1 - Javascript Basics

Introduction

Full documentation: https://javascript.info/js

JavaScript programs can be inserted almost anywhere into an HTML document using the `<script>` tag.

```
<!DOCTYPE HTML>
<html>

<body>

  <p>Before the script...</p>

  <script>
    alert( 'Hello, world!' );
  </script>

  <p>...After the script.</p>

</body>

</html>
```

The tag contains Javascript code which is automatically executed when the browser processes the tag.

The script tag has some attributes that are rarely used but can still be found in the code:

- The **type** attribute: `<script type=…>`, the old HTML4 standard required a script to have a type, usually it was
  `type="text/javascript"`, now it is not required anymore. Now this tag can be used for JS modules.
- The **language** attribute : `<script language=…>`, it used to specify the language of the script but now Javascript is the only language.

External scripts

If we have a lot of JS code, we can put it into separate files.
Script files are attached to HTML with the `src` attribute:

`<script src="/path/to/script.js">`

Here `/path/to/script` is an absolute path to the script from the site root, we can also specify a relative path from the current page, using something like src="script.js", just like src="./script.js" would mean a file "script.js" in the current folder. We could also pass a full URL to a file. To attach multiple files we can just use multiple tags:

`<script src="/path/to/script.js">`
`<script src="/path/to/script2.js">`

Normally more complex scripts are stored in different file rather than inside an HTML, so that the browser will store those files in its cache, this reduces the traffic and makes the pages faster.

If `src` is set, the code inside the tag is ignored as we cannot have both src and code inside.

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

this way i can execute both src script and normal script.

External scripts.zip

Code Structure

Statements

Statements are syntax constructs and commands that performs actions. We can have as many as we want in our code and they can be separated by a semicolon.

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

Javascript interpets the like break as an implicit semicolon, this is also **called automatic semicolon insertion.**

```
alert("Hello")

[1, 2].forEach(alert);
```

In this case a semicolon is mandatory as Javascript would interpret those line as a single continous line. It's safer just to use semicolons everytime.

Comments

We can place comments by using double slashes :

`// this is a comment`

for multi-line comments we can use /* and */

"use strict"

When this line is placed on top of a script, the whole script works the "modern" way. This can be put at the beginning of a function, anda that enables strict mode in that function only, but usually it used for the whole script.
Modern JS uses classes and modules that enables "use strict" by default.

Variables

To declare a variable in JS, we can use something like:

`let message = "hello" ;`

then we can assign data to it using the = operator.

In older versions of JS we could also declare a variable using `var` keyword. Variable naming follow teo rules: the name must contanin only letters, digits or $ and _ keywords, and the first character must not be a digit.

Contstants

To declare a constant unchanging value we use the keyword `const`:

```
const Birthday = '03.01.1997';
```

const variables cannot be reassigned. If a programmer is sure that a variable will never change, he can use the const keyword.

Data types

Javascript is a dynamically typed language, so the declared variables are not tied to a single type, we could write something like:

```
let message = "hello";
message = 123456;
```

Mathematical Operations

There are many operations for numbers, e.g. multiplication *, division /, addition +, subtraction -, and so on.

Besides regular numbers, there are so-called “special numeric values” which also belong to this data type: Infinity, -Infinity and NaN.

```
alert( 1 / 0 ); // Infinity
alert( NaN + 1 ); // NaN
alert( 3 * NaN ); // NaN
alert( "not a number" / 2 - 1 ); // NaN
```

So to say, all odd integers greater than (2^53-1) can’t be stored at all in the “number” type.

For most purposes ±(2^53-1) range is quite enough, but sometimes we need the entire range of really big integers, e.g. for cryptography or microsecond-precision timestamps.

BigInt type was recently added to the language to represent integers of arbitrary length.

A BigInt value is created by appending n to the end of an integer:

```
const bigInt = 1234567890123456789012345678901234567890n; //the n at the end means its a big int
```

String

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
```

Bool

Boolean values also come as a result of comparisons:

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

Objects and Symbols

The object type is special.

All other types are called “primitive” because their values can contain only a single thing (be it a string or a number or whatever). In contrast, objects are used to store collections of data and more complex entities.

Being that important, objects deserve a special treatment. We’ll deal with them later in the chapter Objects, after we learn more about primitives.

The symbol type is used to create unique identifiers for objects. We have to mention it here for the sake of completeness, but also postpone the details till we know objects.

Typeof operator

TYpeof operator returns the type of the argument.

```
typeof undefined // "undefined"

typeof 0 // "number"
typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object"  (1)

typeof null // "object"  (2)

typeof alert // "function"  (3)
```

Math is a built-in object that provides mathematical operations.
The result of **typeof null** is "object". That's an officially recognized error in typeof, coming from very early days of JavaScript and kept for compatibility. Definitely, null is not an object. It is a special value with a separate type of its own. The behavior of typeof is wrong here.
The result of **typeof alert** is "function", because alert is a function.

Both `typeof(x)` and `typeof x` syntax are correct.

Interaction

alert

Shows a message and waits for the user to press "OK".

prompt

This function accepts two arguments:

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

It shows a modal window with a text message, an input field for the visitor, and the buttons OK/Cancel. The square brackets around default in the syntax above denote that the parameter is optional, not required.

The visitor can type something in the prompt input field and press OK. Then we get that text in the result. Or they can cancel the input by pressing Cancel or hitting the Esc key, then we get null as the result.

The call to prompt returns the text from the input 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!
```

confirm

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

The function confirm shows a modal window with a question and two buttons: OK and Cancel.
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
```

All these methods are modal: they pause script execution and don’t allow the visitor to interact with the rest of the page until the window has been dismissed.
There are two limitations shared by all the methods above:

- The exact location of the modal window is determined by the browser. Usually, it’s in the center.
- The exact look of the window also depends on the browser. We can’t modify it.

That is the price for simplicity. There are other ways to show nicer windows and richer interaction with the visitor, but if “bells and whistles” do not matter much, these methods work just fine.

Type conversion

```
alert( Number("   123   ") ); // 123
alert( Number("123z") );      // NaN (error reading a number at "z")
alert( Number(true) );        // 1
alert( Number(false) );       // 0
```

Please note that null and undefined behave differently here: null becomes zero while undefined becomes NaN.

Boolean conversion

- Values that are intuitively “empty”, 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

alert( Boolean("0") ); // true
alert( Boolean(" ") ); // spaces, also true (any non-empty string is true)
```

Other conversions

```
let apples = "2";
let oranges = "3";
alert( apples + oranges ); // "23", the binary plus concatenates strings

let apples = "2";
let oranges = "3";
// both values converted to numbers before the binary plus
alert( +apples + +oranges ); // 5
```

Examples:

```
"" + 1 + 0 = "10" // (1)
"" - 1 + 0 = -1 // (2)
true + false = 1
6 / "3" = 2
"2" * "3" = 6
4 + 5 + "px" = "9px"
"$" + 4 + 5 = "$45"
"4" - 2 = 2
```

```
"4px" - 2 = NaN
"  -9  " + 5 = "  -9  5" // (3)
"  -9  " - 5 = -14 // (4)
null + 1 = 1 // (5)
undefined + 1 = NaN // (6)
" \t \n" - 2 = -2 // (7)
```

- The addition with a string "" + 1 converts 1 to a string: "" + 1 = "1", and then we have "1" + 0, the same rule is applied.
- The subtraction - (like most math operations) only works with numbers, it converts an empty string "" to 0.
- The addition with a string appends the number 5 to the string.
- The subtraction always converts to numbers, so it makes " -9 " a number -9 (ignoring spaces around it).
- null becomes 0 after the numeric conversion.
- undefined becomes NaN after the numeric conversion.
- Space characters, are trimmed off string start and end when a string is converted to a number. Here the whole string consists of space characters, such as \t, \n and a “regular” space between them. So, similarly to an empty string, it becomes 0.

Comparisons

General rules:

- **Treat any comparison with undefined/null except the strict equality === with exceptional care.**
- **Don’t use comparisons >= > < <= with a variable which may be null/undefined, unless you’re really sure of what you’re doing. If a variable can have these values, check for them separately.**
- Comparison operators return a boolean value.
- Strings are compared letter-by-letter in the “dictionary” order.
- When values of different types are compared, they get converted to numbers (with the exclusion of a strict equality check).
- The values null and undefined equal == each other and do not equal any other value.

Labels

Sometimes we need to break out from multiple nested loops at once.

For example, in the code below we loop over i and j, prompting for the coordinates (i, j) from (0,0) to (2,2):

~~~
for (let i = 0; i < 3; i++) {

  for (let j = 0; j < 3; j++) {

    let input = prompt(`Value at coords (${i},${j})`, '');

    // what if we want to exit from here to Done (below)?
  }Callback Functions

Let’s look at more examples of passing functions as values and using
function expressions.

We’ll write a function ask(question, yes, no) with three 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 should ask the question and, depending on the user’s answer,
call yes() or no():

```js
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);
```

The arguments showOk and showCancel of ask are called callback functions or just callbacks.

Using Function Espressions:

```
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 (because they are not assigned to variables), but that's just what we want here. Such code appears in our scripts very naturally, it's in the spirit of JavaScript.

For a better overview of Function Expression and Declaration, please visit:
https://javascript.info/function-expressions

Function declaration or function Expression?

As a rule of thumb, when we need to declare a function, the first thing to consider is Function Declaration syntax. It gives more freedom in how to organize our code, because we can call such functions before they are declared.That's also better for readability, as it's easier to look up function f(…) {…} in the code than let f = function(…) {…};. Function Declarations are more "eye-catching". But if a Function Declaration does not suit us for some reason, or we need a conditional declaration (we've just seen an example), then Function Expression should be used.

}

alert('Done!');

```

We need a way to stop the process if the user cancels the input.

The ordinary break after input would only break the inner loop. That's not
```

~~~
sufficient – labels, come to the rescue!

A label is an identifier with a colon before a loop:

```js
labelName: for (...) {
  ...
}
~~~

```js
outer: for (let i = 0; i < 3; i++) {

  for (let j = 0; j < 3; j++) {

    let input = prompt(`Value at coords (${i},${j})`, '');

    // if an empty string or canceled, then break out of both loops
    if (!input) break outer; // (*)

    // do something with the value...
  }
}

alert('Done!');
```

A break directive must be inside a code block. Technically, any labelled code block will do, e.g.:

```js
label: {
  // ...
  break label; // works
  // ...
}
```

Function Expressions

```
let sayHi = function() {
  alert( "Hello" );
};
```

Here we can see a variable sayHi getting a value, the new function, created as function() { alert("Hello"); }.

As the function creation happens in the context of the assignment expression (to the right side of =), this is a Function Expression.

Please note, there’s no name after the function keyword. Omitting a name is allowed for Function Expressions.

Here we immediately assign it to the variable, so the meaning of these code samples is the same: "create a function and put it into the variable sayHi".

In more advanced situations, that we’ll come across later, a function may be created and immediately called or scheduled for a later execution, not stored anywhere, thus remaining anonymous.

Function is a value

Let’s reiterate: no matter how the function is created, a function is a value. Both examples above store a function in the sayHi variable.

We can even print out that value using alert:

```
function sayHi() {
  alert( "Hello" );
}

alert( sayHi ); // shows the function code
```

Please note that the last line does not run the function, because there are no parentheses after sayHi. There are programming languages where any mention of a function name causes its execution, but JavaScript is not like that.

In JavaScript, a function is a value, so we can deal with it as a value. The code above shows its string representation, which is the source code.

Surely, a function is a special value, in the sense that we can call it like sayHi().

But it’s still a value. So we can work with it like with other kinds of values.

```
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)
```

Here’s what happens above in detail:

- List itemThe Function Declaration (1) creates the function and puts it into the variable named sayHi.
- List itemLine (2) copies it into the variable func. Please note again: there are no parentheses after sayHi. If there were, then func = sayHi() would write the result of the call sayHi() into func, not the function sayHi itself.
- List itemNow the function can be called as both sayHi() and func().

Callback Functions

Callback Functions

Let’s look at more examples of passing functions as values and using function expressions.

We’ll write a function ask(question, yes, no) with three 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 should ask the question and, depending on the user’s answer, call yes() or no():

```
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);
```

The arguments showOk and showCancel of ask are called callback functions or just callbacks.

Using Function Espressions:

```
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 (because they are not assigned to variables), but that’s just what we want here. Such code appears in our scripts very naturally, it’s in the spirit of JavaScript.

For a better overview of Function Expression and Declaration, please visit:
https://javascript.info/function-expressions

Function declaration or function Expression?

As a rule of thumb, when we need to declare a function, the first thing to consider is Function Declaration syntax. It gives more freedom in how to organize our code, because we can call such functions before they are declared.That’s also better for readability, as it’s easier to look up function f(…) {…} in the code than let f = function(…) {…};. Function Declarations are more “eye-catching”. But if a Function Declaration does not suit us for some reason, or we need a conditional declaration (we’ve just seen an example), then Function Expression should be used.

Arrow Functions

There’s another 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 accepts arguments arg1..argN, then evaluates the expression on the right side with their use and returns its result.

```
let sum = (a, b) => a + b;

/* This arrow function is a shorter form of:

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

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

Arrow functions can be used in the same way as Function Expressions.

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

let welcome = (age < 18) ?
  () => alert('Hello!') :
  () => alert("Greetings!");

welcome();
```

Arrow functions may appear unfamiliar and not very readable at first, but that quickly changes as the eyes get used to the structure. They are very convenient for simple one-line actions, when we’re just too lazy to write many words.

We can also declare multi-line arrow functions:

```
let sum = (a, b) => {  // the curly brace opens a multiline function
  let result = a + b;
  return result; // if we use curly braces, then we need an explicit
"return"
};
```

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