Javascript Notes

Nathan Warner

Computer Science
Northern Illinois University
United States

Contents

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

Objects

Objects in javascript can be created with curly braces, for example

```javascript
let dog = { dogName: "JavaScript",
    weight: 2.4,
    color: "brown",
    breed: "chihuahua",
    age: 3,
    burglarBiter: true
};
```

There are two main ways to index objects,

```javascript
console.log(dog["weight"]);
console.log(dog.weight);
```

Logic statements

7.1 If, else if, else

Uses C syntax

```
if (...) {

} else if (...) {

} else {

}
```

7.2 Ternary statements (conditional operator)

Also uses C syntax

```
flag = true

console.log(flag ? "True!" : "False!")
```

7.3 Switch

Also uses *C* syntax

```
switch (expression) {
    case value1:
        ...
        break;
    ...
    default:
        ...
        break;
}
```

Loops

8.1 While and do while loops

Uses *C* syntax

```
let tt = 5, t =0;
while (tt--) {
    console.log(t++);
}
```

And

```
let flag = true
do {
    console.log("Hello world");
    flag = false;
} while (flag);
```

8.2 For loops

Uses *C* syntax

```
for (let i=0; i<5; ++i) {
    console.log(i);
}
```

8.3 For of loop

There is another loop we can use to iterate over the elements of an array: the for of loop. It cannot be used to change the value associated with the index as we can do with the regular loop, but for processing values it is a very nice and readable loop.

```
let A = [1,2,3]
for (let e of A) {
    console.log(e)
}
```

```
let A = [1,2,3]
for (let e of A) {
    ++e;
}

for (let e of A) {
    console.log(e);
}

/*
1
2
3
*/
```

8.4 For in loop

The `for in` loop gives us the indices of an array, or in the case of an object, the keys

```
let A = {
    name: "Nate",
    lang: "js",
    x: 10
};

for (let e in A) {
    console.log(e, ":", A[e]);
}
```

```
let A = [1,2,3]

for (let e in A) {
    console.log(e)
}

/*
0
1
2
*
```

8.5 Looping over objects with C++ structured binding syntax

```
let A = {
    name: "Nate",
    lang: "js",
    x: 10
};


for (let [k,v] of Object.entries(A)) {
    console.log(k, ":", v);
}
```

8.6 Labels with breaking

In js we can give labels to an area in the program, then specify the name of those labels as the argument of a break statement.

```
outer:
for (let i=0; i<5; ++i) {
    inner:
    console.log("i:", i);
    for (let j=0; j<5; ++j) {
        console.log("j:", j);
        if (j == 2) break outer;
    }
}

/*
i: 0
j: 0
j: 1
j: 2
*/
```

Functions

9.1 Simple functions

```
function f(x,y) {
    console.log(x+y);
}
f(1,2); // 3
```

We can define functions inside of functions

```
function f(x,y) {
    function g() {
        console.log(x+y);
    }
    g();
}
f(1,2); // 3
```

9.2 Assigning functions to variables

We can also initialize a variable with a function definition, for example

```
let f = function() {
    console.log("Hello world");
}
f()
```

9.3 Functions missing arguments

Javascript does not complain, it simply gives NaN

```
function f(x,y) {
    console.log(x+y);
}
f(1) // NaN
```

9.4 Arrow functions (Lambdas)

For one line lambdas

```
let f = x => x + 5; // One parameter
let g = (x,y) => x+y; // Multiple parameters
let h = () => console.log("Hello World"); // No parameters

(() => console.log("Hello world"))();
```

For multi line lambdas

```
(x => {
    console.log("Some function");
    console.log("x is", x);
})(5);

let f = x => {
    return x + 10;
};
let x = f(5);
console.log(x); // 15
```

Notice the set of parenthesis around the top function, this is required for immediately invoked lambdas.

9.5 The spread operator

```
function f(x,y,z) {
    return x + y + z;
}

let A = [1,2,3];
console.log(f(...A)) // 6;
```

or

```
function f(x,y,a,b) {
    return (x*y) + (a*b);
}

let A = [1,2];
let B = [3,4]
console.log(f(...A, ...B)) // 14;
```

9.6 Rest parameter

```
function f(...args) {
    for (let arg of args) {
        console.log(arg);
    }
    console.log(typeof(args)); // Object
}
f(1,2,3,4);
```

```
function f(arg1, ...other) {
    console.log("Arg1 is", arg1);
    for (let arg of other) {
        console.log(arg);
    }
}
f(1,2,3,4);
```

9.7 Hoisting

Hoisting is a process during the compilation phase where JavaScript moves the declarations of functions (and variables, if using var) to the top of their scope. For functions, both the name and the function body are hoisted.

```
f(); // Works because f is hoisted

function f() {
    console.log("Hello world");
}
```

In the above code, the entire function f is hoisted. That means the JavaScript engine is aware of the function f from the very start, allowing it to be called even before the line where f is defined.

9.8 setTimeout and setInterval

`setTimeout(callback, delay, param1?, param2?, ... paramN)` takes a function and a time in milliseconds, it calls the callback, but only after the specified time has elapsed. The optional arguments at the end are passed to the function after the delay.

`setInterval(callback, delay, param1, param2?, ..., paramN)` works similarly to setTimeout, repeatedly calls a function or executes a code snippet, with a fixed time delay between each call.

9.9 Restructuring function return values

For a function that returns an array,

```
function f() {
    return [1,2,3];
}

let [a,b,c] = f();
console.log(a,b,c);
```

We can also retrieve only some of the values

```
function f() {
    return [1,2,3];
}

let [a,b] = f();
console.log(a,b);
```

For objects,

```
function f() {
    return {name: "Nate", lang: "Javascript"};
}

let {name, lang} = f();
console.log(name, lang);
/*
Nate Javascript
*/
```

and we can also alias the variables

```
function f() {
    return {name: "Nate", lang: "Javascript"};
}

let {name: n, lang: l} = f();
console.log(n, l);
/*
Nate Javascript
*/
```

We can even get only some of the values

```javascript
function f() {
    return {name: "Nate", lang: "Javascript"};
}

let {name} = f();
console.log(name);
/*
Nate
*/
```

Classes

We can create a class with the syntax

```
class className {
    ...
}
```

We can create constructors with

```
class name {
    constructor(a1,a2,...an) {
        ...
    }
}
```

and create instances of our class with the *new* keyword

```
let obj = new className(a1,a2,...,an)
```

To give a class member variables, we can define them in the constructor, and with the *this* keyword

```
class C {
    constructor() {
        this.x = 15;
    }
};
let c = new C();
console.log(c.x);
```

To create methods, we use the normal function syntax but drop the *function* keyword. We can also define member variables inside methods.

```
class foo {
    constructor() {
        this.x = 15;
    }

    g() {
        this.y = 20;
    }
};

let f = new foo();
console.log(f.x, f.y); // 15 Undefined
f.g();
console.log(f.x, f.y); // 15 20
```

We can only declare member variables (fields) outside of a constructor or method, we cannot define them.

```
class foo {
    x;
    constructor() {
        this.x = 15;
    }
};
let f = new foo();
console.log(f.x);
```

We cannot use *let, var,* or *const* in declaring or defining class fields.

10.1 Private

We can create private fields with the # syntax

```
class foo {
    #x;
    constructor() {
        this.#x = 15;
    }
};
let f = new foo();
console.log(f.x); // Undefined
```

We can also use this syntax to make methods private

```
class foo {
    #print() {
        console.log("Hello world");
    }
};
let f = new foo();
f.print(); // Error
```

10.2 Getters and setters

Getters and setters are special properties that we can use to get data from a class and to set data fields on the class. Getters and setters are computed properties. So, they are more like properties than they are like functions. We call them accessors. They do look a bit like functions, because they have () behind them, but they are not! These accessors start with the get and set keywords.

```
class foo {
    #name;

    set name(name) {
        this.#name = name;
    }

    get name() {
        return this.#name;
    }
};
let f = new foo();
f.x = "Nate";
console.log(f.x); // Nate
```

10.3 Inheritence

We implement inheritance with the *extends* keyword

```
class A {
    x;
    constructor() {
        this.x = 20;
    }

    print() {
        console.log("Hello world");
    }
};

class B extends A {
};

let b = new B();
console.log(b.x); // 20
b.print(); // Hello world
```

Private fields and methods in A are not visible in B

```
class A {
    #x;
    constructor() {
        this.#x = 20;
    }

    print() {
        console.log("Hello world");
    }
};

class B extends A {
};

let b = new B();
console.log(b.x); // Undefined
b.print(); // Hello world
```

10.4 Prototypes

A prototype is the mechanism in JavaScript that makes it possible to have objects. When nothing is specified when creating a class, the objects inherit from the Object.prototype prototype. This is a rather complex built-in JavaScript class that we can use. We don't need to look at how this is implemented in JavaScript, as we can consider it the base object that is always on top of the inheritance tree and therefore always present in our objects

There is a *prototype* property available on all classes, and it is always named "prototype."

Here is how to add a function to this class using prototype

```
class Person {
    constructor(firstname, lastname) {
        this.firstname = firstname;
        this.lastname = lastname;
    }
    greet() {
        console.log("Hi there!");
    }
}

Person.prototype.introduce = function () {
    console.log("Hi, I'm", this.firstname);
};

let p1 = new Person("Bob", "Smith");
p1.introduce(); // Hi, I'm Bob
```

prototype is a property holding all the properties and methods of an object. So, adding a function to prototype is adding a function to the class. You can use prototype to add properties or methods to an object, like we did in the above example in our code with the introduce function. You can also do this for properties

```
Person.prototype.favoriteColor = "green";
```

So the methods and properties defined via prototype are really as if they were defined in the class.

This is something you should not be using when you have control over the class code and you want to change it permanently. In that case, just change the class. However, you can expand existing objects like this and even expand existing objects conditionally. It is also important to know that the JavaScript built-in objects have prototypes and inherit from Object.prototype. However, be sure not to modify this prototype since it will affect how our JavaScript works.

The Document Object Model

11.1 The BOM

The BOM, sometimes also called the window browser object, is the amazing ”magic” element that makes it possible for your JavaScript code to communicate with the browser.

The window object contains all the properties required to represent the window of the browser, so for example, the size of the window and the history of previously visited web pages. The window object has global variables and functions, and these can all be seen when we explore the window object. The exact implementation of the BOM depends on the browser and the version of the browser. This is important to keep in mind while working your way through these sections.

Some of the most important objects of the BOM we will look into in this chapter are:

- History
- Navigator
- Location

You can type the following command in the browser console and press Enter to get information about the window object:

```
console.dir(window);
```

The console.dir() method shows a list of all the properties of the specified object. You can click on the little triangles to open the objects and inspect them even more.

The BOM contains many other objects. We can access these like we saw when we dealt with objects, so for example, we can get the length of the history (in my browser) accessing the history object of the window and then the length of the history object, like this

```
window.history.length;
```

11.1.1 BOM Objects

We have

- **window**: The window object is the top-level object in a browser and serves as the global context for JavaScript running in the browser. All global variables and functions become properties of the window object.
- **history**: The history object allows you to navigate back and forth through the user’s session history.
- **location**: The location object provides information about the current URL and allows you to redirect the browser to a new URL or reload the current page.

- **navigator**: The navigator object exposes properties that provide information about the browser and the underlying operating system.
- **screen**: The navigator object exposes properties that provide information about the browser and the underlying operating system.The screen object contains information about the physical screen of the device. This includes properties like screen.width, screen.height, and screen.availHeight.
- **frames**: Although not an independent object per se, window.frames is a collection that refers to the child windows (or frames) if your document is split into different frames or iframes.

11.1.2 Common window properties and methods

Properties:

- **window.document**: Provides access to the DOM (Document Object Model) of the current page.
- **window.location**: References the location object, which provides details about the current URL (see Location section below).
- **window.history**: Gives access to the history object to navigate user history (see History section below).
- **window.navigator**: Provides details about the browser and the operating system (see Navigator section below).
- **window.innerWidth & window.innerHeight**: Represents the viewport's width and height. Useful for responsive designs and adapting layouts dynamically.
- **window.outerWidth & window.outerHeight**: Offers the total width and height of the browser window including interface elements like toolbars and scrollbars.
- **window.localStorage & window.sessionStorage**: Allow you to store data on the client side persistently or for the session duration, respectively.
- **window.console**: Provides access to the browser's debugging console for logging purposes.
- **window.frames**: Holds a collection of all the frames (or iframes) embedded in the window (further discussed under Frames).

Common Methods:

- **window.alert(message)**: Displays an alert dialog with the specified message.
- **window.confirm(message)**: Opens a modal dialog with OK and Cancel buttons, returning a Boolean based on the user's choice.
- **window.prompt(message, defaultText)**: Shows a dialog prompting the user for input.
- **window.setTimeout(callback, delay)**: Executes a function once after a specified delay (in milliseconds).
- **window.setInterval(callback, interval)**: Repeatedly executes a function with a fixed time delay between each call.

- **window.clearTimeout(timeoutID) & window.clearInterval(intervalID)** Cancel pending timeouts or intervals.
- **window.open(url, name, specs)**: Opens a new browser window or tab.
- **window.close()**: Closes the current window (typically works only if the window was opened by a script).
- **window.addEventListener(event, handler)**: Attaches an event handler to the window (and similarly, removeEventListener to detach).

11.1.3 History Objecti properties and methods

Properties:

- **history.length**: Returns the number of URL entries in the session history.

Methods:

- **history.back()**: Equivalent to clicking the browser's Back button. It moves backward by one in the history stack.
- **history.forward()**: Moves forward by one, similar to clicking the Forward button.
- **history.go(delta)**: Loads a specific page relative to the current page. For example, history.go(-1) is like calling history.back().
- **history.pushState(state, title, url)**: Adds a new state to the browser's session history without reloading the page. This is essential in Single Page Applications (SPAs) for creating navigable states.
- **history.replaceState(state, title, url)**: Modifies the current history entry instead of creating a new one, which is useful for updating the URL or state without affecting the history stack.

11.1.4 Location Object properties and methods

Properties:

- **location.href**: A string containing the full URL of the current page.
- **location.protocol**: The protocol scheme of the URL (e.g., ”http:” or ”https:”).
- **location.host**: Combines the hostname and port (if specified).
- **location.hostname**: The domain name of the web host.
- **location.port**: The port number used by the URL.
- **location.pathname**: The path (directory and file name) part of the URL.
- **location.search**: Contains the query string part of the URL (the portion following the ?).
- **location.hash**: Holds the anchor or fragment identifier of the URL (the portion following the).

Methods:

- **location.assign(url)**: Loads the resource at the specified URL, similar to clicking a link.
- **location.replace(url)**: Loads a new document, replacing the current document in the history. This means the current page won't be available via the Back button.
- **location.reload(forceReload)**: Reloads the current page. Using a truthy value for forceReload forces the page to reload from the server rather than the cache.

11.1.5 Navigator Object properties and methods

Properties:

- **navigator.appName**: Provides the name of the browser (though its use is generally discouraged due to its lack of reliability).
- **navigator.appVersion**: A version string for the browser.
- **navigator.userAgent**: A string that contains details about the browser, its version, and the operating system. This is often used for browser detection.
- **navigator.platform**: Indicates the operating system platform (e.g., ”Win32”, ”MacIntel”).
- **navigator.language**: The default language of the browser.
- **navigator.languages**: An array that lists the user's preferred languages.
- **navigator.onLine**: A Boolean indicating whether the browser is online.
- **navigator.cookieEnabled**: Indicates whether cookies are enabled in the browser.

Methods:

- **navigator.javaEnabled()**: Returns a Boolean indicating if the Java plugin is enabled in the browser.
- **navigator.sendBeacon(url, data)**: Used to asynchronously send data to a server, often employed for sending analytics data without blocking unloading of the document.

11.1.6 Screen Object

Properties

- **screen.width**: The total width of the visitor's screen in pixels.
- **screen.height**: The total height of the visitor's screen in pixels.
- **screen.availWidth**: The width of the screen available for the window (excluding interface elements like the taskbar).
- **screen.availHeight**: The height of the screen available for the window.
- **screen.colorDepth**: The number of bits used to display one color (usually 24 or 32).
- **screen.pixelDepth**: Similar to colorDepth, representing the actual color resolution.