JavaScript

Why JavaScript? What is JavaScript?

- JavaScript is a programming language that allows you to implement **complex things** on web pages.
  - displaying timely content updates, or interactive maps, or animated 2D/3D graphics, or scrolling video jukeboxes, etc.
- JavaScript is a scripting language that enables you to create dynamically updating content, control multimedia, animate images, and pretty much everything else.

More on JavaScript Use

- The JavaScript is executed by the browser's JavaScript engine, after the HTML and CSS have been assembled and put together into a web page.
- A very common use of JavaScript is to dynamically modify HTML and CSS to update a user interface, via the Document Object Model API.

Document Object Model API

- The DOM (Document Object Model) API allows you to manipulate HTML and CSS, creating, removing and changing HTML, dynamically applying new styles to your page, etc.

Interpreted vs Compiled Code

- JavaScript is an interpreted language — the code is run from top to bottom and the result of running the code is immediately returned. You don't have to transform the code into a different form before the browser runs it.
- Compiled languages on the other hand are transformed (compiled) into another form before they are run by the computer. For example C/C++ are compiled into assembly language that is then run by the computer.

Server-Side vs Client Side Code

- Client-side code is code that is run on the user's computer — when a web page is viewed, the page's client-side code is downloaded, then run and displayed by the browser. In this JavaScript module we are explicitly talking about **client-side JavaScript**.
- Server-side code on the other hand is run on the server, then its results are downloaded and displayed in the browser. Examples of popular server-side web languages include PHP, Python, Ruby, and ASP.NET.
- And JavaScript! JavaScript can also be used as a server-side language, for example in the popular Node.js environment.

How do you add JavaScript to your HTML Page?

1. Internal JavaScript
2. External JavaScript
3. Inline JavaScript Handlers

1. Internal JavaScript

- Write Html file.
- Next, go to your text editor and add the following just before your closing </body> tag:
- <script>
-   // JavaScript goes here
- </script>

2. External JavaScript

- First, create a new file in the same directory as your sample HTML file.
- Make sure it has that .js filename extension, as that's how it is recognized as JavaScript.
- Next, copy all of the script out of your current <script> element and paste it into the .js file. Save that file.
- Now replace your current <script> element with the <script src=“scriptwhatever.js”></script>

3. Inline JavaScript Handlers

- Something like:

button onclick="createParagraph()">Click me!</button>

Is inline JavaScript Handler.

It is not a good practice to pollute your HTML with JavaScript handlers.

You'd have to include the onclick="createParagraph()" attribute on every button you wanted the JavaScript to apply to.

```
var buttons = document.querySelectorAll('button’);
for (var i = 0; i < buttons.length ; i++)
{
      buttons[i].addEventListener('click', createParagraph);
}
```

JavaScript Fundamentals

- Hello, world!
- Code structure
- The modern mode, "use strict"
- Variables
- Data types
- Type Conversions
- Operators
- Comparisons
- Interaction: alert, prompt, confirm
- Conditional operators: if, '?'
- Logical operators
- Loops: while and for
- The "switch" statement
- Functions
- Function expressions and arrows
- JavaScript specials

Hello World!

- <!DOCTYPE HTML>
- <html>

- <body>

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

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

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

- </body>

- </html>

Code Structure

- Statements
- Semicolons
- Comments

Use Strict

- The "use strict" directive switches the engine to the “modern” mode, changing the behavior of some built-in features. We’ll see the details as we study.
- The strict mode is enabled by "use strict" at the top. Also there are several language features like “classes” and “modules” that enable strict mode automatically.
- The strict mode is supported by all modern browsers.
- It’s always recommended to start scripts with "use strict".

- alert("some code");
- // "use strict" below is ignored, must be on the top
- "use strict";
- // strict mode is not activated

Variables

- Named storage.
- Keyword used is “let”.

let yourName;

- *Assign value with declaration:* let yourName = “abc”;
- Now string is stored in the memory; with following statement:
  - alert(yourName); //output will be abc
- Declare multiple variables in one line; for example:
  - let yourName=abc, age=10, comment=“hello”; //not recommended

Var in place of let

- Var is in older scripts.
- There are subtle differences but they do not matter now.

Variable Naming

- The name must contain only letters, digits, symbols $ and _.
- The first character must not be a digit.
- Case matters
- There is a list of reserved words, which cannot be used as variable names, because they are used by the language itself.
- An assignment without strict and declaration gives error.

Incorrect Naming:

- let 5a; // cannot start with a digit
- let first-name; // a hyphen '-' is not allowed in the name
- let mes;
- let Mes;
- let class;
- "use strict";
- num = 5; // error: num is not defined

Constants

- To declare a constant (unchanging) variable, one can use const instead of let.
- We generally use upper case for constants that are “hard-coded”. Or, when the value is known prior to execution and directly written into the code.

- const myBirthday = ‘19.04.1982';
- myBirthday = '01.01.2000'; // error, can't reassign the constant!

```
const BIRTHDAY = '18.04.1982'; // make uppercase?

const AGE = someCode(BIRTHDAY); // make uppercase?
```

Naming your variables

- Use human-readable names like userName or shoppingCart.
- Stay away from abbreviations or short names like a, b, c, unless you really know what you're doing.
- Make the name maximally descriptive and concise. Examples of bad names are data and value. Such a name says nothing. It is only ok to use them if it's exceptionally obvious from the context which data or value is meant.
- Agree on terms within your team and in your own mind. If a site visitor is called a "user" then we should name related variables like currentUser or newUser, but not currentVisitor or a newManInTown.

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:
    - // no error
    - let message = "hello";
    - message = 123456
- Therefore JavaScript is known as “dynamically typed”

Seven basic Datatypes 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.
- The **typeof** operator allows us to see which type is stored in the variable.

A number

- The number type serves both for integer and floating point numbers.
- There are many operations for numbers, e.g. multiplication *, division /, addition +, subtraction - and so on.
- Special numeric values" which also belong to that type: **Infinity**, **-Infinity** and **NaN**.
- Infinity represents the mathematical Infinity ∞. It is a special value that's greater than any number. **alert(1 / 0);**
- We can get it as a result of division by zero: **alert(Infinity);**

A number -- continued

- **NaN** represents a computational error. It is a result of an incorrect or an undefined mathematical operation, for instance:
  - alert("not a number" / 6); // NaN, such division is erroneous

A string

- A string in JavaScript must be quoted.
  - let str = "Hello";
  - let str2 = 'Single quotes are ok too';
  - let phrase = `can embed ${str}`;
  - alert(str+ ‘Tong’);

```
let name = "John";

// embed a variable
alert('hello '+ name +'!'); // Hello, John!

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

A boolean

- The boolean type has only two values:
  - true and false

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

```
let isGreater = 4 > 1;

alert( isGreater ); // true (the comparison result is "yes")
```

A null value

- The special null value does not belong to any type of those described above.
- It forms a separate type of its own, which contains only the null value:
  - let age = null;
- In JavaScript null is not a “reference to a non-existing object” or a “null pointer” like in some other languages.
- It’s just a special value which has the sense of “nothing”, “empty” or “value unknown”.
- The code above states that the age is unknown or empty for some reason.

An “undefined” value

- The special value undefined stands apart. It 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“
- 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 or similar.

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.
- The symbol type is used to create unique identifiers for 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).

```
typeof undefined // "undefined"

typeof 0 // "number"

typeof true // "boolean"

typeof "foo" // "string"

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

typeof Math // "object"  (1)

typeof null // "object"  (2)

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

Quick Question

- What is the output?
  - let name = “Adam";
  - alert(`hello ${1}`); // ?
  - alert(`hello ${"name"}`); // ?
  - alert(`hello ${name}`); // ?
  - alert('hello '+ name +'!')

Operators

- Operand and operator
- Unary operator: operator with one operand
- Binary operator: operator with two operands
  - if the binary + is applied to strings, it merges (concatenates) them:
  - if any of operands is a string, then the other one is converted to a string too.
  - String concatenation and conversion is a special feature of the binary plus "+". Other arithmetic operators work only with numbers. They always convert their operands to numbers.
  - the plus operator + applied to a single value, doesn’t do anything with numbers, but if the operand is not a number, then it is converted into it.

Binary +

- let string1 = “3";
- let string2 = “4";
- alert(string1 + string2); // “34",

```
let string1 = “3";
let string2 = “4";
// both values converted to numbers before the binary plus
alert( +string1 + +string2 ); // 7

// the longer variant
// alert( Number(string1) + Number(string2) ); // 7
```

Remainder operator %

- The remainder operator % does not have a relation to percents.

- The result of x % y is the remainder of the integer division of x by y.
- 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.
- For a natural number x the result of x ** y is a multiplied by itself y times.

- For instance:

```
alert( 3 ** 2 ); // 9  (3 * 3)
alert( 3 ** 3 ); // 27  (3 * 3 * 3)
alert( 3 ** 4 ); // 81 (3 * 3 * 3 * 3)
```

Increment and decrement

- Increment ++ increases a variable by 1:

```
let counter = 3;
counter++;      // works same as counter = counter + 1, but shorter
alert( counter ); // 4
```

- Decrement -- decreases a variable by 1:

```
let counter = 4;
counter--;      // works same as counter = counter - 1, but shorter
alert( counter ); // 3
```

Prefix and postfix form

- 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++.
- The “prefix form” is when the operator stands before the variable: ++counter.

```
let counter = 0;
alert( ++counter ); // 1
```

```
let counter = 0;
alert( counter++ ); // 0
```

Quick Check

- What should be the output?
  - let a = 2;
  - let x = 1 + (a *= 2);

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.
- Result is always **Boolean**

String Comparisons

- strings are compared letter-by-letter.
- Strings "Glow" and "Glee" are compared character-by-character:

  - G is the same as G.
  - l is the same as l.
  - o is greater than e. Stop here. The first string is greater.

Strict Equality

- A regular equality check "==" has a problem. It cannot differ 0 from false:
  - alert(0 == false); // true
  - alert('' == false); // true
- A strict equality operator === checks the equality without type conversion.
  - alert(0 === false); // false, because the types are different
- There also exists a “strict non-equality” operator !==, as an analogy for !=.

null and undefined

- Values null and undefined equal == each other and do not equal any other value.
- Be careful when using comparisons like > or < with variables that can occasionally be null/undefined.

Interaction

Syntax: alert(message);
e.g alert(“hello”);

- alert
- prompt
- confirm

result = prompt(title[, default]);

```
let age = prompt('How old are you?', 100);

alert(`You are ${age} years old!`); // You are 100 years old!
```

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

```
let isStudent = confirm("Are you the student?");

alert( isStudent); // true if OK is pressed
```

Interaction Methods:Modal

- All these methods are modal: they pause the script execution and don’t allow the visitor to interact with the rest of the page until the message 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.

Conditional Operators if, ?

- let year = prompt('In which year was ECMAScript-2015 specification published?', '');
- if (year == 2015) alert('You are right!');

```
let year = prompt('In which year was ECMAScript-2015
specification published?', '');

if (year == 2015) {
  alert( 'You guessed it right!' );
}
else {
  alert( 'How can you be so wrong?' ); // any value except
2015
}
```

```
let year = prompt('In which year was
ECMAScript-2015 specification published?',
'');

if (year < 2015) {
  alert( 'Too early...' );
} else if (year > 2015) {
  alert( 'Too late' );
} else {
  alert( 'Exactly!' );
}
```

Ternary operator ?

- let result = condition ? value1 : value2

- let accessAllowed = (age > 18) ? true : false;

Loops

- while (condition) {
-   // code
-   // so-called "loop body"
- }

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

Do while loop

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

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

The for loop

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

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

begin	i = 0	Executes once upon entering the loop.
condition	i < 3	Checked before every loop iteration, if fails the loop stops.
step	i++	Executes after the body on each iteration, but before the condition check.
body	alert(i)	Runs again and again while the condition is truthy

Breaking the loop

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

The break directive is activated in 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.

Continue to the next iteration

- 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
- }

Labels for break/continue

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

```
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 label is the only way for break/continue to escape the nesting and go to the outer loop.

Switch Statement

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- If there is no match, the default code block is executed.

Switch Statement Example

```
let i=5;
switch(i) {
 case 1:
  alert('Too small');
  break;
 case 5:
  alert('Exactly!');
  break;
 case 10:
  alert('Too big');
  break;
 default:
  alert("I don't know");
}
```

Here the switch starts to compare a from the first case variant that is 1. The match fails.

Then 5. That’s a match, so the execution starts from case 5, output “Exactly!”.

If the i = 0; the output will be “I don’t know” because there is no match, the default code block is executed.

JavaScript Function Syntax

Syntax

```
function name(parameter1, parameter2)
{
  // code to be executed
}
```

- Function **parameters** are listed inside the parentheses () in the function definition.
- Function **arguments** are the **values** received by the function when it is invoked.
- A Function is much the same as a Method, Procedure or a Subroutine, in other programming languages.

JavaScript Function Return

```
let x = myFunction(2, 3);
function myFunction(a, b)
{
   return a * b;
}
```

- Function myFunction is called with parameters 2 and 3
- arguments a = 2 and b =3
- Function returns the product of a and b
- Return value will end up in x
- The result in x will be 6

Function expressions and arrows

Function

```
hello = function() {
  return "Hello World!";
}
```

Arrow Function

```
hello = () => "Hello World!";
```

Arrow functions allow us to write shorter function syntax

JavaScript Specials

Variables Declaration

- let
- const (constant)
- var (old-style)

Variable name

- Letters and digits, but the first character may not be a digit.
- Characters $ and _ are normal, on par with letters.

JavaScript Specials

Interaction

prompt(question, [default])

- Ask a question, and return either what the visitor entered or null if they clicked “cancel”.

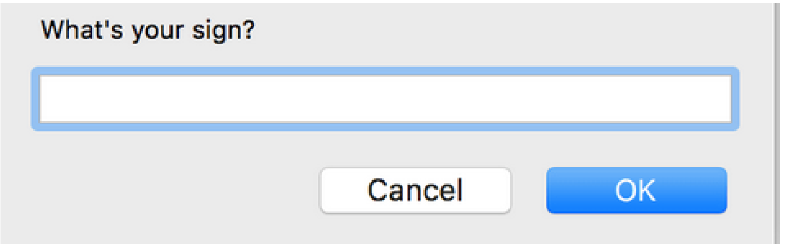
confirm(question)

- Ask a question and suggest to choose between Ok and Cancel. The choice is returned as true/false.

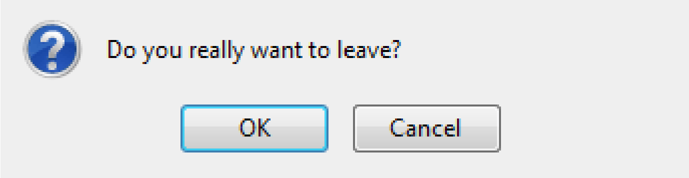
alert(message)

- Output a message.

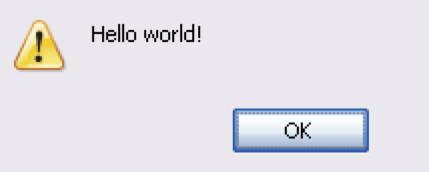
Examples



What's your sign?



Do you really want to leave?



Hello world!