Javascript Notes

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

5 12

Getting started

1.1 Javascript directly in HTML

Here is an example of how to write a very simple web page that will give a pop-up box saying Hi there!:

```
<html>
    <script type="text/javascript"> alert("Hi there!");
    </script>
</html>
```

1.2 External file

First, we are going to create a separate JavaScript file. These files have the postfix .js. I'm going to call it ch1_alert.js. This will be the content of our file:

```
alert("Saying hi from a different file!");
```

Then, in an html file

```
<html>
    <script type="text/javascript" src="ch1_alert.js"></script>
</html>
```

1.3 Running javascript like a script with node

You can run JavaScript code in the terminal using Node.js.

```
console.log("Hello world")
```

```
node script.js
```

1.4 Semicolons

After every statement, you should insert a semicolon. JavaScript is very forgiving and will understand many situations in which you have forgotten one

1.5 Comments

Js has two types of comments, single line // and multi line /* */

1.6 Prompts

Another thing we would like to show you here is also a command prompt. It works very much like an alert, but instead, it takes input from the user

```
prompt("Enter something")
```

1.7 Random

We can generate random reals between 0 and 1 with

```
Math.random()
```

Multiply by 100 to get reals between 0 and 100

```
Math.random() * 100
```

Use the floor function to truncate to integers

```
Math.floor(Math.random() * 100)
```

Essentials

2.1 Variables

2.1.1 let, var, and const

A variable definition consists of three parts: a variable-defining keyword (let, var, or const), a name, and a value. Let's start with the difference between let, var, or const. Here you can see some examples of variables using the different keywords:

```
let nr1 = 12;
var nr2 = 8;
const PI = 3.14159;
```

let and var are both used for variables that might have a new value assigned to them somewhere in the program. The difference between let and var is complex. It is related to scope.

var has global scope and let has block scope. var's global scope means that you can use the variables defined with var in the entire script. On the other hand, let's block scope means you can only use variables defined with let in the specific block of code in which they were defined

On the other hand, const is used for variables that only get a value assigned once— for example, the value of pi, which will not change. If you try reassigning a value declared with const, you will get an error:

2.2 Types

2.2.1 Strings

- Double quotes
- Single quotes
- **Backticks:** special template strings in which you can use variables directly

  Aka string interpolation

```
name = "Nate"
console.log(`Hello ${name}`)
```

2.2.2 Number

The number data type is used to represent, well, numbers. In many languages, there is a very clear difference between different types of numbers. The developers of JavaScript decided to go for one data type for all these numbers: number. To be more precise, they decided to go for a 64-bit floating-point number. This means that it can store rather large numbers and both signed and unsigned numbers, numbers with decimals, and more.

2.2.3 BigInt

A BigInt data type can be recognized by the postfix n:

```
let big = 123n
```

Cannot mix BigInt and other types, use explicit conversions

2.2.4 Boolean

true or false

2.2.5 Undefined

It has a special data type for a variable that has not been assigned a value. And this data type is undefined:

```
let unassigned;
console.log(unassigned);
```

We can also purposefully assign an undefined value. It is important you know that it is possible, but it is even more important that you know that manually assigning undefined is a bad practice:

```
let x = undefined
```

2.2.6 null

```
let empty = null;
```

2.2.7 Getting type

We use typeof

2.2.8 Converting data types

We have String(), Number(), and Boolean()

Operators

- +,-,/,*, **, %
- **Postfix, Prefix**
- ==: Compares value and does type conversions
- ===: Compares value and type, does not do any conversions
- !=
- !==
- <, >, <=, >=
- &&: And
- ||: Or
- !: Not
- []: Index operator

Arrays and their properties

Arrays are lists of values. These values can be of all data types and one array can even contain different data types

4.1 Creating arrays

```
let arr = [1,2,3]
let arr = new Array(3)  // Array of three undefined values
```

```
let arr = [1,2,3,4]
console.log(arr[-1]) // Undefined

arr[-1] = "What?"
console.log(arr[-1]) // What?
```

4.1.1 Methods and member variables

Member variables:

- **.length**

Methods

- **push(...items)**: Adds one or more elements to the end of the array.
- **pop()**: Removes the last element from the array and returns that element.
- **shift()**: Removes the first element from the array and returns that element.
- **unshift(...items)**: Adds one or more elements to the beginning of the array.
- **splice(start, deleteCount, ...items)**: Adds or removes items from the array at the specified index.
- **sort(compareFunction?)**: Sorts the elements of the array in place and returns the sorted array.
- **reverse()**: Reverses the order of the array elements in place.
- **fill(value, start?, end?)**: Fills elements in the array with the specified value.
- **copyWithin(target, start, end?)**: Copies a sequence of elements within the array to another position within the same array.
- **concat(...arrays)**: Returns a new array that is the result of merging two or more arrays.
- **slice(begin?, end?)**: Returns a shallow copy of a portion of an array into a new array.

- **join(separator?)**: Joins all elements of an array into a string using the specified separator.
- **toString()**: Returns a string representing the specified array and its elements.
- **toLocaleString()**: Returns a localized string representing the array and its elements.
- **forEach(callback(currentValue, index, array))**: Executes a provided function once for each array element.
- **map(callback(currentValue, index, array))**: Creates a new array with the results of calling a provided function on every element.
- **filter(callback(currentValue, index, array))**: Creates a new array with all elements that pass the test implemented by the provided function.
- **reduce(callback(accumulator, currentValue, index, array), initialValue?)**: Applies a function against an accumulator and each element to reduce the array to a single value.
- **reduceRight(callback(accumulator, currentValue, index, array), initialValue?)**: Same as reduce(), but processes the array from right-to-left.
- **every(callback(currentValue, index, array))**: Tests whether all elements pass the provided test function.
- **some(callback(currentValue, index, array))**: Tests whether at least one element passes the provided test function.
- **find(callback(currentValue, index, array))**: Returns the first element that satisfies the provided testing function.
- **findIndex(callback(currentValue, index, array))**: Returns the index of the first element that satisfies the provided testing function.
- **indexOf(searchElement, fromIndex?)**: Returns the first index at which a given element can be found.
- **lastIndexOf(searchElement, fromIndex?)**: Returns the last index at which a given element can be found.
- **includes(searchElement, fromIndex?)**: Determines whether an array includes a certain element, returning true or false.
- **flat(depth?)**: Creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.
- **flatMap(callback(currentValue, index, array))**: First maps each element using a mapping function, then flattens the result into a new array.
- **at(index)**: Returns the item at the given index, supporting negative indices to count back from the end (introduced in ES2022).

Iterators

- **entries()**: Returns a new Array Iterator object that contains key/value pairs.
- **keys()**: Returns a new Array Iterator object that contains the keys for each index.
- **values()**: Returns a new Array Iterator object that contains the values for each index.

Array Static Methods: These methods are called on the Array constructor itself rather than on an instance:

- **Array.from(iterable, mapFunction?, thisArg?)**: Creates a new, shallow-copied Array instance from an array-like or iterable object.
- **Array.isArray(value)**: Determines whether the passed value is an Array.
- **Array.of(...elements)**: Creates a new Array instance with a variable number of arguments, regardless of number or type of the arguments.

String properties