```

Removing Array Elements

The pop() method removes the last element from an array:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.pop(); // Removes the last element ("Mango") from fruits
```

For Loops

Loops are used to execute a block of code a number of times.

```
for(var i = 0; i < 10; i++) {
  // code block to be executed 10 times
}
```

A for loop always needs three statements:

```
for(statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

`statement 1` is executed (one time) before the execution of the code block.

`statement 2` defines the condition for executing the code block.

`statement 3` is executed (every time) after the code block has been executed.

Best used when you know how often you want to run some code.

While Loops

While Loops can execute a block of code as long as a specified condition (inside the brackets) is true.

```
var condition = true;
while (condition) {
  // code block to be executed
  condition = false;
}
```

Best used for when you don't know how often some code has to run until a certain condition is met.

if .. then Blocks

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

Everything in the blog will only be executed if the condition (can be a variable or a comparison) is true.

```
if (condition) {
  // code block A to be executed
} else {
  // code block B to be executed
}
```

‘else’ and ‘else if’

With **`else if`** you can check for another condition and with **`else`** you can define an alternative block that is executed if the condition is (or all previous conditions are) false:

```
if (conditionA) {
  // code block A to be executed
} else if (conditionB) {
  // code block B to be executed
} else {
  // code block C to be executed
}
```

You can add any number of `else if`'s, but there can only be one (single) `if` and one (single) `else`.
You don't have to define an `else`. But if you use it, it has to be at the end.

Optional knowledge:

The 'break' keyword

```
for (var i = 0; i < 10; i++) {
  if (condition) {
    break; // leave the loop
  }
  // code to be executed
}
```

With **`break`** you can stop the `for` loop. All following code in the `for` block will be ignored, and instead it will continue to execute the code below the `for` loop.

Functions

Functions are code blocks with a name.

```
function printSomething() {
  console.log(“some text”);
}
```

You can execute a function by *calling* it like this:

```
printSomething();   // will print “some text”
```

Parameters

You can pass parameters to a function:

```
function print(name) {
  console.log(name);
}
```

If you use parameters, you have to put them into the brackets when calling the function:

```
var myName = “Teresa”;
printName(myName);   // will print “Teresa”
```

Return values

Functions can return a value.
The return statement stops the execution of a function and returns a value from that function.

```
function calculateAverage(number1, number2) {
  var sum = number1 + number 2;
  var average = sum / 2;
  return average;
}
```

You can, for example, write the returned value into a variable:

```
var a = 27;
var b = 45;
var average = calculateAverage(a, b);
```

Functions can only return one value.

Scope

In JavaScript there are two types of scope:

- Local scope
- Global scope

JavaScript has 'function scope': Each function creates its own new scope.

Scope determines the accessibility (visibility) of these variables.

Variables declared inside a function are not accessible (visible) from outside the function. They become *local* to the function.

Local variables have function scope: They can only be accessed from within the function.

Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

Local variables are created when a function starts, and deleted when the function is completed.

A variable declared outside a function becomes *global*.

A global variable has global scope: All scripts and functions on a web page can access it.

Matrix

A Matrix is an array within an array.

```
var hotelRooms = [
     [  1,  2,  3,  4,  5 ],
     [ 11, 12, 14, 15, 16 ],
     [ 21, 22, 23, 24, 25 ]
];
```

You can access its values just like with an array, only now you need to specify two indexes: row and column.

```
var room = hotelRooms[1][2];
console.log(room);  // prints 14
```

Objects

Objects are are variables too, but (like arrays) can contain many values. Other than in arrays its values are not ordered but written as **name : value** pairs.

```
var person = {
    firstName : "John",
    lastName  : "Doe",
    age       : 50,
    eyeColor  : "blue",
    student   : true
};
console.log(person.firstName + " is " + person.age
    + " years old."); // will print “John is 50 years old.”
```

JavaScript objects are written with curly braces: **{}**

The **`name:value`** pairs in JavaScript objects are called **properties**. They are separated by commas. Object properties can be single values like strings or numbers or other arrays, objects or functions.

You can access object properties in two ways:

- `objectName.propertyName` or
- `objectName["propertyName"]`

This is also called the JavaScript Object Notation - **JSON**.

Optional knowledge:

Methods

Objects can also have *methods*: a method is a function stored as a property.

```
var person = {
    firstName : "John",
    lastName  : "Doe",
    sayHello  : function() {
                  console.log("Hello my name is John Doe");
                 }
};
```

You call it like this:

```
person.sayHello();    // prints "Hello my name is John Doe"
```

In JavaScript almost everything is an **object**. For example:

`Math.PI`	PI is a **property** of MATH with the value of ≈ 3.14
`Math.random()`	random() is a function stored as a property of Math and returns a random number

Method Chaining

For objects that have methods, you can build a chain with the method calls, as an alternative of storing each return value in a variable.

Let's have a look at an example. We have two objects with methods:

```
var johnDoe = {
    firstName  : "John",
    lastName   : "Doe",
    doExercise : function() {
                   console.log(“Doing a coding exercise...”);
                 }
};
var mentor = {
    firstName : "Max",
    lastName  : "Mustermann",
    mentee    : johnDoe,
    getMentee : function() {
                  return this.mentee;
                }
};
```

On these we could call their methods one after the other:

```
var mentee = mentor.getMentee();
mentee.doExercise();  // Prints out “Doing a coding exercise...”
```

This would be the same as doing:

```
mentor.getMentee().doExercise();
```

This is called method chaining.

Method chaining also works with line breaks. This is useful if the chain gets too long.

```
mentor.getMentee()
      .doExercise();  // Would work the same way
```

JavaScript in Web Development (Optional Knowledge)

DOM

Now you have an idea what an object is, try to see the whole webpage as an object.

The <body> of a webpage for example is a property of the webpage and can therefore be addressed like this:

```
document.body  // the <body> tag
```

You also already used one of its functions / methods:

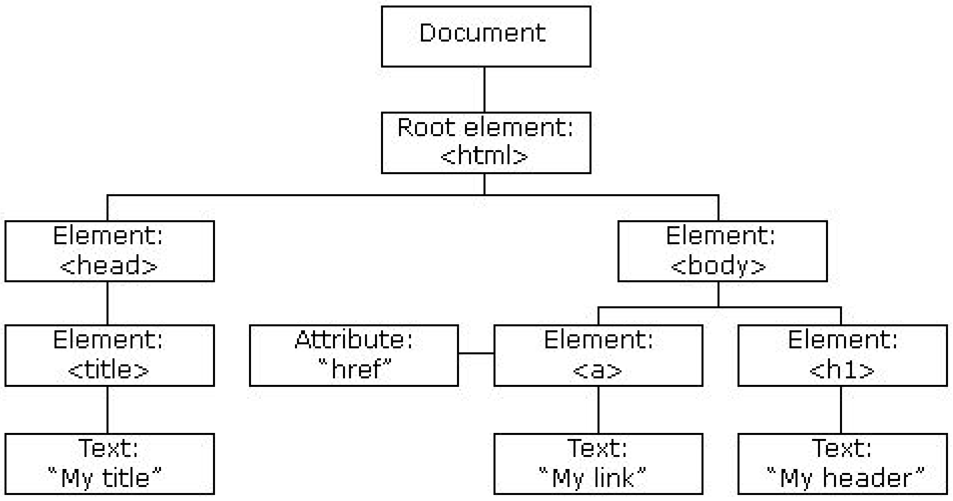
`write()` is a method of the object `document` and `document.write()` writes an HTML expression to the webpage.

With your current vocabulary you probably get what these are doing:

```
document.images.length /* addresses the property with the number
of images on a page as its value */

document.body.style.backgroundColor = "yellow"; /* this will set
the property backgroundColor to its new value “yellow” */
```

What JavaScript does is it builds a tree (nested objects) of all tags in an HTML document in the background, so you can access it with JavaScript when needed.



(Source: https://www.w3schools.com/whatis/whatis_htmldom.asp)

This is called the DOM: the Document Object Model.

You can access DOM elements with JavaScript like this:

`document.getElementById("demo");`

The getElementById() method returns the element that has an ID attribute with the specified value:

```
<div id=”demo”>...</div>
```

This method is one of the most common methods in the HTML DOM, and is used almost every time you want to manipulate, or get info from, an element on your document.

It returns `null` if no elements with the specified ID exists.

An ID should be unique within a page. However, if more than one element with the specified ID exists, the `getElementById()` method returns the first element in the source code.

Manipulating Elements

To manipulate the element you can access it like any other property in an object:

```
// first get the element with id="demo"
var x = document.getElementById("demo");
// Change the color of the element
x.style.color = "red";
// Change the text inside an element
x.innerHTML = “New content”;
```

Input and Output

Output:

`console.log(“Text”);`	Prints “Text” on the console.
`document.write(“Text”);`	Writes “Text” into the HTML file.
`document.getElementById("demo").innerHTML = "Text"`	Writes “Text” into element with id=”demo”
`alert("Text");`	Displays a popup that says “Text”.

Input:

```
var person = prompt("Please enter your name", "Harry Potter");
```

Shows a popup that says “Please enter your name” and has a text field that saves the value into the variable person. If no text input is given, it uses the default “Harry Potter”.

In web development, input is usually done with forms, which is not part of this course.

JavaScript and HTML Events

HTML events are "things" that happen to HTML elements.

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Often, when events happen, you may want to do something.

JavaScript lets you execute code when events are detected.

The **addEventListener()** method is a built-in function in JavaScript which takes the event to listen for, and a second argument to be called whenever the described event gets fired. Any number of event handlers can be added to a single element without overwriting existing event handlers.

```
element.addEventListener(event, function);
```

event : event can be any valid JavaScript event like ‘click’. Have a look at all HTML DOM events you can target: https://www.w3schools.com/jsref/dom_obj_event.asp

function : specifies the function to run when the event occurs. This function does not need a name, it is automatically invoked (called) when the event becomes true.

Example:

```
<body>
  <button id="myBtn">Click here</button>
  <h1 id="text">Some text</h1>
  <script>
    function putNewText() {
        document.getElementById("text").innerHTML = "Now there is
some new text in the h1";
    }
    document.getElementById("myBtn")
            .addEventListener("click", putNewText());
  </script>
</body>
```

Advanced Topics (Optional Knowledge)

Anonymous Functions

If you want to pass a function as a parameter (like putNewText() in the example above), there is also a shorter way to do that.

You could also write the example above like this:

```
document.getElementById("myBtn")
        .addEventListener("click", function() {
            document.getElementById("text").innerHTML = "Now there
is some new text in the h1";
        });
```

This way we didn't define a function first and then use it, but we are defining it where we pass it as a parameter. This is called an anonymous function.

It is called "anonymous", because this way we cannot call this method at any other place, because it does not have a name.

You could also say it doesn't *need* a name because it is not meant to be called anywhere outside its context.

Classes

Classes can be seen as useful templates or blueprints to create new objects with predefined properties.

You use the keyword `class` to create a class and always add a constructor method. (By convention classes are named using PascalCase.)

```
class Person {
  constructor(name, age, city) {
    this.name = name;
    this.age = age;
    this.city = city;
  }
}
```

Now, in order to create a new object with the properties of name, age and city we can use the `new` keyword:

```
var p1 = new Person(“Martha”, 40, “London”);
var p2 = new Person(“Joe”, 18, “NYC”);
```

Each will create a new object. The first one (p1) will look like this:

```
var p1 = {
   name : “Martha”,
   age : 40,
   city : “London”
}
```

A class is really a type of function, but instead of using the keyword `function` to define it, we use the keyword `class` and the properties are assigned inside a `constructor()` method.

The constructor method is called each time the class object is initialized with the keyword new.

The ‘this’ Keyword

In JavaScript, the thing called `this` is the object that "owns" the code.

The value of `this`, when used in an object, is the object itself.

```
var person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

In a function definition, `this` refers to the ‘owner’ of the function.
In the example above, `this` is the ‘person’ object that "owns" the `fullName` function.
In other words, `this.firstName` means the firstName property of this object.

The JavaScript `this` keyword refers to the object it belongs to.
It has different values depending on where it is used:

- In a method, this refers to the owner object.
- Alone, this refers to the global object.
- In a function, this refers to the global object.
- In an HTML event, this refers to the element that received the event.

Object-Oriented Programming

In JavaScript we usually do *imperative programming*. This means, we define functions, and then call them, and the code gets executed statement by statement from top to bottom.

Object-oriented programming (OOP) is a different programming paradigm. Here, we create classes, object instances of these classes, and call their methods.

In JavaScript, you can use both and mix them.

Examples for languages that are completely object-oriented are Java, C++ and TypeScript (a “dialect” of JavaScript).

As soon as you use classes, you use object-oriented programming.

The concept of object-oriented programming is quite complex and not easy to understand. Deep knowledge of OOP is not part of this course. Don’t worry if you don’t understand the concept.

You can become an excellent JavaScript developer and still avoid object-oriented programming. ;-)

Glossary

`console.log();`	writes to the console
`document.write();`	writes an HTML expression to the webpage
`alert();`	displays an alert box with a specified message
`prompt();`	displays a dialog box that prompts the visitor for input
`var`	Declares a variable
`let`	Declares a variable (with *Block Scope* - optional knowledge)
`const`	Declares a variable (with *Block Scope* - optional knowledge)
`for`	Declares a *for loop*
`while`	Declares a *while loop*
`if`	Specifies a block of code to be executed if a condition is true
`else if`	Specifies a block of code to be executed if above conditions are false and a new condition is true
`else`	Specifies a block of code to be executed if all previous conditions are false.
`break`	ends the execution of the loop
`length`	Returns the number of elements in an array or the number of characters in a string

`function`	Declares a function
`return`	The return statement stops the execution of a function and returns a value from that function.
`parseInt()`	takes a string and returns an integer (whole number)
`Math.random()`	returns a random number between 0 (inclusive) and 1 (exclusive)
`Math.floor()`	rounds a number downwards to the nearest integer (whole number), and returns the result.
`Math.ceil()`	Rounds a number up to the next higher integer (whole number), and returns the result.
`Math.PI`	returns the number Pi (π)
`this`	refers to the object it belongs to
`DOM - Document Object Model`	All tags of an HTML document as one big, nested JavaScript object.
`push()`	The push() method adds a new element to (the end of) an array.
`pop()`	The pop() method removes the last element from an array
`null`	In JavaScript **null** is meant to be "nothing". Unfortunately, in JavaScript, the data type of null is an object. (You can consider it a bug in JavaScript that the type of null is an object.) Therefore **undefined** and **null** are equal in value (empty) but different in type (undefined <> object).

`undefined`	A variable without a value, has the value undefined. The type is also undefined.
`NaN`	“Not a Number”
`class`	keyword to create a class
`constructor()`	method (function) that is called each time the class object is initialized
`new`	creates a new object by executing the constructor function of the addressed class
`substr()`	extracts parts of a string, beginning at the character at the specified position, and returns the specified number of characters.
`document`	name of the object that represents your webpage
`getElementById()`	addresses an HTML element by its ID
`innerHTML`	addresses the inner HTML of the specified element
`addEventListener()`	attaches an event handler to the specified element

Useful Links

https://www.w3schools.com/js

https://developer.mozilla.org/en-US/docs/Web/JavaScript

https://eloquentjavascript.net/

You Don't Know JS book series: https://github.com/getify/You-Dont-Know-JS/tree/1st-ed

An introduction for new programmers: http://jsforcats.com/

DOM: https://javascript.info/dom-navigation

JSON: http://json.org/

A list of funny and tricky JavaScript examples: https://github.com/denysdovhan/wtfjs