Welcome to

JavaScript Essentials

1-3 March 2021

Some questions to you

before we dive in!

Number of programs written?

(all by yourself)

Favourite programming language?

Parts of JavaScript you know?

(you can give more than one answer)

Parts of JavaScript you wish you knew better?

(max 3 answers)

What is JavaScript?

JavaScript is

a dynamically typed, interpreted, scripting language

JavaScript lets you

script event driven user interactions inside the browser

script event driven server-side I/O with Node.js

JavaScript has

Java-like syntax, garbage collection,
prototypal inheritance, first-class functions

History of JavaScript

1995

Brendan Eich invents JavaScript at Netscape
In just 10 days!

1996

Netscape 2.0 embeds the SpiderMonkey engine

JScript powers Internet Explorer 3

The First Browser War has begun

1997

First version of ECMAScript (ECMA-262 standard)

1998

ECMAScript 2 is also standardized as ISO/IEC 16262

1999

ES3 is standardized based on Netscape 4 features

2000

Netscape 6 succumbs to Internet Explorer 5

AJAX takes off thanks to IE5's XMLHTTP ActiveX control

The First Browser War is over

2001

2002

2003

2004

Work on ES4 standardization begins

Firefox is born from open-sourced Netscape code

Mozilla and Opera found the WHATWG

The Second Browser War begins

2005

2006

2007

ES4 standardization fails because of internal dissent

Standardization of ES3.1 begins in parallel

2008

TC39 is founded to work on ES3.1

ES4 features are reworked into ECMAScript Harmony

ES4 is definitely abandoned

Google releases the first version of Chrome

2009

ECMAScript 3.1 is standardized as ECMAScript 5

2010

2011

2012

2013

2014

2015

ES6 Harmony is standardized as ECMAScript 2015

2016

ECMAScript 2016 is standardized

2017

ECMAScript 2017 is standardized

Google Chrome owns over 60% browser market share

The Second Browser War is over

2018

ECMAScript 2018 is standardized

2019

ECMAScript 2019 is standardized

2020

ECMAScript 2020 is standardized

You get it...

What does JavaScript look like?

Question:

What does JavaScript have in common with Java?

JavaScript is vaguely Java-like

JavaScript

```
for (var i = 0; i < 10; i++) {
  console.log("number " + i);
}
```

Java

```
for (int i = 0; i < 10; i++) {
  System.out.println("number " + i);
}
```

It's got names

```
for (var i = 0; i < 10; i++) {
  console.log("number " + i);
}
```

Names can contain letters, numbers, _ and $

```
document  getElementById  guest1  a_number  _text  $schema  $$
```

It's got names

```
for (var i = 0; i < 10; i++) {
  console.log("number " + i);
}
```

Names cannot start with a number

```
guest1  _2f  $3  4wheels
```

It's got names

```
for (var i = 0; i < 10; i++) {
  console.log("number " + i);
}
```

Names are case sensitive

```
my_name ≠ MY_NAME
```

It's got keywords

```
for (var i = 0; i < 10; i++) {
  console.log("number " + i);
}
```

Keywords are reserved names

```
await break case catch class const continue debugger default
delete do else enum export extends false finally for function
if implements import in instanceof interface let new null
package private protected return super switch this throw true
try typeof var void while with yield
```

It's got literals

```
for (var i = 0; i < 10; i++) {
  console.log("number " + i);
}
```

Literals represent values

```
null
undefined
true  false
123.45 1e-12  0xFEEDBEE5  123456789123456789n
"Hello, World!\n"  'Foo Bar Baz...'
{ name: "Jim", age: 20 }
[1, 2, 3]  ["apples", "kiwis", "bananas"]
```

It's got punctuation

```
for (var i = 0; i < 10; i++) {
  console.log("number " + i);
}
```

Lots of it!

Curly braces...

```
for (var i = 0; i < 10; i++) {
  console.log("number " + i);
}
```

… and semicolons

```
for (var i = 0; i < 10; i++) {
  console.log("number " + i)
}
```

You can skip semicolons, but it's not advisable to do so…

It's got comments, too

```
/* This is a small fragment of my script:
   It's a 'for' loop, guess what it does... */
for (var i = 0; i < 10; i++) {
  // For sure, it prints something to the console!
  console.log("number " + i);
}
```

They are ignored by the interpreter

What about whitespace?

Whitespace between names is mandatory

```
var x = 10;
var      y = 20;
var
  z = 30;
vark = 40;
```

Whitespace around punctuation is optional

```
for (var i = 0; i < 10; i++) {
  console.log(i * 2);
}

for(var i=0;i<10;i++){console.log(i*2);}
```

Strict mode

A JavaScript variant

```
"use strict"

/* Your code here... */
```

- Avoid *error-prone features*
- Semantical changes
- Additional error checking with *noisy* errors
- Some marginal performance gains

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

Example:

Hello, World!

- writing the code
- running
- debugging

Question:

Is HTML a programming language?

How does JavaScript work?

Execution model

- Scripts are lists of *statements*
- Statements are composed of
  - syntactical constructs
  - *expressions*
- Expressions are evaluated to produce
  - *values*
  - *side effects*
- Expressions are composed of
  - *operators*
  - more values or expressions

A simple example

```
console.log(5 * 10);
```

A simple example

```
console.log(5 * 10);
```

A statement...

A simple example

```
console.log(5 * 10);
```

... ended by a semicolon

A simple example

```
console.log(5 * 10);
```

An expression

A simple example

```
console.log(5 * 10);
```

It's made of an operator...

A simple example

```
console.log(5 * 10);
```

... and two literals, representing values!

A simple example

```
console.log(  50  );
```

The whole expression evaluates to a value...

A simple example

```
console.log(  50  );
```

... and is fed into a larger expression!

A simple example

```
undefined; // prints: 50
```

It produces both a *value* and a *side effect*!

Values

Values are carried by

- Literals
- Names
- Expressions

Names with associated values are called ***variables***

Types

Values can be of different types:

Primitive	Reference
Null	Object
Undefined	Function
Boolean	
Number	
String	
Symbol ES2015	
BigInt ES2020	

Type coercion

- Expressions can mix values of different types
- Values are *automatically* converted to a valid type
- Automatic conversions are called *type coercions*

```
"HAL " + 9000; // "HAL 9000"
"9" * [5]; // 45
```

Type coercions can be confusing

```
2 + 2 + "2"; // 42
```

Type coercion

- Expressions can mix values of different types
- Values are *automatically* converted to a valid type
- Automatic conversions are called *type coercions*

```
"HAL " + 9000; // "HAL 9000"
"9" * [5]; // 45
```

To avoid problems, use manual conversion

```
2 + 2 + Number("2"); // 6
```

Operators

Arithmetic	Increment	Bitwise
+	++	&
-	--	|
*		^
/		~
%		>>
** ES2016		>>>
		<<

Operators

Equality	Relational	Logical
`==`	`<`	`!`
`!=`	`<=`	`&&`
`===`	`>`	`||`
`!==`	`>=`	`??` ES2020
	`in`	
	`instanceof`	

Truthiness

In JavaScript, some values are considered **falsey**

```
undefined == false;
null == false;
"" == false;
0 == false;
```

All the others are **truthy**

Sometimes it gets very confusing...

```
"" == 0; // falsey
"0" == 0; // type coercion
```

Operators

Assignment

=	&=	&&= ES2021
+=	|=	||= ES2021
-=	^=	??= ES2021
*=	>>=	
/=	>>>=	
%=	<<=	
**= ES2016		

Operators

Dereference	Conditional	Special
`[ ]`	`? :`	`,`
`.`		`new`
`?.` ES2020		`delete`
		`void`
		`typeof`
		`await` ES2017

Statements

Variable declaration

```
var x = 10;
var y;
var z = 4, w = 12;
```

vars are *function scoped* . More on that later...

Code blocks

- Group statements together
- Seldom used alone
- Often used as part of other statements
- No trailing semicolon needed

```
{
  // statements...
}
```

Block scoped variables ES2015

```
const x = 3;
// x = 6; // TypeError: invalid assignment to const 'x'

let y = 4, z = 5;
{
    let x = 7;
    y = 8;
    console.log(x, y, z); // prints: 7 8 5
}
console.log(x, y, z); // prints: 3 8 5
```

Block scope is easier to reason about than `var`s

Always prefer `const` and `let`

Expression evaluation

```
y = x + 2;
console.log(y * z);
; // empty statement
```

Function declaration

```
function sum(a, b) {
  return a + b;
}

function logInfo(message) {
  console.log("INFO", message);
}
```

if / else

```
if (z > x) {
  console.log("Big z");
}

if (x > 20) {
  console.log("Very big x");
} else if (x > 10) {
  console.log("Big x");
} else {
  console.log("Small x");
}
```

switch / case

```
switch (name) {
  case "Jim":
  case "Joe":
    console.log("Hey, man!");
    break;

  case "Jane":
  case "Jill":
    console.log("Good morning, Madam");
    break;

  default:
    console.log("Hello dear guest");
}
```

Loops: `while` and `do / while`

```
while (x > 0) {
  console.log("until x becomes zero", x--);
}

while (true) console.log("infinite loop");

do {
  console.log("at least once")
} while (false);
```

Loops: for

```
for (var i = 0; i < 10; i++) {
  console.log("number", i);
}
```

Breaking statements

- `continue` = leave current loop iteration
- `break` = leave current loop
- `return` = leave current function
- `throw` = crash the script

Labeled statements

```
mainLoop: for (var i = 0; i < 100; i++) {
  for (var j = 0; j < 100; j++) {
    if (i * j > 1000) {
      break mainLoop;
    }
  }
}
```

Error handling

Some operations may *throw* errors

```
const nobody = null;
nobody.name; // TypeError: nobody is null

throw "some error"; // some error
throw new Error("another error"); // Error: another error
```

Thrown errors disrupt the flow of execution

Errors can be recovered by *catching* them

```
try {
    const nobody = null;
    nobody.name;
} catch (e) {
    console.error(e.message); // prints: nobody is null
}
```

Data structures

One name -> multiple values

Objects

- Take up some space in memory
- Aggregate several *properties*
- Each property is like an embedded variable
  - a *key* with an assigned *value*
- Objects can be *referenced* by variables

Object literals

```
const jerry = {
  name: "Jerry", // "name" = key, "Jerry" = value
  age: 33,
  skills: {  // Nested object value
    JavaScript: 5,
    HTML: 3,
    CSS: 3,
    "Node.js": 2, // Key is not a valid name (contains .)
  },
};
```

Property access

```
jerry.name; // "Jerry"
jerry["age"]; // 33
jerry.skills.JavaScript; // 5

const k = "JavaScript";
jerry.skills[k]; // 5

jerry.team; // undefined
```

```
jerry.age += 1; // set property 'age'
jerry.age; // 34

jerry.team = "A Team"; // set new property 'team'
jerry.team; // "A Team"
```

Null pointer dereferencing

```
let jimmy;
console.log(jimmy.name); // TypeError: jimmy is undefined
console.log(jimmy && jimmy.name); // would print: undefined
```

Always add *null-checks* when accessing object properties!

Optional chaining operator ES2020

```
let jimmy;
console.log(jimmy?.name); // prints: undefined

jimmy = { name: "Jimmy" };
console.log(jimmy?.name); // prints: Jimmy
console.log(jimmy.skills?.["Play guitar"]); // prints: undefined
```

A convenient syntax for marking access to optional properties

Object references

Objects are *referenced* by variables

```
const me = jerry;

me.age = 35;
jerry.age; // 35
```

References can be *shared*

Shared references point to ***the same*** object

Loops: `for / in`

Loop over an object's *keys*

```
for (let k in jerry) {
  console.log(k, jerry[k]);
}
```

Arrays

A kind of object holding an *indexed sequence* of values

Array literals

```
const numbers = [1, 2, 3, 4, 5, 6, 7];
const fruit = ["Apple", "Orange", "Peach", "Mango"];

const stuff = [0, true, [undefined, "dinosaur"], { value: 5 }];

const people = [
  { name: "John", job: "singer" },
  { name: "Paul", job: "guitarist" },
  { name: "George", job: "guitarist" },
  { name: "Ringo", job: "drummer" },
];
```

Element access

```
fruit[0]; // Apple
fruit[1]; // Orange
fruit[2]; // Peach
fruit[3]; // Mango

fruit.length; // 4

fruit[4]; // undefined
fruit.xyzzy; // undefined;
```

```
fruit[0] = "Kiwi"; // replace element
fruit; // [Kiwi, Orange, Peach, Mango]

fruit[4] = "Coconut"; // append element
fruit; // [Kiwi, Orange, Peach, Mango, Coconut]
fruit.length; // 5

fruit[7] = "Pineapple";
fruit; // [Kiwi, Orange, Peach, Mango, Coconut, , , Pineapple]
fruit.length; // 8
```

Array length

- The `length` property always tracks the actual length
- It can be read and written as well

```
fruit.length = 2; // truncate at length 2
fruit; // [Kiwi, Orange]

fruit.length = 0; // empty out array
fruit; // []
```

Destructuring ES2015

A convenient syntax to extract values from a data structure and assign them to names

Iterable destructuring

```
const numbers = [1, 2, 3, 4, 5, 6, 7];
const [x, y, ...rest] = numbers;
x; // 1
y; // 2
rest; // [3, 4, 5, 6, 7]
```

```
const message = "Hello";
const [a, b, , , c] = message;
a; // H
b; // e
c; // o
```

```
[a, b, c = "default"] = ["first"];
a; // first
b; // undefined
c; // default
```

Object destructuring

```
const jerry = {
  name: "Jerry",
  age: 33,
  interests: ["Surf", "Kendo", "Pizza"],
};

const { name, age, interests: hobbies } = jerry;
name; // Jerry
age; // 33
hobbies; // [Surf, Kendo, Pizza]
```

```
const { interests: [fun, moreFun, best] } = jerry;
fun; // Surf
moreFun; // Kendo
best; // Pizza

const { interests, ...other } = jerry;
interests; // [Surf, Kendo, Pizza]
other; // {name: Jerry, age: 33}

const { x, y = "default" } = jerry;
x; // undefined
y: // default
```

```
const key = "name";
const { [key]: value } = jerry;
value; // Jerry
```

```
const people = [
  { name: "John", jobs: ["singer", "songwriter"] },
  { name: "Paul", jobs: ["bass player", "guitarist"] },
  { name: "George", jobs: ["guitarist"] },
  { name: "Ringo", jobs: ["drummer"] },
];

const [{ name }, { jobs: [, job1] }, , { jobs: [job2] }] = people;

name; // John
job1; // guitarist
job2; // drummer
```

Spread syntax ES2015

A convenient syntax to expand an iterable in place of zero or more elements

```
const xs = [1, 2, 3], ys = [4, 5, 6];
const zs = [...xs, ...ys];
zs; // [1, 2, 3, 4, 5, 6]

const greeting = "Hello", recipient = "Kyle";
const message = [...greeting, " ", ...recipient];
message; // [H, e, l, l, o,  , K, y, l, e]
console.log(...message); // H e l l o   K y l e
```

Object spread ES2018

```
const personalData = { name: "Margaret", age: 72 }
const info = {
  ...personalData,
  hobbies: ["Baking cakes", "Skydiving"],
};
info; // { name: Margaret, age: 72, hobbies: [...] }
```

```
const options = { color: "blue", width: "50%" };
options = {
  color: "black", width: "100%", height: "300px",
  ...options,
};
options; // { color: blue, width: 50%, height: 300px }
```

Question:

Why do we need data structures?

Functions

Are executable objects made of reusable instructions

Declaration

```
function hello() { // function name
  // function body
  console.log("Hello, World!");
}
```

- It's similar to a variable declaration
- Binds a name (`hello`) to a function

Invocation

Execute the function body

```
hello(); // prints: Hello, World!
```

Parameters

```
function sayHello(from, to) {
  console.log(from, "says hello to", to);
}
```

Parameters are like input variables

```
sayHello("Joe", "Jody"); // prints: Joe says hello to Jody
```

Arguments are values we *pass into* a function invocation

```
sayHello(); // prints: undefined says hello to undefined
```

If not given, they default to `undefined`

Default parameters ES2015

```
function sayHello(from="He", to="her") {
  console.log(from, "says hello to", to);
}

sayHello("Joe", "Jody"); // prints: Joe says hello to Jody
sayHello("Joe"); // prints: Joe says hello to her
sayHello(); // prints: He says hello to Jody
```

Rest parameters ES2015

Easily define *variadic* functions

```
function sayHello(from, ...to) {
    to instanceof Array;  // true
    for (let i = 0; i < to.length; i++) {
        console.log(from, "says hello to", to[i]);
          /* prints: John says hello to George
                     John says hello to Paul
                     John says hello to Ringo */
    }
}

sayHello("John", "George", "Paul", "Ringo");
```

Spread arguments ES2015

Easily *apply* functions to iterable collections of values

```
sayHello(...["Jake", "Elwood"]);
                    // prints: Jake says hello to Elwood

// strings are iterable, too!
sayHello(..."ABC"); /* prints: A says hello to B
                               A says hello to C */
```

Return value

```
function constant42() {
  return 42;
}
const theAnswer = constant42(); // 42
```

```
function sum(a, b) {
  return a + b;
}
const total = sum(1, sum(2, 3)); // 6
```

Invocation is a kind of expression

```
function helloTo(who) {
  who = who || "World";
  return "Hello, " + who + "!";
}

console.log(helloTo("Jason")); // Hello, Jason!
console.log(helloTo()); // Hello, World!
```

Default return value is undefined

```
function printHelloTo(who) {
  console.log(helloTo(who));
}

const result = printHelloTo("Jason"); // prints: Hello, Jason!
result; // undefined
```

Functions are values, too

```
typeof hello === "function";

const greeting = hello;

typeof greeting === "function";
greeting === hello;
```

Higher order functions

```
function doTwice(action) {
  action();
  action();
}
function printMessage() {
  console.log("Jeez, Louise");
};
doTwice(printMessage); // prints message twice
```

Functions can be *consumed* as arguments

Higher order functions

```
function makeDoTwice(action) {
  return function() {
    action();
    action();
  };
}
const twiceTheMessage = makeDoTwice(printMessage);
twiceTheMessage(); // prints message twice
```

Functions can be *consumed* as arguments or *returned* by other functions!

Function literals

(a.k.a. *anonymous functions, function expressions, lambda expressions*)

```
const multiply = function(a, b) {
  return  a * b;
};
```

They make certain coding patterns easier...

… like some functional programming idioms

```
function filter(array, predicate) {
  const result = [];
  for (let i = 0; i < array.length; i++) {
    if (predicate(array[i]))
      result[result.length] = array[i];
  }
  return result;
}

filter([1, 2, 3, 4, 5], function(x) { return x < 3 }); // [1, 2]
```

Arrow functions ES2015

```
const logValue = x => {
    console.log(x);
    return x;
};

const sum = (a, b) => { return a + b; };
const mult = (a, b) => a * b; // implicit return
```

A less verbose literal function syntax...

... makes using functional idioms more readable.

```
// regular function syntax
filter([1, 2, 3, 4, 5], function(x) { return x < 3 });

// arrow function syntax
filter([1, 2, 3, 4, 5], x => x < 3);
```

More features explained later...

Methods

```
const actor = {
  act: function() {
    console.log("To be or not to be?");
  },
};

actor.act(); // prints: To be or not to be

actor.greet = hello; // method assignment
actor.greet(); // method invocation
```

A ***method*** is a function assigned to an object's property

JavaScript APIs

Interfaces to functionality available to JavaScript code

ECMAScript Core API

- General purpose functionality
- Embedded in every ECMAScript implementation

String manipulation, arrays, collections, dates, regular expressions, objects, functions, ...

Web APIs

- Client-side, browser-specific functionality
- Defined by the HTML 5 standard

DOM, AJAX, Web Storage, Web Workers, Canvas, ...

Node.js APIs

- Server-side functionality
- Defined by the Node.js specification

File system, IO streams, networking, OS, processes ...

Third-party APIs

- Libraries
- Frameworks
- Interfaces to external services

ECMAScript Core API

- Utility functions
- Fundamentals: `Object`, `Function`, `Boolean`, `Symbol`
- Numbers & time: `Number`, `BigInt`, `Math`, `Date`
- Text processing: `String`, `RegExp`
- Indexed sequences: `Array`, typed arrays
- Keyed collections: `Map`, `Set`, weak collections
- Structured data: `JSON`
- Error objects: `Error`
- Async control abstraction: `Promise`
- Internationalization: `Intl`

developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

Web APIs

Document Object Model (DOM)

Dynamically manipulate structured documents

DOM tree

- The document is modeled as a *tree*
- The tree is composed of *nodes*
- Nodes have:
  - 0..1 *parent*
  - 0..* *children*

Nodes

- Nodes are represented by JavaScript *objects*
- We can
  - *Navigate* the tree to access nodes
  - *Modify* node *attributes* and *properties*
  - *Add* and *remove* nodes to the tree

Nodes - Navigation

- **`document`**
  The entry point to the DOM
- **`document.documentElement`**
  The *root* node of the document tree
- **`document.head`**
  The HTML *head* node
- **`document.body`**
  The HTML *body* node

Nodes - Navigation

- **`node.childNodes`**
  A *live* array-like object listing `node`'s children
- **`node.firstChild`**
  The first of `node`'s children, or `null` if `node` has no children
- **`node.lastChild`**
  The last of `node`'s children, or `null` if `node` has no children
- **`node.hasChildNodes()`**
  Returns `true` if `node` has at least one child

Nodes - Navigation

- **`node.parentNode`**
  `node`'s parent node, or `null` if `node` is the document root
- **`node.nextSibling`**
  The node immediately after `node` in the parent's `childNodes`, or `null` if `node` is the last child
- **`node.previousSibling`**
  The node immediately before `node` in the parent's `childNodes`, or `null` if `node` is the first child

Nodes - Navigation

- **`node.contains(otherNode)`**
  Returns `true` if `otherNode` is a descendant of `node` (or `node` itself)
- **`node.isConnected`**
  `true` if `node` is actually a descendant of a document or document fragment
- **`node.ownerDocument`**
  The document or document fragment of which `node` is a descendant

Nodes - Navigation

- **`node.compareDocumentPosition(otherNode)`**
  Returns a number describing the position of `otherNode` relative to `node`, as bitwise composition of:
  - `Node.DOCUMENT_POSITION_DISCONNECTED`
  - `Node.DOCUMENT_POSITION_PRECEDING`
  - `Node.DOCUMENT_POSITION_FOLLOWING`
  - `Node.DOCUMENT_POSITION_CONTAINS`
  - `Node.DOCUMENT_POSITION_CONTAINED_BY`

Nodes - Manipulation

- **`node.textContent`** Ⓡⓦ
  The text contained in `node` and its children
- **`node.cloneNode(deep)`**
  Return a copy of `node`. If `deep` is `true`, copy descendant nodes too.
  `deep` is optional, but it's *safer* to always provide it.

Nodes - Manipulation

- **`node.appendChild(newChild)`**
  - Detaches `newChild` from its parent
  - Attaches `newChild` to `node` as last child
- **`node.insertBefore(newChild, childNode)`**
  - Detaches `newChild` from its parent
  - Attaches `newChild` to `node` as `childNode`'s previous sibling

Nodes - Manipulation

- **`node.removeChild(oldChild)`**
  - Detaches `oldChild` from `node`
  - Returns `oldChild`
- **`node.replaceChild(newChild, oldChild)`**
  - Detaches `newChild` from its parent
  - Attaches `newChild` to `node` in place of `oldChild`
  - Detaches `oldChild` from `node`
  - Returns `oldChild`

Nodes - Type

- **`node.nodeType`**

  A number representing node's type. One of:

  - `Node.ELEMENT_NODE`
  - `Node.TEXT_NODE`
  - `Node.CDATA_SECTION_NODE`
  - `Node.PROCESSING_INSTRUCTION_NODE`
  - `Node.COMMENT_NODE`
  - `Node.DOCUMENT_NODE`
  - `Node.DOCUMENT_TYPE_NODE`
  - `Node.DOCUMENT_FRAGMENT_NODE`

Elements

- A kind of DOM node
  - The ones we are usually interested in
- Support a higher-level API and unique features
  - Attributes
  - Positioning
  - CSS style
  - …

Elements - Navigation

- **`element.children`**
  A *live* array-like object listing `element`'s child elements
- **`element.firstElementChild`**
  The first item of `element.children`, or `null` if empty
- **`element.lastElementChild`**
  The last item of `element.children`, or `null` if empty
- **`element.childElementCount`**
  The number of `element`'s child elements

Elements - Navigation

- **`node.parentElement`**
  `node`'s parent element, or `null` if `node`'s parent is not an element
- **`element.nextElementSibling`**
  The element immediately after `element` in the parent's `children`, or `null` if `element` is the last child
- **`element.previousElementSibling`**
  The element immediately before `element` in the parent's `children`, or `null` if `element` is the first child

Elements - Querying

- **`document.getElementById(id)`**
  Returns the element with the "id" attribute equal to `id`, or `null` if not found
- **`element.getElementsByClassName(className)`**
  Returns a *live* array-like object of `element`'s descendants with the "class" attribute containing `className` as whole word
- **`element.getElementsByTagName(tagName)`**
  Returns a *live* array-like object of `element`'s descendants with html tag name equal to `tagName`

Elements - Querying

- **`element.querySelector(cssSelector)`**
  Returns the first of `element`'s descendants matching the given `cssSelector`, or `null` if not found
- **`element.querySelectorAll(cssSelector)`**
  Returns an array-like object of `element`'s descendants matching the given `cssSelector`

Elements - Creation

- **`document.createElement(tagName)`**
  Returns a new disconnected element with the given `tagName`

Elements - Attributes

Most attributes are directly accessible as (RW) properties of the element object

Elements - Attributes

- **`element.className`** (RW)
  The full value of `element`'s "class" attribute
- **`element.classList`**
  A *live*, *rich* array-like object containing all words of `element.className`
  - **`element.classList.contains(class)`**
  - **`element.classList.add(class)`**
  - **`element.classList.remove(class)`**
  - **`element.classList.toggle(class)`**
  - **`element.classList.replace(oldCls, newCls)`**

Elements - Attributes

- **`element.dataset`**
  A *live* object dictionary of `element`'s custom "data-" attributes. The dictionary's *properties*
  - are stripped of the "data-" prefix
  - are converted from `kebab-case` to `camelCase`

Elements - Attributes

Use the following methods for advanced use cases or to manipulate non-standard attributes

Elements - Attributes

- **`element.attributes`**
  A live array-like object of `element`'s attributes
- **`attribute.name`**
  `attribute`'s key
- **`attribute.value`**
  `attribute`'s value
- **`element.hasAttributes()`**
  Returns `true` if `element.attributes` is not empty

Elements - Attributes

- **`element.getAttributeNames()`**
  Returns an array containing the names of `element`'s attributes
- **`element.getAttribute(key)`**
  Returns the value of `element`'s attribute whose name equals `key`
- **`element.hasAttribute(key)`**
  Returns `true` if `element` has an attribute whose name equals `key`

Elements - Attributes

- **`element.setAttribute(key, value)`**
  Sets the value of `element`'s attribute whose name equals `key` to `value`
- **`element.toggleAttribute(key, value?)`**
  Toggles `element`'s boolean attribute whose name equals `key`. If `value` is provided, it forces the attribute's state instead
- **`element.removeAttribute(key)`**
  Removes `element`'s attribute whose name equals `key`

Elements - Manipulation

- **`element.innerText`** (RW)
  The rendered text of `element` and its children
- **`element.innerHTML`** (RW)
  The HTML source text of `element`'s children
- **`element.outerHTML`** (RW)
  The HTML source text of `element` and its children

Setting HTML properties causes a new *subtree* to be parsed and attached in place of the old content

This can have security implications

Elements - Manipulation

- **`element.insertAdjacentElement(pos, newElem)`**
  Inserts `newElem` in position `pos` relative to `element`
- **`element.insertAdjacentHTML(pos, htmlSource)`**
  Inserts a subtree parsed from `htmlSource` in position `pos` relative to `element`
- **`element.insertAdjacentText(pos, text)`**
  Inserts `text` in position `pos` relative to `element`

`pos` must be one of: `"beforebegin"`, `"afterbegin"`, `"beforeend"` and `"afterend"`

Elements - Manipulation

- **`element.before(...newSiblings)`** DOM4
  Adds `newSiblings` as preceding siblings of `element`
- **`element.after(...newSiblings)`** DOM4
  Adds `newSiblings` as subsequent siblings of `element`
- **`element.replaceWith(...newSiblings)`** DOM4
  Detaches `element` and adds `newSiblings` in place of it
- **`element.remove()`** DOM4
  Detaches `element`

`newSiblings` can contain both elements and text strings

Elements - Manipulation

- **`element.append(...newChildren)`** DOM4
  Adds `newChildren` at the end of `element`'s children list
- **`element.prepend(...newChildren)`** DOM4
  Adds `newChildren` at the start of `element`'s children list

`newChildren` can contain both elements and text strings

Events

- Web page objects can be *target* of *events*
- Events are *dispatched* on various occurrences:
  - Loading
  - Rendering
  - Media playback
  - User interaction
  - …

Listeners

- A *listener* can be attached to a target, declaring
  - an *event type*
  - a *callback* function
- When an event of the requested type is dispatched to the target, the callback is executed

Listener callback

```
function logEvent(evt) {
  console.log(
    evt.type,
    evt.target,
    evt, // many more properties
  );
}
```

Declaring listeners

As HTML attributes - *bad practice*

```
<form>
  <button id="log_event" onclick="logEvent(event);">
    Echo text
  </button>
</form>
```

Declaring listeners

As DOM element properties

```
const button = document.getElementById("log_event");
button.onclick = logEvent;
```

At most one listener per target

Declaring listeners

- **`elem.addEventListener(type, cb, opts?)`**
  Attaches a listener to target `elem` for event type `type`. Will call function `cb` when the event fires. `opts` can be:
  - omitted
  - a boolean, equivalent to setting the `capture` flag
  - an object[DOM4] with properties:
    - `capture`: if `true`, enables capture mode
    - `once`: if `true`, the listener is invoked at most once
    - `passive`: if `true`, the event cannot be cancelled

Event bubbling

When an event occurs:

1. The innermost target is notified
2. The target's parent is notified
3. The target's ancestors are notified in turn, all the way up to `window`

Capture mode

When an event occurs in *capture mode*:

1. The outermost *capturing target* is notified
2. Descendant capturing targets are notified in turn, down to the innermost target
3. Normal event bubbling resumes from the innermost *non-capturing target*
4. The target's non-capturing ancestors are notified in turn, all the way up to `window`

Stopping event bubbling

```html
<div id="outer">
  OUTER
  <div id="inner">INNER</div>
</div>
```

```javascript
const outer = document.getElementById("outer");
const inner = document.getElementById("inner");

outer.onclick = function() {
  console.log("outer");
};
inner.onclick = function(e) {
  e.stopPropagation(); // IE 9+
  e.cancelBubble = true; // IE 8-
  console.log("inner");
};
```

Cancelling events

```
<a href="https://stackoverflow.com" id="link">Click me!</a>
```

```
const link = document.getElementById("link");
link.onclick = function(e) {
  e.preventDefault(); // IE 9+
  e.returnValue = false; // IE 8-
};
```

Events - more APIs

- **`elem.removeEventListener(type, cb, opts?)`**
  Detaches an existing listener from `elem`, matching `type`, `cb` and `opts`
- **`elem.dispatchEvent(event)`**
  Sends a synthetic `event` to `elem`
- **`new CustomEvent(type, { detail })`**
  Constructs a custom event object of a specific `type`. Custom event data passed as `detail` will be available on the event object's "detail" key

Exercise

On button click, retrieve value from input