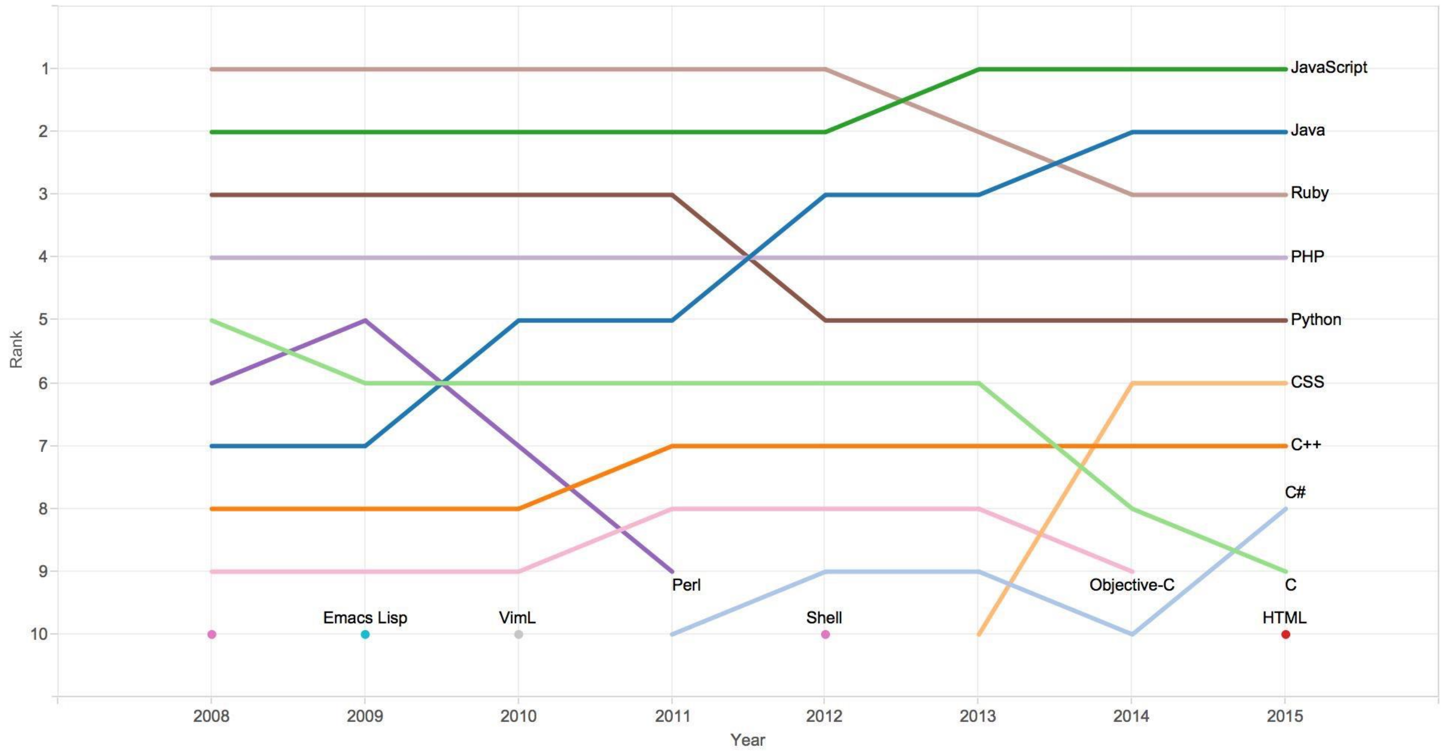
JavaScript Fundamentals

Roi Yehoshua
2018

JavaScript

- **The** programming language of the future?

Rank of top languages on GitHub.com over time



Source: GitHub.com

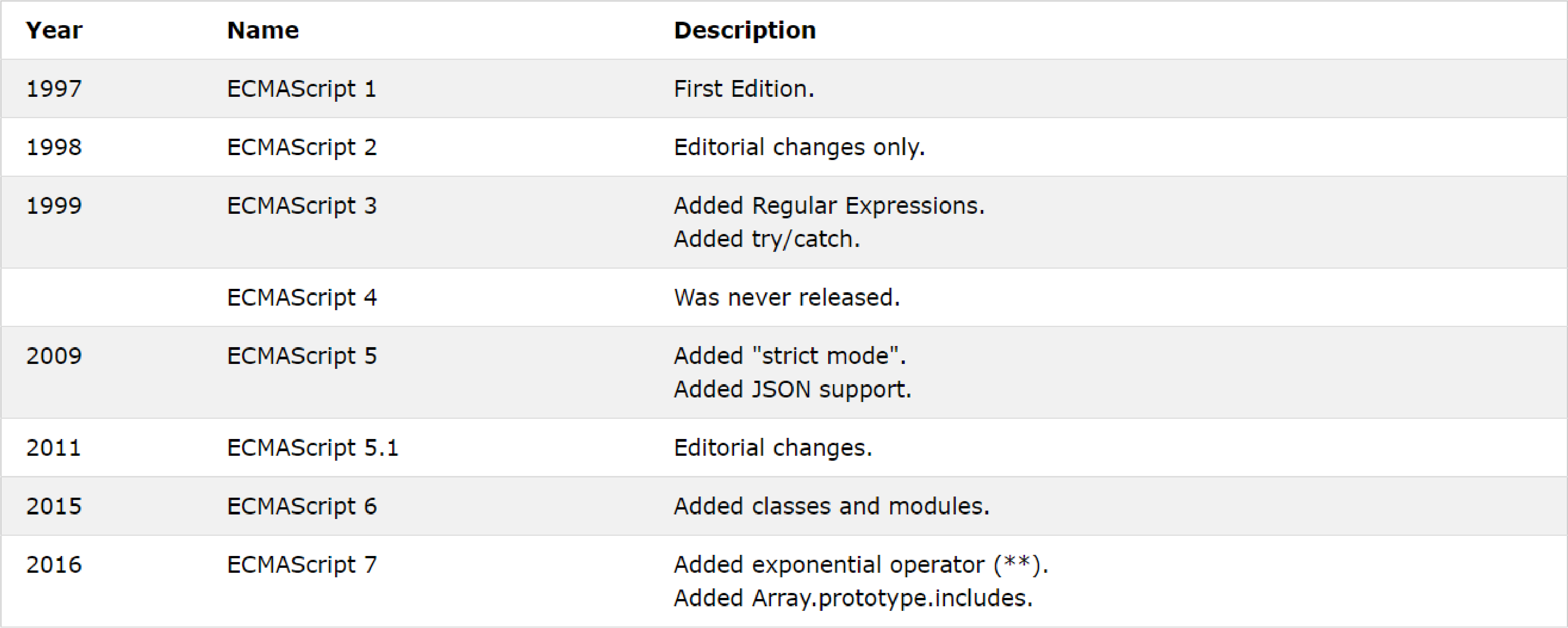
JavaScript

- Created In 1995 by Brendan Eich as a scripting language for Netscape Navigator
- Standardized as ECMAScript in 1997
- JavaScript (often abbreviated as JS) enables interactive web pages and thus is an essential part of web applications
- Initially created as a browser-only language, but now it is used in many other environments as well:
  - HTML5 mobile apps
  - Server side development (NodeJS)
  - JS on devices – the internet of things
    - Huge potential of running JavaScript on embedded devices

JavaScript Main Features

- Interpreter based (no compilation) scripting language
- Loosely typed and dynamic language
- Uses syntax influenced by that of Java
  - However, has very different semantics than Java
- Main components
  - The Core (ECMAScript)
  - The DOM (Document Object Model)
  - The BOM (Browser Object Model)

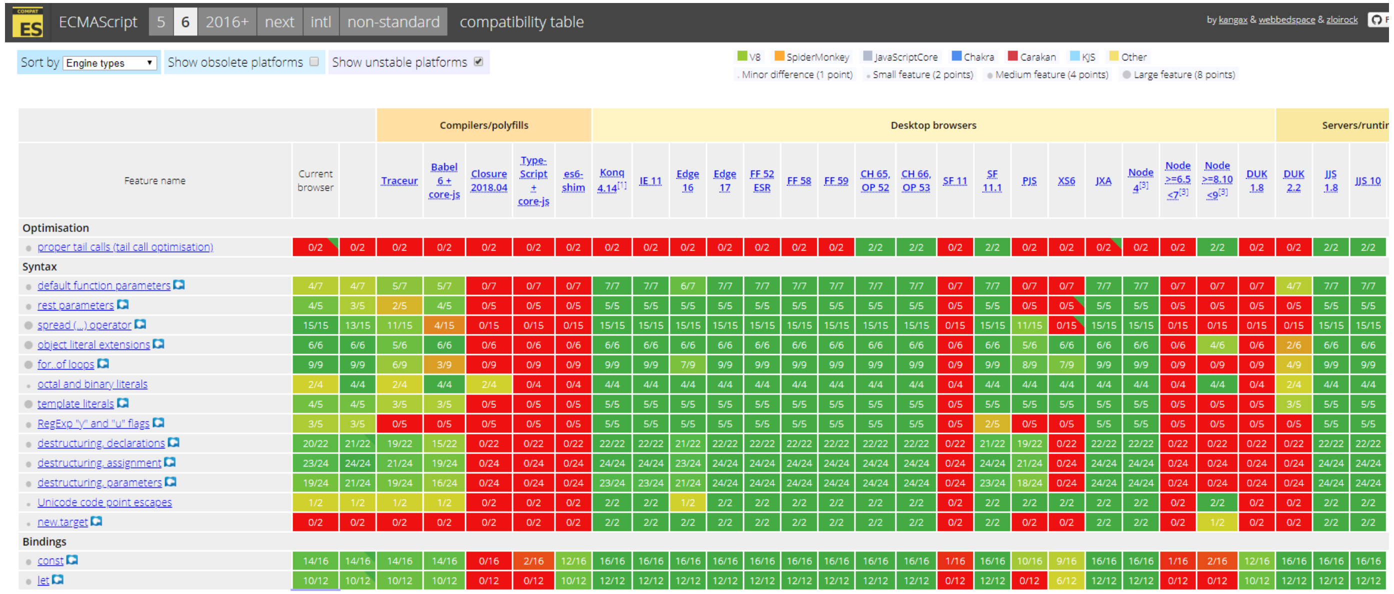
JavaScript Versions



Year	Name	Description
1997	ECMAScript 1	First Edition.
1998	ECMAScript 2	Editorial changes only.
1999	ECMAScript 3	Added Regular Expressions. Added try/catch.
	ECMAScript 4	Was never released.
2009	ECMAScript 5	Added "strict mode". Added JSON support.
2011	ECMAScript 5.1	Editorial changes.
2015	ECMAScript 6	Added classes and modules.
2016	ECMAScript 7	Added exponential operator (**). Added Array.prototype.includes.

Browser Compatibility

- The JavaScript language steadily evolves
- Teams behind JavaScript engines have their own ideas about what and when to implement new standards of the language
- A good page to see the current state of support for language features is https://kangax.github.io/compat-table/es6/



ES
ECMAScript 5 6 2016+ next intl non-standard compatibility table
by kangax & webbedspace & zloirock

Sort by Engine types ▾ Show obsolete platforms □ Show unstable platforms ☑

🟩 V8 🟧 SpiderMonkey ⬜ JavaScriptCore 🟦 Chakra 🟥 Carakan 🟦 KJS 🟨 Other
● Minor difference (1 point) ● Small feature (2 points) ● Medium feature (4 points) ● Large feature (8 points)

			Compilers/polyfills					Desktop browsers																			Servers/runtir		
Feature name	Current browser		Traceur	Babel 6 + core-js	Closure 2018.04	Type-Script + core-js	es6-shim	Konq 4.14[1]	IE 11	Edge 16	Edge 17	FF 52 ESR	FF 58	FF 59	CH 65, OP 52	CH 66, OP 53	SF 11	SF 11.1	PJS	XS6	JXA	Node 4[3]	Node >=6.5 <7[3]	Node >=8.10 <9[3]	DUK 1.8	DUK 2.2	JJS 1.8	JJS 10
Optimisation																												
proper tail calls (tail call optimisation)	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	0/2	2/2	0/2	0/2	0/2	0/2	0/2	2/2	0/2	0/2	2/2	2/2
Syntax																												
default function parameters	4/7	4/7	5/7	5/7	0/7	0/7	0/7	7/7	7/7	6/7	7/7	7/7	7/7	7/7	7/7	7/7	0/7	7/7	0/7	0/7	7/7	7/7	0/7	0/7	0/7	4/7	7/7	7/7
rest parameters	4/5	3/5	2/5	4/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	0/5	5/5	0/5	0/5	5/5	5/5	0/5	0/5	0/5	0/5	5/5	5/5
spread (...) operator	15/15	13/15	11/15	4/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	0/15	15/15	11/15	0/15	15/15	15/15	0/15	0/15	0/15	0/15	15/15	15/15
object literal extensions	6/6	6/6	5/6	6/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	0/6	6/6	5/6	6/6	6/6	6/6	0/6	4/6	0/6	2/6	6/6	6/6
for..of loops	9/9	9/9	6/9	3/9	0/9	0/9	0/9	9/9	9/9	7/9	9/9	9/9	9/9	9/9	9/9	9/9	0/9	9/9	8/9	7/9	9/9	9/9	0/9	0/9	0/9	4/9	9/9	9/9
octal and binary literals	2/4	4/4	2/4	4/4	2/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	0/4	4/4	4/4	4/4	4/4	4/4	0/4	4/4	0/4	2/4	4/4	4/4
template literals	4/5	4/5	3/5	3/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	0/5	5/5	5/5	5/5	5/5	5/5	0/5	0/5	0/5	3/5	5/5	5/5
RegExp "y" and "u" flags	3/5	3/5	0/5	0/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	0/5	2/5	0/5	0/5	5/5	5/5	0/5	0/5	0/5	0/5	5/5	5/5
destructuring, declarations	20/22	21/22	19/22	15/22	0/22	0/22	0/22	22/22	22/22	21/22	22/22	22/22	22/22	22/22	22/22	22/22	0/22	21/22	19/22	0/22	22/22	22/22	0/22	0/22	0/22	0/22	22/22	22/22
destructuring, assignment	23/24	24/24	21/24	19/24	0/24	0/24	0/24	24/24	24/24	23/24	24/24	24/24	24/24	24/24	24/24	24/24	0/24	24/24	21/24	0/24	24/24	24/24	0/24	0/24	0/24	0/24	24/24	24/24
destructuring, parameters	19/24	21/24	19/24	16/24	0/24	0/24	0/24	23/24	23/24	21/24	24/24	24/24	24/24	24/24	24/24	24/24	0/24	23/24	18/24	0/24	24/24	24/24	0/24	0/24	0/24	0/24	24/24	24/24
Unicode code point escapes	1/2	1/2	1/2	1/2	0/2	0/2	0/2	2/2	2/2	1/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	2/2	2/2	2/2	2/2	2/2	0/2	2/2	0/2	0/2	2/2	2/2
new.target	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	2/2	0/2	0/2	2/2	2/2	0/2	1/2	0/2	0/2	2/2	2/2
Bindings																												
const	14/16	14/16	14/16	14/16	0/16	2/16	12/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	1/16	16/16	10/16	9/16	16/16	16/16	1/16	2/16	12/16	16/16	16/16	16/16
let	10/12	10/12	10/12	10/12	0/12	0/12	10/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	0/12	12/12	0/12	6/12	12/12	12/12	0/12	0/12	10/12	12/12	12/12	12/12

In-Browser JavaScript

- In-browser JavaScript can do everything related to webpage manipulation, interaction with the user and the web server. For instance, in-browser JS is able to:
  - Add new HTML to the page, change the existing content, modify styles
  - React to user actions, run on mouse clicks, pointer movements, key presses
  - Send requests over the network to remote servers, download and upload files (AJAX)
  - Remember the data on the client-side (“local storage”)
- However, JavaScript in the browser is limited for the sake of the user’s safety:
  - JavaScript on a webpage may not read/write arbitrary files on the hard disk, copy them or execute programs
  - JavaScript has no direct access to OS system functions
  - JavaScript from one page may not access another page if they come from different sites
    - This is called the “Same Origin Policy”

JavaScript Engines

- A **JavaScript engine** is a program or interpreter which executes JavaScript code
- Web browsers have an embedded JavaScript engine, aka “JavaScript virtual machine”
- Different engines have different “codenames”, for example:
  - V8 - Chrome and Opera
  - SpiderMonkey - Firefox
  - ChakraCore - Microsoft Edge
- A JavaScript engine workflow consists of the following stages:
  - The engine reads (“parses”) the script
  - It converts (“compiles”) the script to the machine language
  - Then the machine code runs, pretty fast
- The engine applies optimizations on every stage of the process

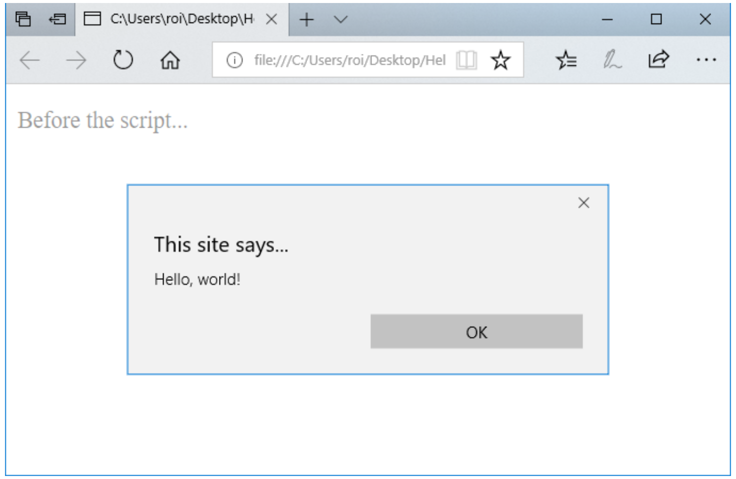
Code Editors

- IDEs (Integrated Development Environment) offer a full-scale “development environment”
- IDEs for frontend development:
  - Visual Studio - a free version is available (Visual Studio Community), works only on Windows
  - WebStorm
  - Netbeans
- “Lightweight editors” are not as powerful as IDEs, but they’re fast, elegant and simple
- The following options deserve your attention:
  - Visual Studio Code (cross-platform, free)
  - Atom (cross-platform, free)
  - Sublime Text (cross-platform, shareware)
  - Notepad++ (Windows, free)
  - Vim and Emacs are also cool, if you know how to use them

The <script> tag

- JavaScript programs can be inserted in any part of an HTML document with the help of the <script> tag

```
<html>
<head>
    <title>My First Script</title>
</head>
<body>
    <p>Before the script...</p>
    <script>
        alert('Hello, world!');
    </script>
    <p>...After the script.</p>
</body>
</html>
```



Before the script...

This site says...

Hello, world!

  - The <script> tag contains JavaScript code which is automatically executed when the browser meets the tag
- Current practice often places it just before the closing body tag

Developer Console

- Code is prone to errors
- But in the browser, a user doesn’t see the errors by default. So, if something goes wrong in the script, we won’t see what’s broken and can’t fix it.
- To see errors and get a lot of other useful information about scripts, browsers have embedded “developer tools”
- Most often developers lean towards Chrome or Firefox for development, because those browsers have the best developer tools

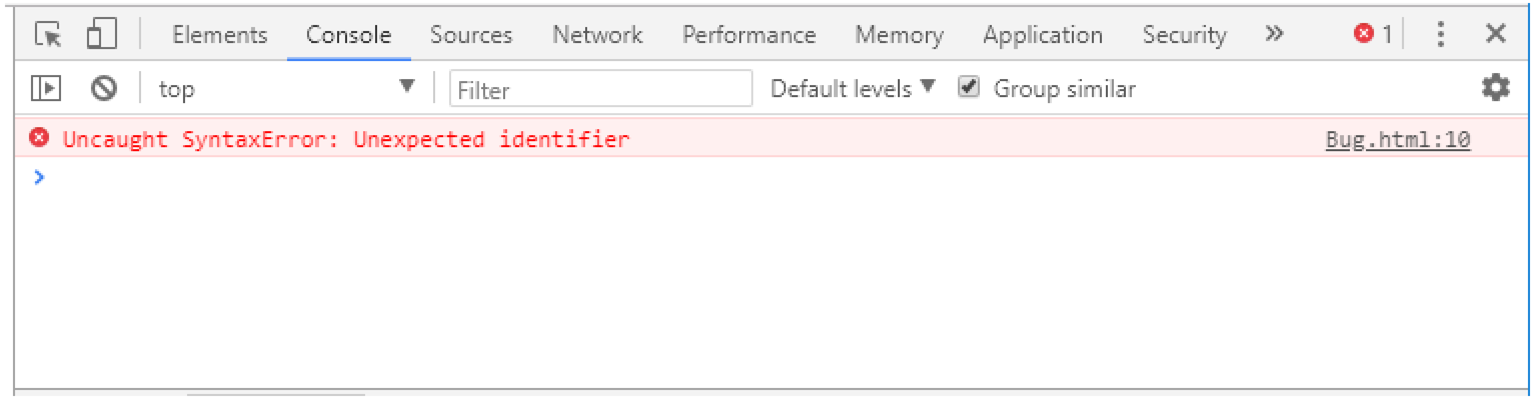
Developer Console

- Create the following page:

```
<html>
<head>
    <title>Buggy page</title>
</head>
<body>
    There is an error in the script on this page.
    <script>
        bla bla
    </script>
</body>
</html>
```

- Open it in the browser
- There's an error in the JavaScript code on it
- It's hidden from a regular visitor's eyes, so let's open developer tools to see it
- Press F12 or, if you're on Mac, then Cmd+Opt+J
- The developer tools will open on the Console tab by default

Developer Console



```
Uncaught SyntaxError: Unexpected identifier                                   Bug.html:10
>
```

- Here we can see the red-colored error message
  - In this case the script contains an unknown “bla bla” command
- On the right, there is a clickable link to the source bug.html:10 with the line number where the error has occurred
- Below the error message there is a blue > symbol.
  - It marks a “command line” where we can type JS commands and press Enter to run them

External Scripts

- As a rule, only the simplest scripts are put into HTML
- More complex ones reside in separate .js files
  - The benefit of a separate file is that the browser will download it and then store in its cache
- The script file is attached to HTML with the src attribute:

```
<html>
<head>
    <title>External Script</title>
</head>
<body>
    <script src="/scripts/myapp.js"></script>
</body>
</html>
```

  - Here /scripts/MyScript.js is an absolute path to the script file (from the site root)
  - It is also possible to provide a path relative to the current page
    - For instance, src="myapp.js" would mean a file "myapp.js" in the current folder

External Scripts

- To attach several scripts, use multiple tags:

```
<script src="/scripts/script1.js"></script>
<script src="/scripts/script2.js"></script>
```

- A single <script> tag can't have both the src attribute and the code inside
- This won't work:

```
<script src="file.js">
    alert(1); // the content is ignored, because src is set
</script>
```

- The example above can be split into two scripts to work:

```
<script src="file.js"></script>
<script>
    alert(1);
</script>
```

Code Structure

- Statements in JavaScript are separated by a semicolon

```
alert('Hello'); alert('World');
```

- Usually each statement is written on a separate line – thus the code becomes more readable:

```
alert('Hello');
alert('World');
```

- A semicolon may be omitted in most cases when a line break exists
  - This would also work:

```
alert('Hello')
alert('World')
```

- However, it's highly recommended to put semicolons between statements even if they are separated by newlines

Comments

- Comments can be put into any place of the script
- One-line comments start with two forward slash characters //
  - The rest of the line is a comment. It may occupy a full line of its own or follow a statement.

```
// This comment occupies a line of its own
alert('Hello');

alert('World'); // This comment follows the statement
```

- Multiline comments start with a forward slash and an asterisk /*, and end with an asterisk and a forward slash */

```
/* An example with two messages.
    This is a multiline comment.
*/
alert('Hello');
alert('World');
```

- Please, don’t hesitate to comment your code

Variables

- A variable is a “named storage” for data
- To create a variable in JavaScript, we need to use the let keyword
- The statement below creates (*declares* or *defines*) a variable named “message”:

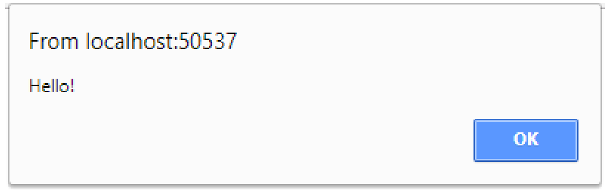
```
let message;
```

- Now we can put some data into it by using the assignment operator =

```
let message;
message = 'Hello!';
```

- The string is now saved into the memory area associated with the variable
- We can access it using the variable name:

```
alert(message); // shows the variable content
```


From localhost:50537

Hello!

Variables

- To be concise we can merge the variable declaration and assignment into a single line:

```
let message = 'Hello!';
```

- We can also declare multiple variables in one line:

```
let user = 'John', age = 25, message = 'Hello';
```

  - That might seem shorter, but it’s not recommended. For the sake of better readability, please use a single line per variable.

The Old “var”

- In older scripts you may also find another keyword: var instead of let

```
var message = 'Hello!';
```

- The var keyword is *almost* the same as let
- There are two main differences of var:
  - Variables have no block scope, they are visible minimum at the function level
  - Variable declarations are processed at function start
- These differences are actually a bad thing most of the time
- So, for new scripts var is used exceptionally rarely

Variable Naming

- There are two limitations for a variable name in JavaScript:
  - The name must contain only letters, digits, symbols $ and _
  - The first character must not be a digit
- When the name contains multiple words, camelCase is commonly used
  - i.e., words go one after another, each word starts with a capital letter: myVeryLongName
- JavaScript is case-sensitive, e.g., variables named apple and Apple are two different variables
- There is a list of reserved words, which cannot be used as variable names, because they are used by the language itself.
  - For example, the words let, class, return, function are reserved.
- Please name the variables sensibly. Take time to think if needed.
- Variable naming is one of the most important and complex skills in programming

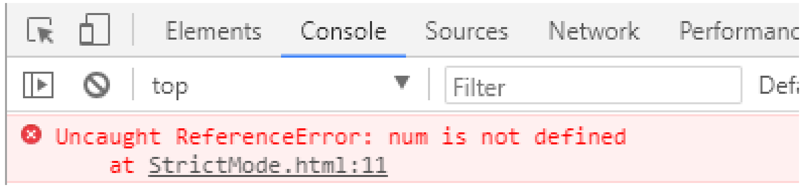
Strict Mode

- Normally, we need to define a variable before using it
- But in the old times, it was technically possible to create a variable by a mere assignment of the value, without let

```
num = 5; // the variable "num" is created if didn't exist
alert(num); // 5
```

- **Strict mode** is a way to introduce better error-checking into your code
- You can declare strict mode by adding "use strict"; at the beginning of a file, a program, or a function
- When you use **strict mode**, you cannot, for example, use implicitly declared variables

```
"use strict";
num = 5; // error: num is not defined
```



```
Uncaught ReferenceError: num is not defined
    at StrictMode.html:11
```

Constants

- To declare a constant (unchanging) variable, use const instead of let:

```
const message = 'hello';
message = 'bye'; // error, can't reassign the constant!
```

- There is a widespread practice to use constants as aliases for difficult-to-remember values that are known prior to execution
- Such constants are named using capital letters and underscores
- Example:

```
const COLOR_GREEN = '#0F0';
const COLOR_BLUE = '#00F';
const COLOR_ORANGE = '#FF7F00';

// ...when we need to pick a color
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Exercise (1)

- Declare two variables: admin and name
- Assign the value "John" to name
- Copy the value from name to admin
- Show the value of admin using alert (must output “John”)

Data Types

- A variable in JavaScript can contain any data
- A variable can at one moment be a string and later receive a numeric value:

```
let foo = 42;     // foo is now a number
foo = 'bar';      // foo is now a string
foo = true;       // foo is now a boolean
```

- Programming languages that allow such things are called “**dynamically typed**”
  - Meaning that there are data types, but variables are not bound to any of them

Data Types

- There are seven basic data types in JavaScript:
  - **number** for numbers of any kind: integer or floating-point
  - **string** for strings
    - A string may have one or more characters, there’s no separate single-character type
  - **boolean** for true/false
  - **null** for unknown values
    - a standalone type that has a single value null
  - **undefined** for unassigned values
    - a standalone type that has a single value undefined
  - **object** for more complex data structures
  - **symbol** for unique identifiers

Numbers

- The **number** type serves both for integer and floating point numbers
- Numbers are stored in memory as double precision 64-bit floating point numbers
- Besides regular numbers, there are three special symbols which also belong to the number type: Infinity, -Infinity and NaN
- Infinity represents the mathematical Infinity ∞

```
alert(1 / 0);     // Infinity
alert(Infinity); // Infinity
```

- NaN (Not a Number) represents a computational error. It is a result of an incorrect or an undefined mathematical operation, for instance:

```
alert("hello" * 2); // NaN, such division is erroneous
```

- NaN is sticky. Any further operation on NaN would give NaN:

```
alert("hello" * 2 + 5); // NaN
```

Strings

- A string in JavaScript must be quoted
- There are 3 types of quotes:
  - Double quotes: "Hello“
  - Single quotes: 'Hello'
  - Backticks: `Hello`

```
let str = "Hello";
let str2 = 'Single quotes are ok too';
let phrase = `can embed ${str}`;
```

- There’s no difference between double and single quotes in JavaScript
- Backticks are “extended functionality” quotes. They allow us to embed variables and expressions into a string by wrapping them in ${…}, for example:

```
let name = "John";

// embed a variable
alert(`Hello, ${name}!`); // Hello, John!

// embed an expression
alert(`the result is ${1 + 2}`); // the result is 3
```

Exercise (2)

- What is the output of the following script?

```
let name = "Dan";

alert(`hello ${name + 1}`); // ?

alert(`hello ${"name"}`); // ?

alert("hello ${name}"); // ?

alert('hello ${"name"}'); // ?
```

Boolean

- The boolean type has only two values: true and false
- This type is commonly used to store yes/no values: true means “yes, correct”, and false means “no, incorrect”
- For example:

```
let nameFieldChecked = true; // yes, name field is checked
let ageFieldChecked = false; // no, age field is not checked
```

- Boolean values also come as a result of comparisons:

```
let isGreater = 4 > 1;
alert(isGreater); // true (the comparison result is "yes")
```

Null

- null forms a separate type of its own, which contains only the null value:

```
let age = null;
```

- null expresses a lack of identification, indicating that a variable points to no object
  - null is often found in a place where an object can be expected but no object is relevant
- The code above states that age is known to exist now but it has no type or value

Undefined

- The special value undefined also makes a type of its own, just like null
- The meaning of undefined is “value is not assigned”
- If a variable is declared, but not assigned, then its value is exactly undefined:

```
let x;
alert(x); // shows "undefined"
```

- Technically, it is possible to assign undefined to any variable:

```
let a = 123;
a = undefined;
alert(a); // "undefined"
```

- But it’s not recommended to do that
- Normally, we use null to write an “empty” or an “unknown” value into the variable, and undefined is only used for checks, to see if the variable is assigned

Objects and Symbols

- The object type is special
- All other types are called “primitive”, because their values can contain only a single thing (a string, a number, etc.)
- In contrast, objects are used to store collections of data and more complex entities
- We’ll learn about objects in JavaScript later in the course
- The symbol type is used to create unique identifiers for objects (new in ES6)
- We’ll study about the symbol type after objects

The typeof Operator

- The typeof operator returns the type of the argument
- It’s useful when we want to process values of different types differently, or just want to make a quick check
- It supports two forms of syntax:
  - As an operator: typeof x
  - Function style: typeof(x)
- The call to typeof x returns a string with the type name:

```
typeof 0 // "number"
typeof "foo" // "string"
typeof true // "boolean"
typeof null // "object" null is recognized erroneously by typeof as an object (historical reasons)
typeof undefined // "undefined"
typeof Math // "object"
typeof Symbol("id") // "symbol"
typeof alert // "function" functions belong to the object type, but typeof treats them differently
```

Type Conversions

- Most of the time, data types are converted automatically as needed during script execution
  - For example, alert automatically converts any value to a string to show it
  - Mathematical operations convert values to numbers
- There are also cases when we need to explicitly convert a value to put things right
- There are three most widely used type conversions: to string, to number and to boolean

Conversion To String

- String conversion happens when we need the string form of a value
- For example, alert(value) does it to show the value
- We can also call String(value) function for that:

```
let value = true;
alert(typeof value); // boolean

value = String(value); // now value is a string "true"
alert(typeof value); // string
```

Conversion To Number

- Numeric conversion happens in mathematical functions and expressions automatically
  - For example, when multiplication * is applied to non-numbers:

```
alert('3' * 2); // 6
```

- Addition (+) is exceptional: if one of the added values is a string, then the other one is also converted to a string and then it concatenates (joins) them:

```
alert(1 + '2'); // '12' (string to the right)
alert('1' + 2); // '12' (string to the left)
```

- Explicit conversion is usually required when we read a value from a string-based source like a prompt or a text field, but we expect a number to be entered
- We can use a Number(value) function to explicitly convert a value:

```
let str = '123';
let num = Number(str); // becomes a number 123
alert(typeof num); // number
```

Conversion to Number

- If the string is not a valid number, the result of such conversion is NaN, for instance:

```
let age = Number('hello');
alert(age); // NaN, conversion failed
```

- Numeric conversion rules:

Value	Becomes…
undefined	NaN
null	0
true and false	1 and 0
string	Whitespaces from the start and the end are removed. Then, if the remaining string is empty, the result is 0. Otherwise, the number is “read” from the string. An error gives NaN.

Conversion To Boolean

- Boolean conversion is the simplest one
- It happens in logical operations, but also can be performed manually with the call of Boolean(value)
- The conversion rule:
  - "Empty" values, like 0, an empty string, null, undefined and NaN become false
  - Other values become true

```
alert(Boolean(1)); // true
alert(Boolean(0)); // false

alert(Boolean("hello")); // true
alert(Boolean("")); // false
```

Exercise (3)

- What are results of these expressions?

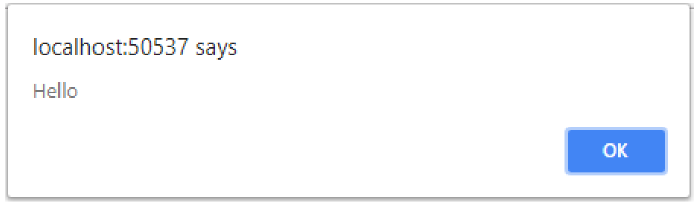
```
"" + 1 + 0
"" - 1 + 0
true + false
6 / "3"
"2" * "3"
4 + 5 + "px"
"$" + 4 + 5
'4' - 2
"4px" - 2
7 / 0
' -9 ' + 5
" -9 " - 5
null + 1
undefined + 1
```

- Think well, write down and then check your answer in the browser

Interaction: alert, prompt, confirm

- The browser supplies a few user-interface functions: alert, prompt and confirm
- alert(message) - shows a message and pauses the script execution until the user presses “OK”
- The mini-window with the message is called a *modal window*
- The word “modal” means that the visitor can’t interact with the rest of the page, press other buttons etc, until they have dealt with the window

```
alert('Hello');
```



localhost:50537 says

Hello

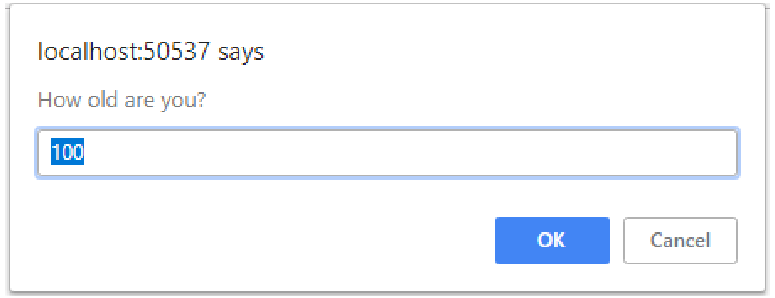
Interaction: alert, prompt, confirm

- prompt shows a modal window with a text message, an input field for the visitor and buttons OK/CANCEL
- It accepts two arguments:

```
result = prompt(title[, default]);
```

  - title - the text to show to the visitor
  - default - an optional second parameter, the initial value for the input field
- The call to prompt returns the text from the field or null if the input was canceled

```
let age = prompt('How old are you?', 100);
alert(`You are ${age} years old!`); // You are 100 years old!
```



localhost:50537 says

How old are you?

100

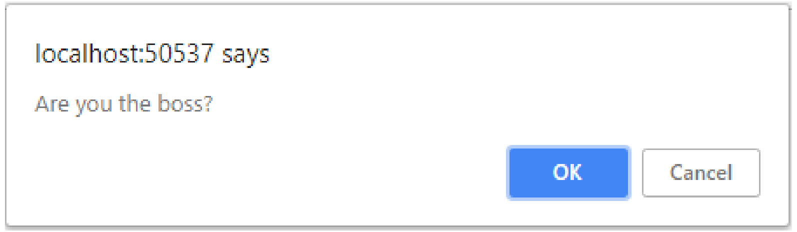
Interaction: alert, prompt, confirm

- confirm shows a modal window with a question and two buttons: OK and CANCEL

```
result = confirm(question);
```

- The result is true if OK is pressed and false otherwise

```
let isBoss = confirm("Are you the boss?");
alert(isBoss); // true if OK is pressed
```



localhost:50537 says

Are you the boss?

Operators

- An **operand** – is what operators are applied to
- For example, in multiplication 5 * 2 there are two operands: the left operand is 5, and the right operand is 2.
- An operator is unary if it has a single operand
  - For example, the unary negation - reverses the sign of the number:

```
let x = 1;

x = -x;
alert(x); // -1, unary negation was applied
```

- An operator is binary if it has two operands.
  - The same minus exists in the binary form as well:

```
let x = 1, y = 3;
alert(y - x); // 2, binary minus subtracts values
```

String Concatenation

- Usually the plus operator + sums numbers
- But if the binary + is applied to strings, it concatenates them:

```
let s = "my" + "string";
alert(s); // mystring
```

- If any of the operands is a string, then the other one is converted to a string too:

```
alert('1' + 2); // "12"
alert(2 + '1'); // "21"
```

- However, operations run from left to right. If there are two numbers followed by a string, the numbers will be added before being converted to a string:

```
alert(2 + 2 + '1'); // "41" and not "221"
```

Integer Division and Remainder %

- The division operator a / b produces the exact quotient of its operands
- Integer division can be achieved by applying **parseInt()** on the quotient

```
alert(5 / 2) // 2.5
alert(parseInt(5 / 2)) // 2
```

- The result of a % b is the remainder of the integer division of a by b
- For instance:

```
alert(5 % 2); // 1 is a remainder of 5 divided by 2
alert(8 % 3); // 2 is a remainder of 8 divided by 3
alert(6 % 3); // 0 is a remainder of 6 divided by 3
```

Exponentiation **

- The exponentiation operator ** is a recent addition to the language (ES6)
- For a natural number b, the result of a ** b is a multiplied by itself b times

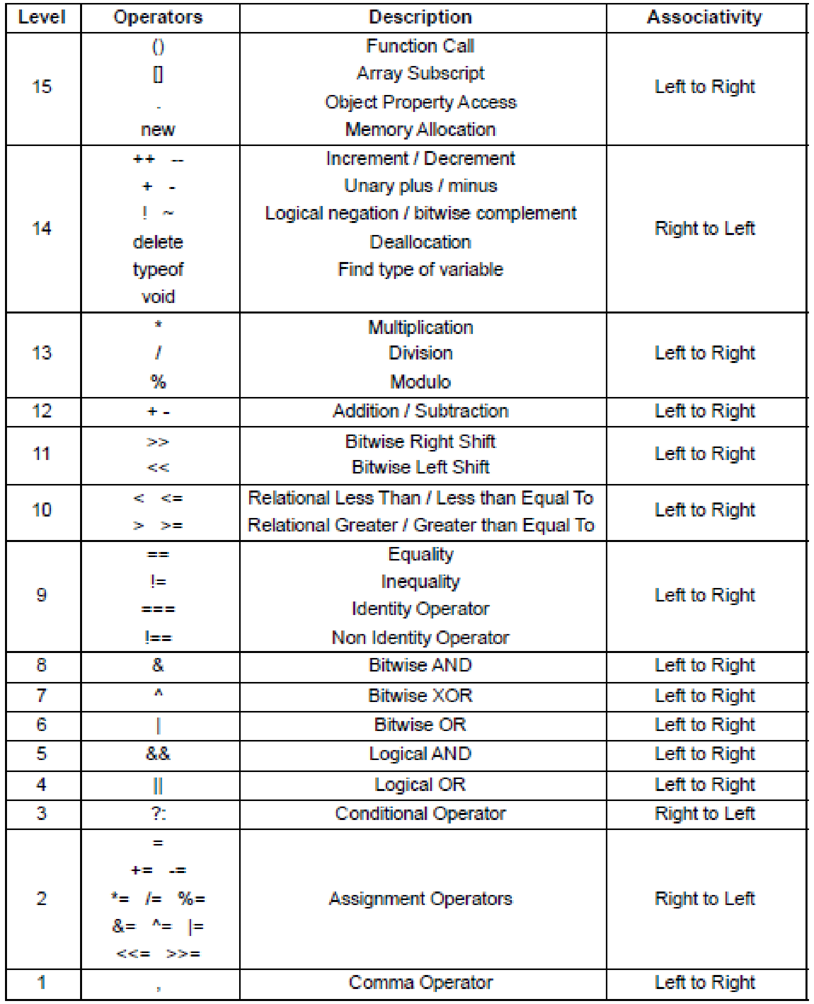
```
alert(2 ** 2); // 4  (2 * 2)
alert(2 ** 3); // 8  (2 * 2 * 2)
alert(2 ** 4); // 16 (2 * 2 * 2 * 2)
```

- The operator works for non-integer numbers of a and b as well, for instance:

```
alert(4 ** (1 / 2)); // 2 (power of 1/2 is the same as a square root)
alert(8 ** (1 / 3)); // 2 (power of 1/3 is the same as a cubic root)
```

Operators Precedence

- Operator precedence determines the way in which operators are parsed with respect to each other
- Operators with higher precedence become the operands of operators with lower precedence
- Parentheses override any precedence



Level	Operators	Description	Associativity
15	() [] . new	Function Call Array Subscript Object Property Access Memory Allocation	Left to Right
14	++ -- + - ! ~ delete typeof void	Increment / Decrement Unary plus / minus Logical negation / bitwise complement Deallocation Find type of variable	Right to Left
13	* / %	Multiplication Division Modulo	Left to Right
12	+ -	Addition / Subtraction	Left to Right
11	>> <<	Bitwise Right Shift Bitwise Left Shift	Left to Right
10	< <= > >=	Relational Less Than / Less than Equal To Relational Greater / Greater than Equal To	Left to Right
9	== != === !==	Equality Inequality Identity Operator Non Identity Operator	Left to Right
8	&	Bitwise AND	Left to Right
7	^	Bitwise XOR	Left to Right
6	|	Bitwise OR	Left to Right
5	&&	Logical AND	Left to Right
4	||	Logical OR	Left to Right
3	?:	Conditional Operator	Right to Left
2	= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operators	Right to Left
1	,	Comma Operator	Left to Right

Assignment =

- An assignment = is also an operator
- It is listed in the precedence table with the very low priority of 3
- That’s why when we assign a variable, like x = 2 * 2 + 1, then the calculations are done first, and afterwards the = is evaluated, storing the result in x
- Every operator returns a value, including the assignment operator
- The call x = value writes the value into x *and then returns it*

```
let a = 1;
let b = 2;
let c = 3 - (a = b + 1);

alert(a); // 3
alert(c); // 0
```

  - The result of (a = b + 1) is the value which is assigned to a (that is 3). It is then used to subtract from 3.

Assignment =

- It is possible to chain assignments:

```
let a, b, c;
a = b = c = 2 + 2;

alert(a); // 4
alert(b); // 4
alert(c); // 4
```

- Chained assignments evaluate from right to left
- First the rightmost expression 2 + 2 is evaluated then assigned to the variables on the left: c, b and a
- At the end, all variables share a single value

Increment/Decrement

- Increasing or decreasing a number by one is among the most common numerical operations
- So, there are special operators for that:
  - **Increment** ++ increases a variable by 1:

```
let counter = 2;
counter++;      // works the same as counter = counter + 1, but is shorter
alert(counter); // 3
```

  - **Decrement** -- decreases a variable by 1:

```
let counter = 2;
counter--;      // works the same as counter = counter - 1, but is shorter
alert(counter); // 1
```

- Increment/decrement can be applied only to a variable
  - An attempt to use it on a value like 5++ will give an error

Increment/Decrement

- Operators ++ and -- can be placed both after and before the variable
  - When the operator goes after the variable, it is called a “postfix form”: counter++.
  - When it goes before the variable, it is called a “prefix form”: ++counter
- Both of these records do the same: increase counter by 1
- Is there any difference? Yes, but we can only see it if we use the returned value of ++/--
- The prefix form returns the new value, while the postfix form returns the old value (prior to increment/decrement)
- To see the difference, here’s the example:

```
let counter = 1;
let a = ++counter; // prefix increment

alert(a); // 2
```

```
let counter = 1;
let a = counter++; // postfix increment

alert(a); // 1
```

Modify-in-place

- We often need to apply an operator to a variable and store the new result in it
- For example:

```
let n = 2;
n = n + 5;
n = n * 2;
```

- This notation can be shortened using operators += and *=:

```
let n = 2;
n += 5; // now n = 7 (same as n = n + 5)
n *= 2; // now n = 14 (same as n = n * 2)
alert(n); // 14
```

- Short “modify-and-assign” operators exist for all arithmetical and bitwise operators: /=, -= etc.
- Such operators have the same precedence as a normal assignment, so they run after most other calculations

Exercise (4)

- Create a web page that asks the user to enter two numbers and displays their sum and their product

Exercise (5)

- What will be the output of the code below?

```
let a = 1, b = 1;

let c = ++a;
let d = c++;

alert(a); // ?
alert(c); // ?
alert(d++); // ?
```

Comparisons

- Many comparison operators we know from maths:
  - Greater/less than: a > b, a < b.
  - Greater/less than or equals: a >= b, a <= b.
  - Equality check is written as a == b
    - Please note the double equation sign ==
    - A single symbol a = b would mean an assignment
- Not equals: In maths the notation is ≠, in JavaScript it's written as an assignment with an exclamation sign before it: a != b
- Just as all other operators, a comparison returns a value
- The value is of the boolean type:
  - true – means "yes" or "correct"
  - false – means "no" or "wrong"

Comparisons

- For example:

```
alert(2 > 1);  // true (correct)
alert(2 == 1); // false (wrong)
alert(2 != 1); // true (correct)
```

- A comparison result can be assigned to a variable, just like any value:

```
let result = 5 > 4; // assign the result of the comparison
alert(result); // true
```

String Comparison

- To see which string is greater than the other, the so-called “dictionary” or “lexicographical” order is used
- In other words, strings are compared letter-by-letter.
- For example:

```
alert('Z' > 'A'); // true
alert('Glow' > 'Glee'); // true
alert('Bee' > 'Be'); // true
```

- Note that case matters. A capital letter "A" is not equal to the lowercase "a".
- Which one is greater? Actually, the lowercase "a" is. Why? Because the lowercase character has a greater index in the internal encoding table (Unicode)
  - You can find the table here: https://www.rapidtables.com/code/text/unicode-characters.html

Comparison of Different Types

- When comparing values that belong to different types, they are converted to numbers:

```
alert('2' > 1); // true, string '2' becomes a number 2
alert('01' == 1); // true, string '01' becomes a number 1
```

- For boolean values, true becomes 1 and false becomes 0:

```
alert(true == 1); // true
alert(false == 0); // true
```

- An empty string converts to 0
- A non-numeric string converts to NaN which is always false

Strict Equality

- A regular equality check == has a problem: it cannot differ 0 or empty string from false

```
alert(false == 0); // true
alert('' == false); // true
```

  - That’s because operands of different types are converted to a number by the equality operator
  - An empty string, just like false, becomes a zero.
- What to do if we’d like to differentiate 0 from false?
- **A strict equality operator ===** checks the equality without type conversion
  - If a and b are of different types, then a === b immediately returns false without an attempt to convert them

```
alert(0 === false); // false, because the types are different
```

- There also exists a “strict non-equality” operator !==, as an analogy for !=

Comparison with null and undefined

- There’s a non-intuitive behavior when null or undefined are compared with other values
- For a strict equality check === these values are different
  - because each of them belongs to a separate type of its own

```
alert(null === undefined); // false
```

- For a non-strict check == there’s a special rule:
  - These two are a “sweet couple”: they equal each other (in the sense of ==), but not any other value

```
alert(null == undefined); // true
```

- For maths and other comparisons < > <= >= values null/undefined are converted to a number: null becomes 0, while undefined becomes NaN
  - NaN is a special numeric value which returns false for all comparisons

Exercise (6)

- What will be the output of the following script:

```
alert(5 > 4); // ?
alert("apple" > "pineapple"); // ?
alert("2" > "12"); // ?
alert(undefined == null); // ?
alert(undefined === null); // ?
alert(null == " 0 "); // ?
alert(null === +"0"); // ?
```

- Write down and then check your answer in the browser

Conditions

- Sometimes we need to perform different actions based on a condition
- The **if** statement gets a condition, evaluates it and, if the result is true, executes the code

```
let num = prompt('Please enter a number');
if (num % 2 == 0) alert('The number is even');
```

- If there is more than one statement to be executed if the condition holds, we have to wrap our code block inside curly braces:

```
let num = prompt('Please enter a number');
if (num % 2 == 0) {
    alert('The number is even');
    alert('Have fun');
}
```

- It is recommended to wrap your code block with curly braces {} every time with if, even if there is only one statement. That improves readability.

Boolean Conversion

- The if (…) statement evaluates the expression in parentheses and converts it to the boolean type
  - A number 0, an empty string "", null, undefined and NaN become false
  - Other values become true,
- So, the code under this condition would never execute:

```
if (0) { // 0 is falsy
    ...
}
```

- And inside this condition – always works:

```
if (x = 5) { // the expression x = 5 has the value of 5 which is truthy
    ...
}
```

  - Always use == inside conditions!

The “else” Clause

- The if statement may contain an optional “else” block
- It executes when the condition is wrong
- For example:

```
let num = prompt('Please enter a number');
if (num % 2 == 0) {
    alert('The number is even');
}
else {
    alert('The number is odd');
}
```

Several Conditions: else if

- Sometimes we’d like to test several variants of a condition. There is an else if clause for that.
- For example:

```
let num = prompt('Please enter a number');
if (num > 0) {
    alert('The number is positive');
}
else if (num < 0) {
    alert('The number is negative');
} else {
    alert('The number is zero');
}
```

Ternary Operator '?'

- Sometimes we need to assign a variable depending on a condition, e.g.,

```
let accessAllowed;
let age = prompt('How old are you? ');
if (age > 18) {
    accessAllowed = true;
} else {
    accessAllowed = false;
}
alert(accessAllowed);
```

- The "ternary" or "question mark" operator lets us do that shorter and simpler

```
let result = condition ? value1 : value2
```

  - The condition is evaluated, if it's truthy then value1 is returned, otherwise – value2.

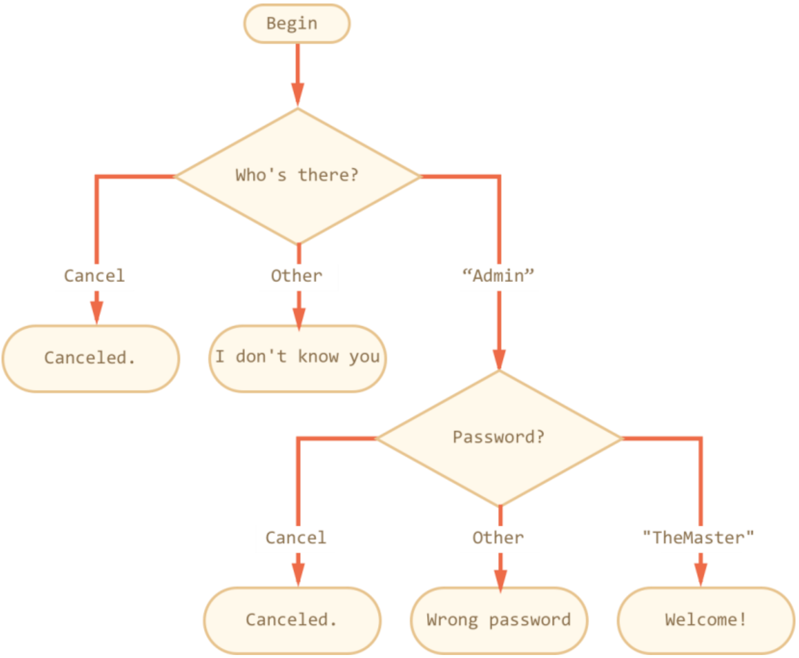
- For example:

```
let accessAllowed = age > 18 ? true : false;
```

  - In this example, we could also have written `let accessAllowed = age > 18;`

Exercise (7)

- Write the code which asks for a login with prompt
  - If the visitor enters “Admin“, then prompt for a password
  - If the input is an empty line or Esc – show “Canceled.”
  - If it’s another string – then show “I don’t know you”
- The password is checked as follows:
  - If it equals “TheMaster”, then show “Welcome!”
  - Another string – show “Wrong password”
  - For an empty string or cancelled input, show “Canceled”



- Hint: passing an empty input to a prompt returns an empty string '', while pressing ESC during a prompt returns null

Logical Operators

- There are three logical operators in JavaScript: || (OR), && (AND), ! (NOT).
  - Although they are called “logical”, they can be applied to values of any type, not only boolean

- The “OR” operator is represented with two vertical line symbols
- If any of its arguments are true, then it returns true, otherwise it returns false

```
let hour = 9;

if (hour < 10 || hour > 18) {
    alert('The office is closed.');
}
```

- If an operand is not boolean, then it’s converted to boolean for the evaluation:

```
if (1 || 0) { // works just like if(true || false)
    alert('truthy!');
}
```

Short-Circuit Evaluation

- OR evaluates and tests its operands from left to right
- It returns the first truthy value or the last value if none were found

```
alert(1 || 0); // 1 (1 is truthy)
alert(true || 'no matter what'); // (true is truthy)
alert(null || 1); // 1 (1 is the first truthy value)
alert(null || 0 || 1); // 1 (the first truthy value)
alert(undefined || null || 0); // 0 (all falsy, returns the last value)
```

- The evaluation of operands stops when a truthy value is reached:

```
let x;
true || (x = 1);
alert(x); // undefined, because (x = 1) not evaluated
```

- This process is called “a short-circuit evaluation”, because it goes as short as possible from left to right

AND Operator

- The AND operator is represented with two ampersands &&
- AND returns true if both operands are truthy and false otherwise:

```
let hour = 12;
let minute = 30;

if (hour == 12 && minute == 30) {
    alert('Time is 12:30');
}
```

- Just as for OR, any value is allowed as an operand of AND:

```
if (1 && 0) { // evaluated as true && false
    alert("won't work, because the result is falsy");
}
```

AND Operator

- AND returns the first falsy value or the last value if none were found

```
// if the first operand is truthy, AND returns the second operand:
alert(1 && 0); // 0
alert(1 && 5); // 5

// if the first operand is falsy, AND returns it.
// The second operand is ignored
alert(null && 5); // null
alert(0 && "no matter what"); // 0
```

- AND && executes before OR ||

```
alert(5 || 1 && 0); // 5
```

NOT Operator

- The boolean NOT operator is represented with an exclamation sign !
- The operator accepts a single argument and does the following:
  - Converts the operand to boolean type: true/false
  - Returns an inverse value

```
alert(!true); // false
alert(!0); // true
```

- NOT has higher precedence than the AND and OR operators

```
alert(!5 || 1); // 1
alert(!(5 || 1)); // false
```

Exercise (8)

- What the code below is going to output?

```
alert(null || 2 || undefined); // ?
alert(1 && null && 2); // ?
alert(null || 2 && 3 || 4); // ?
alert(!1 && !2 || 3); // ?
```

Exercise (9)

- Create a web page that asks the user to enter a year, and prints whether this year is a leap year
- Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400
- For example, the years 1700, 1800, and 1900 were not leap years, but the years 1600 and 2000 were

The switch statement

- A switch statement can replace multiple if checks
- It gives a more descriptive way to compare a value with multiple variants
- The switch has one or more case blocks and an optional default
- It looks like this:

```
switch(x) {
    case 'value1':  // if (x === 'value1')
    ...
    [break]

    case 'value2':  // if (x === 'value2')
    ...
    [break]

    default:
    ...
    [break]
}
```

- The value of x is checked for a strict equality to the value from the first case (value1) then to the second (value2) and so on
- If the equality is found, switch starts to execute the code starting from the corresponding case, until the nearest break (or until the end of switch)
- If no case is matched then the default code is executed (if it exists)

The switch statement - example

```
let a = 2 + 2;

switch (a) {
    case 3:
        alert('Too small');
        break;
    case 4:
        alert('Exactly!');
        break;
    case 5:
        alert('Too large');
        break;
    default:
        alert("I don't know such values");
}
```

- Here the switch starts to compare *a* from the first case variant that is 3
- The match fails
- Then 4. That’s a match, so the execution starts from case 4 until the nearest break.

The switch statement

- If there is no break then the execution continues with the next case without any checks

```
let a = 2 + 2;

switch (a) {
    case 3:
        alert('Too small');
    case 4:
        alert('Exactly!');
    case 5:
        alert('Too big');
    default:
        alert("I don't know such values");
}
```

- In the example above we'll see sequential execution of three alerts:
  - alert('Exactly!');
  - alert('Too big');
  - alert("I don't know such values");

The switch statement

- Any expression can be a switch/case argument
- For example:

```
let a = "1";
let b = 0;

switch (+a) {
    case b + 1:
        alert("this runs, because +a is 1, exactly equals b+1");
        break;

    default:
        alert("this doesn't run");
}
```

Grouping of Cases

- Several variants of case which share the same code can be grouped.
- For example, if we want the same code to run for case 3 and case 5:

```
let a = 2 + 2;

switch (a) {
    case 4:
        alert('Right!');
        break;

    case 3:                   // (*) grouped two cases
    case 5:
        alert('Wrong!');
        alert("Why don't you take a math class?");
        break;

    default:
        alert('The result is strange. Really.');
}
```

Exercise (10)

- Write the code using if..else which would correspond to the following switch:

```
switch (browser) {
    case 'Edge':
        alert("You've got the Edge!");
        break;

    case 'Chrome':
    case 'Firefox':
    case 'Safari':
    case 'Opera':
        alert('Okay we support these browsers too');
        break;

    default:
        alert('We hope that this page looks ok!');
}
```

Exercise (11)

- Write a simple calculator app
- Ask the user for two numbers and an operation (+, -, *, /)
- Display the result of applying the operation on the input numbers

Loops

- We often have a need to perform similar actions many times in a row
- For example, when we need to output goods from a list one after another. Or just run the same code for each number from 1 to 10.
- *Loops* are a way to repeat the same part of code multiple times
- There are three loop types in JavaScript:
    - While loops
    - Do-while loops
    - For loops

While Loop

- The while loop has the following syntax:

```
while (condition) {
    // the loop body
}
```

  - While the condition is true, the code from the loop body is executed

- For instance, the loop below outputs i while i < 5:

```
let i = 0;
while (i < 5) { // shows 0, 1, ..., 4
    alert(i);
    i++;
}
```

- A single execution of the loop body is called *an iteration*
  - The loop in the example above makes 5 iterations

While Loop

- Any expression or a variable can be a loop condition, not just a comparison. They are evaluated and converted to a boolean by while.
- For instance, the shorter way to write while (i != 0) could be while (i)

```
let i = 3;
while (i) { // when i becomes 0, the condition becomes falsy, and the loop stops
    alert(i);
    i--;
}
```

- If the loop body has a single statement, we can omit the brackets {…}:

```
let i = 3;
while (i) alert(i--);
```

The “do…while” loop

- The condition check can be moved *below* the loop body using the do..while syntax:

```
do {
    // loop body
}
while (condition);
```

- The loop will first execute the body, then check the condition and, while it’s truthy, execute it again and again
- For example:

```
let i = 0;
do {
    alert(i);
    i++;
} while (i < 3)
```

- This form of syntax is rarely used except when you want the body of the loop to execute **at least once** regardless of the condition being truthy

Exercise (12)

- Ask the user to enter a number
- If the user provides a non-numeric value (such as “abc”), display an error message and ask the user to try again
- Hint: use the function isNaN() to check if the conversion to number failed

Exercise (13)

- Get a number from the user and print the sum of its digits
- For example, if the user enters the number 57103, then your script should print 16 (5+7+1+0+3)

For Loop

- The for loop has the following syntax:

```
for (begin; condition; step) {
    // ... loop body ...
}
```

  - **begin** is executed once before entering the loop
  - **condition** is checked before every loop iteration, if fails the loop stops
  - **step** is executed after the body on each iteration, but before the condition check

- For instance, the following loop runs alert(i) for i from 0 up to (but not including) 3:

```
for (let i = 0; i < 3; i++) {
    alert(i); // 0, 1, 2
}
```

```
// run begin
let i = 0
// if condition → run body and run step
if (i < 3) { alert(i); i++ }
// if condition → run body and run step
if (i < 3) { alert(i); i++ }
// if condition → run body and run step
if (i < 3) { alert(i); i++ }
// ...finish, because now i == 3
```

For Loop - Inline Variable Declaration

- Here the “counter” variable i is declared right in the loop
- That’s called an “inline” variable declaration
- Such variables are visible only inside the loop

```
for (let i = 0; i < 3; i++) {
    alert(i); // 0, 1, 2
}
alert(i); // error, no such variable
```

- Instead of defining a variable, we can use an existing one:

```
let i = 0;

for (i = 0; i < 3; i++) { // use an existing variable
    alert(i); // 0, 1, 2
}

alert(i); // 3, visible, because declared outside of the loop
```

For Loop - Skipping Parts

- Any part of **for** can be skipped
- For example, we can omit begin if we don’t need to do anything at the loop start

```
let i = 0; // we have i already declared and assigned

for (; i < 3; i++) { // no need for "begin"
    alert(i); // 0, 1, 2
}
```

- We can also remove the step part:

```
let i = 0;

for (; i < 3;) {
    alert(i++);
}
```

  - The loop became identical to while (i < 3)

For Loop - Skipping Parts

- We can actually remove everything, thus creating an infinite loop:

```
for (; ;) {
    // repeats without limits
}
```

- Note that the two for semicolons ; must be present, otherwise it would be a syntax error

Breaking the Loop

- Normally the loop exits when the condition becomes falsy
- But we can force the exit at any moment using the break directive
- For example, the loop below asks the user for a series of numbers, but “breaks” when no number is entered:

```
let sum = 0;
while (true) {
    let num = Number(prompt("Enter a number", ''));
    if (!num) break; // (*)
    sum += value;
}
alert('Sum: ' + sum);
```

  - The break directive is activated at the line (*) if the user enters an empty line or cancels the input
  - It stops the loop immediately, passing the control to the first line after the loop. Namely, alert.
- The combination “infinite loop + break as needed” is great for situations when the condition must be checked not in the beginning/end of the loop, but in the middle

Continue to the Next Iteration

- The continue directive doesn't stop the whole loop. Instead it stops the current iteration and forces the loop to start a new one (if the condition allows).
- We can use it if we're done on the current iteration and would like to move on to the next
- The loop below uses continue to output only odd values:

```
for (let i = 0; i < 10; i++) {

    // if true, skip the remaining part of the body
    if (i % 2 == 0) continue;

    alert(i); // 1, then 3, 5, 7, 9
}
```

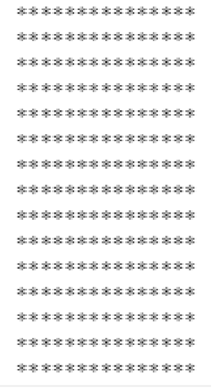
- The directive continue helps to decrease nesting level

Exercise (14)

- Get from the user two numbers: min and max
- Output all the even numbers between min and max (note that min and max themselves might be odd numbers)
- For example, if the user enters min = 5 and max = 14, you should print the numbers 6,8,10,12,14

Exercise (15)

- Get from the user a number
- Print to the console a square of stars whose length is the number specified by the user
- For example, if the user entered the number 15, your should print:



```
***************
***************
***************
***************
***************
***************
***************
***************
***************
***************
***************
***************
***************
***************
***************
```

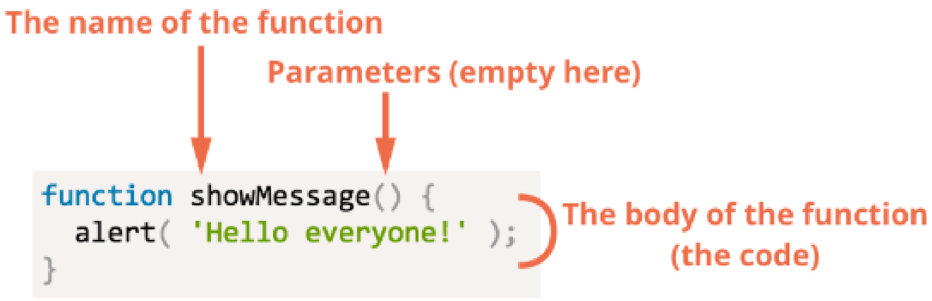
- Hint: Use the character ‘\n’ to start a new line

Functions

- Quite often we need to perform a similar action in many places of the script
  - For example, we need to show a nice-looking message when a visitor logs in, logs out and maybe somewhere else
- Functions are the main “building blocks” of the program
- They allow the code to be called many times without repetition
- We’ve already seen examples of built-in functions, like alert(message) and prompt(message, default), but we can create functions of our own as well

Function Declaration

- To create a function we can use a *function declaration*:



The name of the function

Parameters (empty here)

```
function showMessage() {
  alert( 'Hello everyone!' );
}
```

The body of the function (the code)

- Our new function can be called by its name: showMessage()

```
function showMessage() {
    alert('Hello everyone!');
}
showMessage();
showMessage();
```

  - The call showMessage() executes the code of the function
  - This example clearly demonstrates one of the main purposes of functions: avoid code duplication

Local Variables

- A variable declared inside a function is only visible inside that function.
- For example:

```
function showMessage() {
    let message = 'Hello, I'm JavaScript!'; // local variable
    alert(message);
}
showMessage(); // Hello, I'm JavaScript!
alert(message); // <-- Error! The variable is local to the function
```

Global Variables

- Variables declared outside of any function, are called *global*
- Global variables are visible from any function

```
let userName = 'John';
function showMessage() {
    let message = 'Hello, ' + userName;
    alert(message);
}
showMessage(); // Hello, John
```

- If a same-named variable is declared inside the function, it *shadows* the outer one:

```
let userName = 'John';
function showMessage() {
    let userName = 'Bob'; // declare a local variable
    let message = 'Hello,' + userName; // Bob
    alert(message);
}
// the function will create and use its own userName
showMessage();
alert(userName); // John, unchanged, the function did not access the outer variable
```

Global Variables

- Usually, a function declares all variables specific to its task
- Global variables only store project-level data, so when it’s important that these variables are accessible from anywhere
- Modern code has few or no globals
- Most variables reside in their functions

Parameters

- We can pass arbitrary data to functions using parameters (also called *function arguments*)
- In the example below, the function has two parameters: from and text

```javascript
function showMessage(from, text) { // arguments: from, text
    alert(from + ': ' + text);
}

showMessage('Ann', 'Hello!'); // Ann: Hello!
showMessage('Ann', "What's up?"); // Ann: What's up?
```

- When the function is called, the given values are copied to local variables from and text, i.e. the arguments are passed **by-value**

Pass By Value

- If a function changes one of its parameters, the change is not seen outside, because a function always gets a copy of the value:

```
function showMessage(from, text) {
    from = '*' + from + '*'; // make "from" look nicer
    alert(from + ': ' + text);
}

let from = 'Ann';
showMessage(from, 'Hello'); // *Ann*: Hello

// the value of "from" is the same, the function modified a local copy
alert(from); // Ann
```

Default Values

- If a parameter is not provided, then its value becomes undefined
- For instance, the function showMessage(from, text) can be called with a single argument:

```
showMessage('Ann');
```

- That’s not an error. Such a call would output "Ann: undefined"
- There’s no text, so it’s assumed that text === undefined
- If we want to use a “default” text in this case, then we can specify it after =:

```
function showMessage(from, text = 'no text given') {
    alert(from + ": " + text);
}

showMessage('Ann'); // Ann: no text given
```

  - Now if the text parameter is not passed, it will get the value "no text given"

Default Parameters Old-Style

- Old editions of JavaScript (before ES6) did not support default parameters
- There are alternative ways to support them, that you can find mostly in older scripts
- For instance, an explicit check for being undefined:

```
function showMessage(from, text) {
    if (text === undefined) {
        text = 'no text given';
    }

    alert(from + ": " + text);
}
```

- Or the || operator:

```
function showMessage(from, text) {
    // if text is falsy then text gets the "default" value
    text = text || 'no text given';
    ...
}
```

Returning a Value

- A function can return a value back into the calling code as the result
- The simplest example would be a function that sums two values:

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

let result = sum(1, 2);
alert(result); // 3
```

- The directive **return** can be in any place of the function
- When the execution reaches it, the function stops, and the value is returned to the calling code

Returning a Value

- It is possible to use return without a value - that causes the function to exit immediately. For example:

```
function showMovie(age) {
    if (age < 18)
        return;

    alert('Showing you the movie');
}
```

- If a function does not return a value, it is the same as if it returns undefined:

```
function doNothing() { /* empty */ }

alert(doNothing() === undefined); // true
```

Naming Functions

- A function name should clearly describe what the function does
- When we see a function call in the code, a good name instantly gives us an understanding what it does and returns
- A function is an action, thus it is a widespread practice to start a function with a verbal prefix which vaguely describes the action
- For instance, functions starting with…
  - "show…" – usually show something.
  - "get…" – return a value
  - "calc…" – calculate something
  - "create…" – create something
  - "check…" – check something and return a boolean

One Function – One Action

- Functions should be short and do exactly one thing
- Two independent actions usually deserve two functions, even if they are usually called together (in that case we can make a 3rd function that calls those two)
- A separate function is not only easier to test and debug – its very existence is a great comment!
- A few examples of breaking this rule:
  - getAge – would be bad if it shows an alert with the age (should only get)
  - createForm – would be bad if it modifies the document, adding a form to it (should only create it and return)
  - checkPermission – would be bad if displays the access granted/denied message (should only perform the check and return the result)

Exercise (16)

- Write a function pow(x,n) that returns x in power n, or in other words, multiplies x by itself n times and returns the result
  - e.g., pow(3, 4) = 3 * 3 * 3 * 3 = 81
- The function should support only natural values of n (i.e., integer from 1 up)
- Create a web page that prompts for x and n, and then shows the result of pow(x,n)

Exercise (17)

- Write a function **isPrime**(n) that gets a natural value of n and returns a boolean indicating is n is a prime number or not
- A prime number is a natural number that divides only by 1 and itself
  - e.g., 7, 11 and 13 are prime numbers while 8, 12 and 15 are not primes
- Write another function **showPrimes**(n) that outputs all the prime numbers up to n
  - This function should use isPrime(n) to test for primality
- Create a web page that prompts for n, and then shows all the prime numbers up to n

Function Expressions

- The **function** keyword can be used to define a function inside an expression

```
let getRectArea = function (width, height) {
    return width * height;
}

console.log(getRectArea(3, 4)); // 12
```

- The function name can be omitted in function expression, in which case the function is **anonymous**
- Function expressions in JavaScript are not hoisted, unlike function declarations, i.e., you can't use function expressions before you define them:

```
notHoisted(); // ReferenceError: notHoisted is not a function

let notHoisted = function () {
    console.log('test');
};
```

Functions as Values

- In JavaScript, a function is a value, so we can work with it like with other kinds of values
- For example, we can copy a function to another variable:

```
function sayHi() {   // (1) create
    alert('Hello');
}

let func = sayHi;    // (2) copy

func(); // Hello     // (3) run the copy (it works)!
sayHi(); // Hello    // this still works too (why wouldn't it)
```

Callback Functions

- You can also pass functions as arguments to other functions
- For example, we will write a function `ask(question, yes, no)` with 3 parameters:
  - question – text of the question
  - yes - Function to run if the answer is “Yes”
  - no - Function to run if the answer is “No”
- The function asks the question and depending on the user’s answer calls yes() or no():

```javascript
function ask(question, yes, no) {
    if (confirm(question)) yes()
    else no();
}
function showOk() {
    alert('You agreed.');
}
function showCancel() {
    alert('You canceled the execution.');
}
// usage: functions showOk, showCancel are passed as arguments to ask
ask('Do you agree?', showOk, showCancel);
```

Callback Functions

- The arguments of ask are called ***callback functions*** or just ***callbacks***
- The idea is that we pass a function and expect it to be “called back” later if necessary
  - In our case, showOk becomes the callback for the “yes” answer, and showCancel for the “no”
- We can use Function Expressions to write the same function much shorter:

```
function ask(question, yes, no) {
    if (confirm(question)) yes()
    else no();
}

ask(
    'Do you agree?',
    function() { alert('You agreed.'); },
    function() { alert('You canceled the execution.'); }
);
```

  - Here, functions are declared right inside the ask(...) call. They have no name, and so are called *anonymous*. Such functions are not accessible outside of ask, but that’s just what we want here.

Arrow Functions

- There’s one more very simple and concise syntax for creating functions, that’s often better than Function Expressions
- It’s called “arrow functions”, because it looks like this:

```
let func = (arg1, arg2, ...argN) => expression
```

  - This creates a function func that has arguments arg1...argN, evaluates the expression on the right side with their use and returns its result

- It’s roughly the same as:

```
let func = function(arg1, arg2, ...argN) {
    return expression;
}
```

Arrow Functions

- Example:

```
let sum = (a, b) => a + b;
/* The arrow function is a shorter form of:

let sum = function(a, b) {
    return a + b;
};
*/

alert(sum(1, 2)); // 3
```

- If we have only one argument, then parentheses can be omitted:

```
let double = n => n * 2;
    // same as
// let double = function(n) { return n * 2 }

alert(double(3)); // 6
```

Arrow Functions

- If there are no arguments, parentheses should be empty:

```
let sayHi = () => alert("Hello!");

sayHi();
```

- Arrow functions can also be used as callback functions:

```
function ask(question, yes, no) {
    if (confirm(question)) yes()
    else no();
}

ask(
    'Do you agree?',
    () => alert('You agreed.'),
    () => alert('You canceled the execution.')
);
```

Multiline Arrow Functions

- Sometimes arrow functions need to be a little bit more complex, like having multiple expressions or statements
- It is also possible, but we should enclose them in curly braces, and then use a normal return within them

```
let sum = (a, b) => {  // the curly brace opens a multiline function
    let result = a + b;
    return result; // if we use curly braces, use return to get results
};

alert(sum(1, 2)); // 3
```

Summary

- Functions are values. They can be assigned, copied or declared in any place of the code.
- If the function is declared as a separate statement in the main code flow, that's called a "Function Declaration".
- If the function is created as a part of an expression, it's called a "Function Expression".
- Function Declarations are processed before the code block is executed. They are visible everywhere in the block.
- Function Expressions are created when the execution flow reaches them.
- We should use a Function Expression only when a Function Declaration is not fit for the task.
- Arrow functions are handy for one-liners. They come in two flavors:
  - Without curly braces: (...args) => expression – the right side is an expression: the function evaluates it and returns the result.
  - With curly braces: (...args) => { body } – brackets allow us to write multiple statements inside the function, but we need an explicit return to return something.

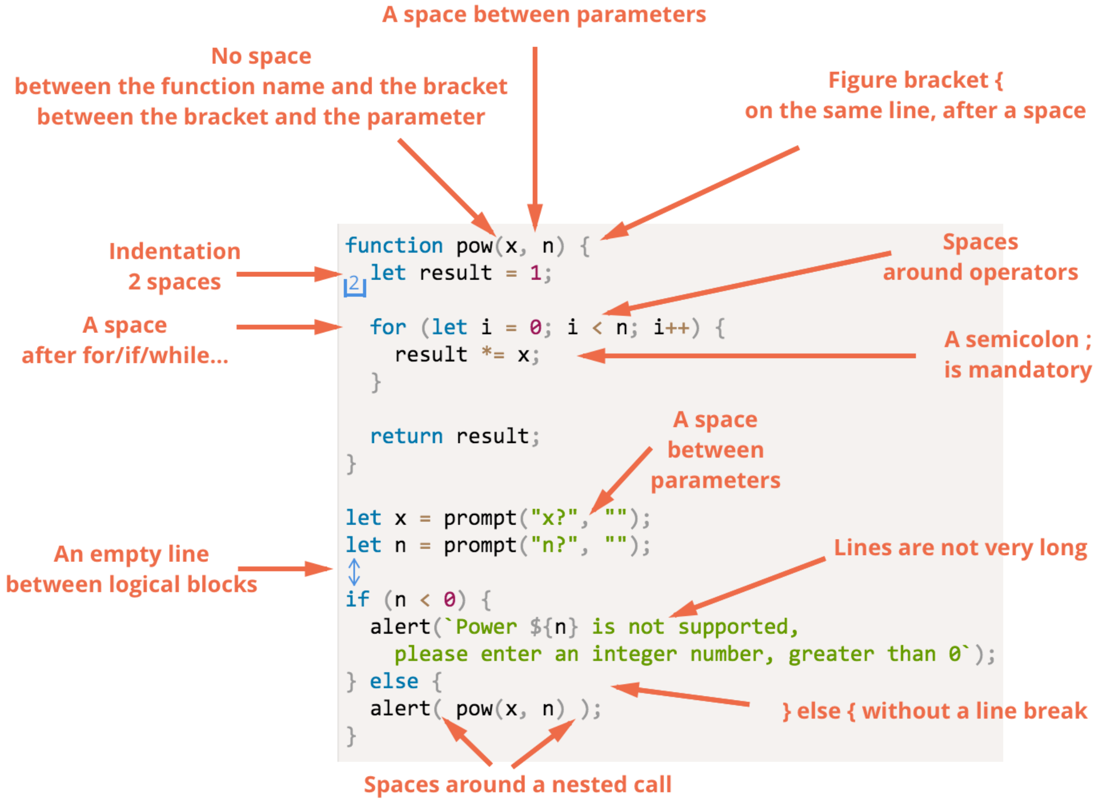
Exercise (18)

- Replace the functions grantAccess() and denyAccess() below with arrow functions:

```
function checkAge(age, granted, denied) {
    if (age < 18) denied();
    else granted();
}

let age = prompt('What is your age?', 18);

function grantAccess() {
    alert('Access granted');
}

function denyAccess() {
    alert('Access denied');
}

checkAge(age, grantAccess, denyAccess);
```

Coding Style

- Our code must be as clean and easy to read as possible
- You should follow the following coding style rules:



```
function pow(x, n) {
  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}

let x = prompt("x?", "");
let n = prompt("n?", "");

if (n < 0) {
  alert(`Power ${n} is not supported,
    please enter an integer number, greater than 0`);
} else {
  alert( pow(x, n) );
}
```

A space between parameters

No space
between the function name and the bracket
between the bracket and the parameter

Figure bracket {
on the same line, after a space

Indentation
2 spaces

Spaces
around operators

A space
after for/if/while…

A semicolon ;
is mandatory

A space
between
parameters

An empty line
between logical blocks

Lines are not very long

} else { without a line break

Spaces around a nested call

Debugging in Chrome

- All modern browsers support “debugging” – a special UI in developer tools that makes finding and fixing errors much easier
- We’ll be using Chrome here, because it’s probably the most feature-rich in this aspect
- Create the following example page and open it in Chrome:

```
function ask(question, yes, no) {
    if (confirm(question)) yes()
    else no();
}
function showOk() {
    alert('You agreed.');
}
function showCancel() {
    alert('You canceled the execution.');
}
// usage: functions showOk, showCancel are passed as arguments to ask
ask('Do you agree?', showOk, showCancel);
```

Debugging in Chrome

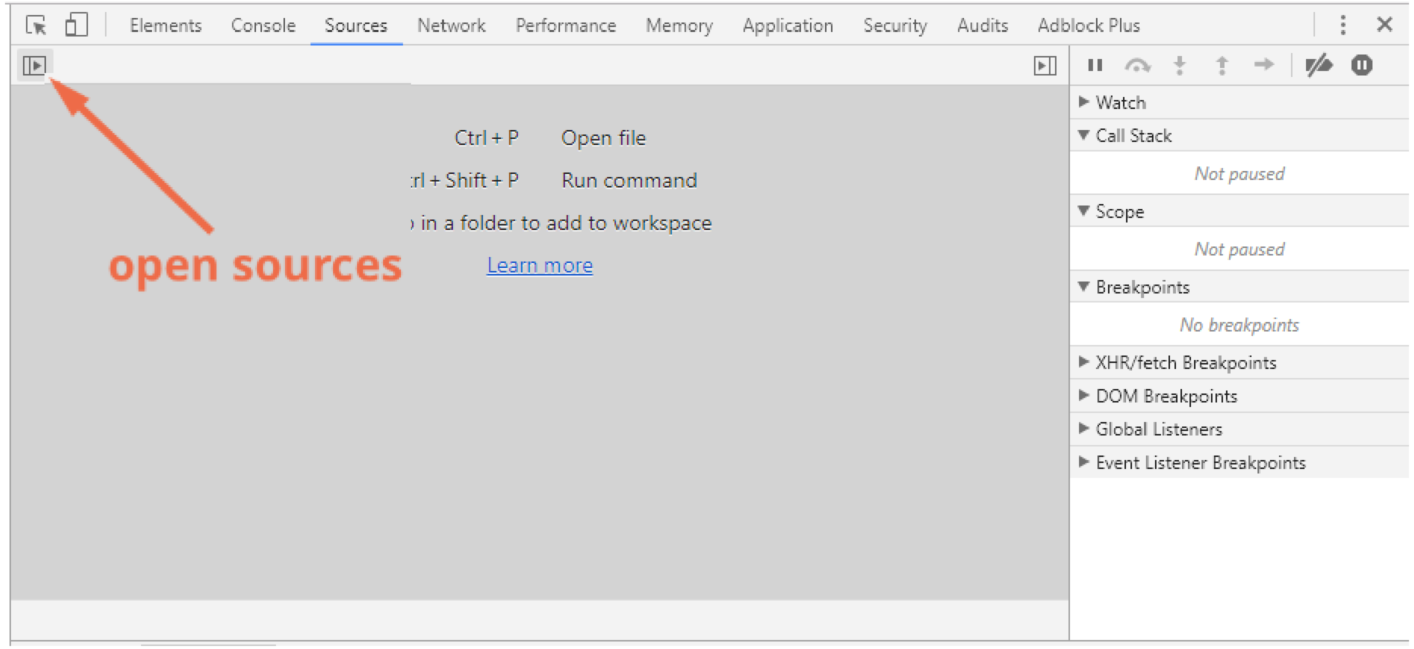
- All modern browsers support “debugging” – a special UI in developer tools that makes finding and fixing errors much easier
- We’ll be using Chrome here, because it’s probably the most feature-rich in this aspect
- Create the following index.html page and hello.js script:

```html
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <script src="hello.js"></script>

    An example for debugging.

    <script>
        hello("John");
    </script>
</body>
</html>
```

```javascript
// hello.js
function hello(name) {
    let phrase = `Hello, ${name}!`;
    say(phrase);
}

function say(phrase) {
    alert(`** ${phrase} **`);
}
```

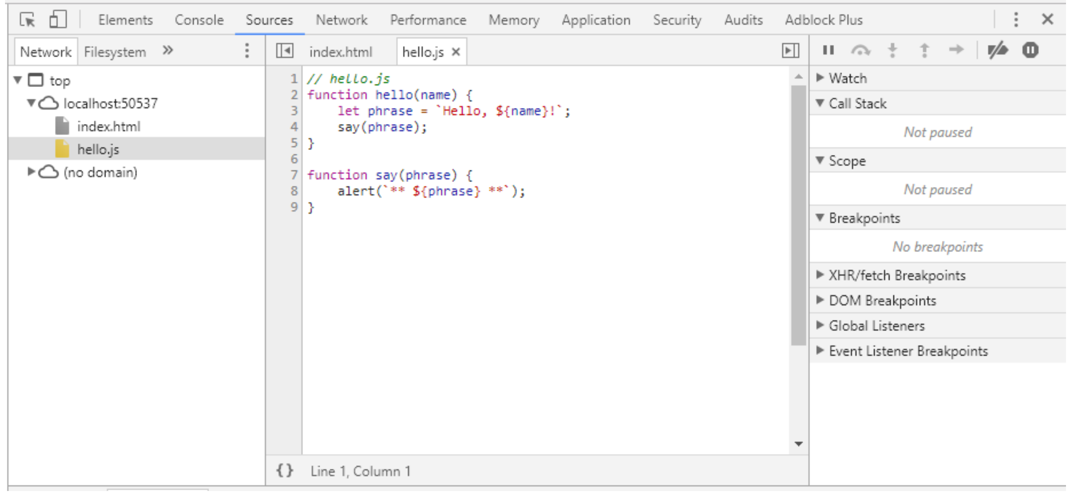
Debugging in Chrome

- Open the HTML page in Chrome
- Turn on developer tools with F12
- Select the sources pane
- Here's what you should see if you are doing it for the first time:



Debugging in Chrome

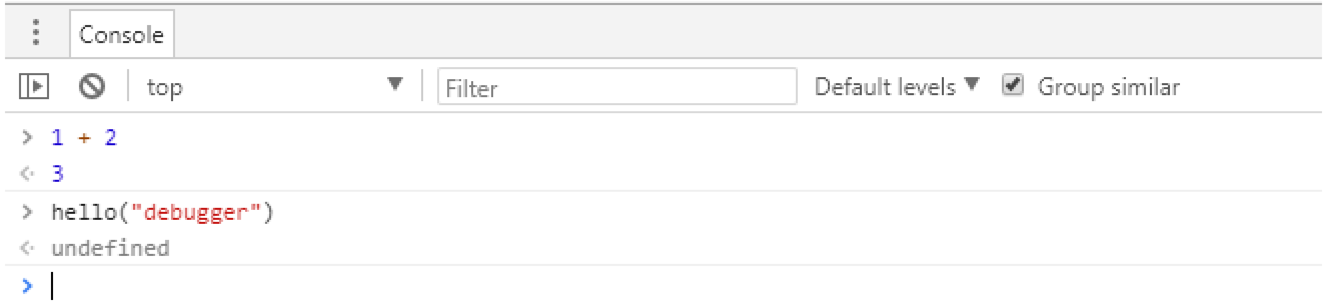
- The toggler button opens the tab with files
- Let’s click it and select index.html and then hello.js in the tree view
- Here we can see three zones:
  - The **Resources zone** lists HTML, JavaScript, CSS and other files, including images that are attached to the page
  - The **Source zone** shows the source code
  - The **Information and control zone** is for debugging, we’ll explore it soon
- Now you could click the same toggler again to hide the resources list and give the code some space



```
// hello.js
function hello(name) {
    let phrase = `Hello, ${name}!`;
    say(phrase);
}

function say(phrase) {
    alert(`** ${phrase} **`);
}
```

Console

- If we press Esc, then a console opens below
- We can type commands there and press Enter to execute
- After a statement is executed, its result is shown below.
- For example, here 1+2 results in 3, and hello("debugger") returns nothing, so the result is undefined:

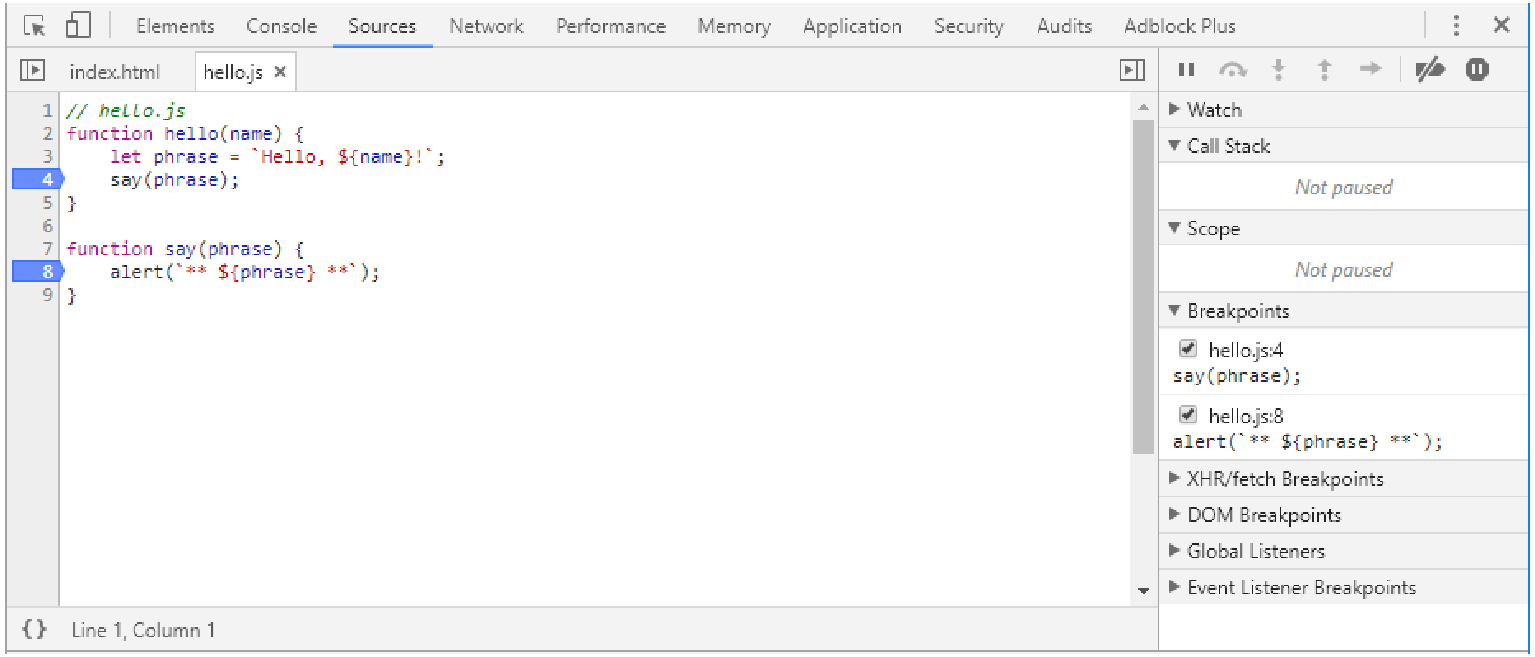


```
> 1 + 2
<· 3
> hello("debugger")
<· undefined
> |
```

Breakpoints

- Let’s examine what’s going on within the code of the example page
- In hello.js, click at line number 4. Yes, right on the 4 digit, not on the code.
- Congratulations! You’ve set a breakpoint. Please also click on the number for line 8.
- It should look like this (blue is where you should click):

breakpoints



```
1 // hello.js
2 function hello(name) {
3     let phrase = `Hello, ${name}!`;
4     say(phrase);
5 }
6
7 function say(phrase) {
8     alert(`** ${phrase} **`);
9 }
```

Call Stack: *Not paused*

Scope: *Not paused*

Breakpoints

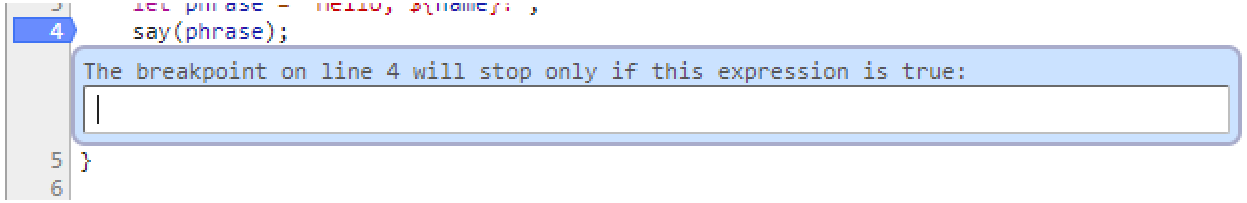
☑ hello.js:4
say(phrase);

☑ hello.js:8
alert(`** ${phrase} **`);

Line 1, Column 1

Breakpoints

- A **breakpoint** is a point of code where the debugger will automatically pause the JavaScript execution
- While the code is paused, we can examine current variables, execute commands in the console etc. In other words, we can debug it.
- We can always find a list of breakpoints in the right pane
- Right click on the line number allows to create a **conditional** breakpoint
  - It only triggers when the given expression is truthy
  - That's handy when we need to stop only for a certain variable value or for certain function parameters



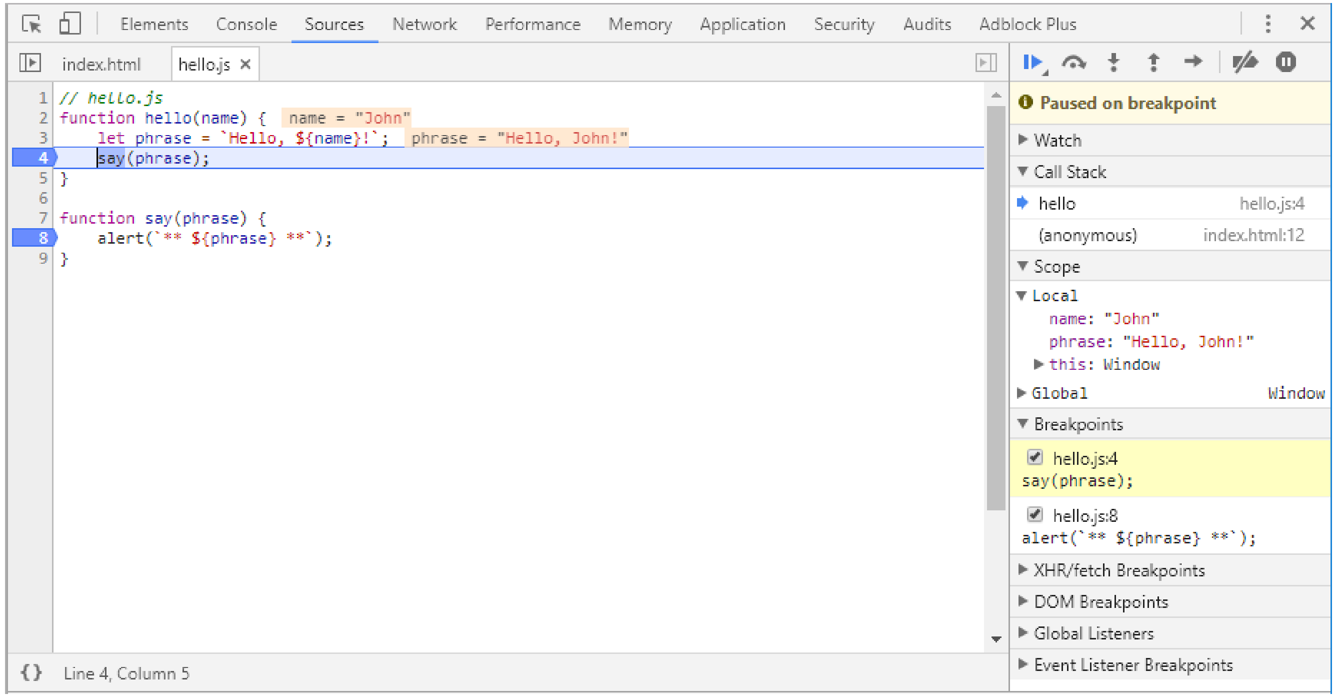
```
3      let phrase = `Hello, ${name}!`;
4      say(phrase);
```

The breakpoint on line 4 will stop only if this expression is true:

```
5  }
6
```

Pause and Look Around

- In our example, hello() is called during the page load, so the easiest way to activate the debugger is to reload the page.
- So let’s press F5 (Windows, Linux) or Cmd+R (Mac)
- As the breakpoint is set, the execution pauses at the 4th line:



```
1  // hello.js
2  function hello(name) {  name = "John"
3      let phrase = `Hello, ${name}!`;  phrase = "Hello, John!"
4      say(phrase);
5  }
6
7  function say(phrase) {
8      alert(`** ${phrase} **`);
9  }
```

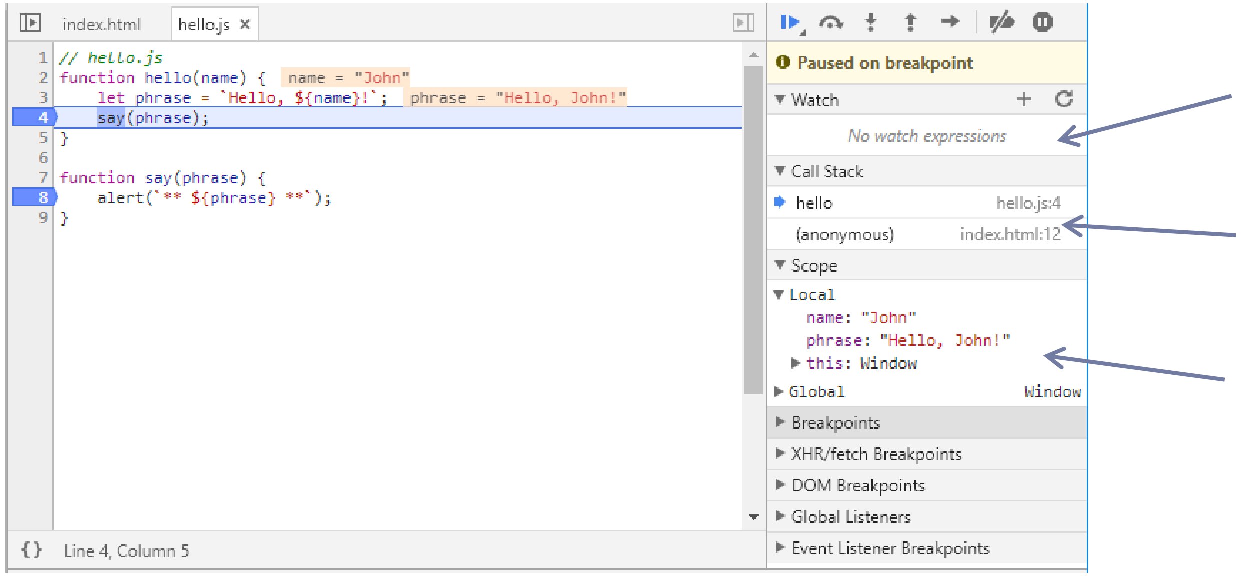
Line 4, Column 5

Paused on breakpoint

- Watch
- Call Stack
  - hello — hello.js:4
  - (anonymous) — index.html:12
- Scope
  - Local
    - name: "John"
    - phrase: "Hello, John!"
    - this: Window
  - Global — Window
- Breakpoints
  - ☑ hello.js:4 `say(phrase);`
  - ☑ hello.js:8 ``alert(`** ${phrase} **`);``
- XHR/fetch Breakpoints
- DOM Breakpoints
- Global Listeners
- Event Listener Breakpoints

Pause and Look Around

- In our example, hello() is called during the page load, so the easiest way to activate the debugger is to reload the page.
- So let’s press F5 (Windows, Linux) or Cmd+R (Mac)
- As the breakpoint is set, the execution pauses at the 4th line:



```
1 // hello.js
2 function hello(name) {   name = "John"
3     let phrase = `Hello, ${name}!`;   phrase = "Hello, John!"
4     say(phrase);
5 }
6
7 function say(phrase) {
8     alert(`** ${phrase} **`);
9 }
```

Line 4, Column 5

Paused on breakpoint

hello hello.js:4
(anonymous) index.html:12

```
name: "John"
phrase: "Hello, John!"
this: Window
Global Window
```

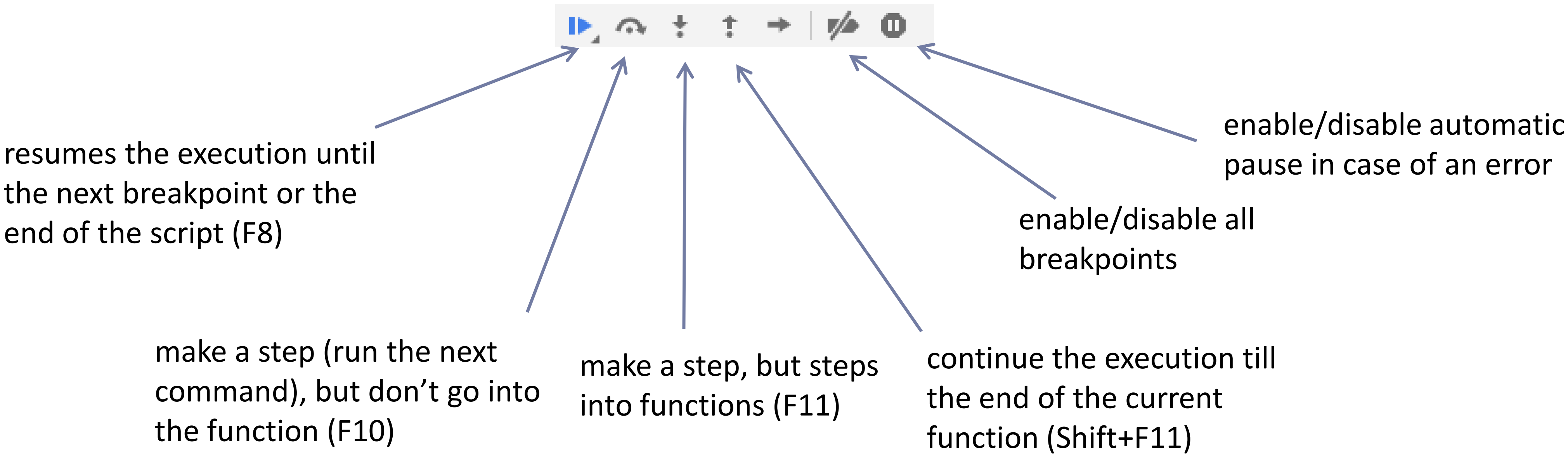
Watch – shows current values for any expressions that you enter

Call Stack – shows the nested calls chain. If you click on a stack item, the debugger jumps to the corresponding code

Scope – current variables.
Local shows local function variables.
Global has global variables (out of any functions).

Tracing the Execution

- Now it's time to *trace* the script
- There are buttons for it at the top of the right pane



resumes the execution until the next breakpoint or the end of the script (F8)

make a step (run the next command), but don't go into the function (F10)

make a step, but steps into functions (F11)

continue the execution till the end of the current function (Shift+F11)

enable/disable all breakpoints

enable/disable automatic pause in case of an error