FITPED

Javascript fundamentals

Ján Skalka
Jozef Kapusta
Ľubomír Benko
Arkadiusz Nowakowski
Zenón José Hernández-Figueroa
José Daniel González-Domínguez
Juan Carlos Rodríguez-del-Pino
Jan Francisti
Tomáš Hála

www.fitped.eu

2021

Co-funded by the
Erasmus+ Programme
of the European Union

Work-Based Learning in Future
IT Professionals Education
(Grant. no. 2018-1-SK01-KA203-046382)

JavaScript Fundamentals

Published on

November 2021

Authors

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

Jozef Kapusta | Pedagogical University of Cracow, Poland

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

Arkadiusz Nowakowski | University of Silesia in Katowice, Poland

Zenón José Hernández-Figueroa | University of Las Palmas de Gran Canaria, Spain

José Daniel González-Domínguez | University of Las Palmas de Gran Canaria, Spain

Juan Carlos Rodríguez-del-Pino | University of Las Palmas de Gran Canaria, Spain

Jan Francisti | Constantine the Philosopher University in Nitra, Slovakia

Tomáš Hála | Mendel University in Brno, Czech Republic

Reviewers

Martin Drlík | Constantine the Philosopher University in Nitra, Slovakia

Cyril Klimeš | Mendel University in Brno, Czech Republic

Piet Kommers | Helix5, Netherland

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Peter Švec | Teacher.sk, Slovakia

Graphics

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

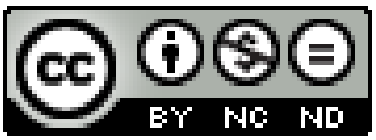
David Sabol | Constantine the Philosopher University in Nitra, Slovakia

Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

Co-funded by the
Erasmus+ Programme
of the European Union



ISBN 978-80-558-1788-0

Table of Contents

Introduction to JavaScript

Chapter 1

1.1 Linking to a JavaScript

🕮 1.1.1

JavaScript is known as the language of modern web browsers. *JavaScript* is a flexible, and fast programming language which is used for web development.

Since *JavaScript* remains at the core of web development, it is often the first language learned by self-taught coders eager to learn and build.

Be careful, JavaScript and Java are completely ***different languages***, both in concept and design. JavaScript was invented by Brendan Eich in 1995.

📝 1.1.2

JavaScript and Java are the same languages, JavaScript is a part of the language Java containing functions aimed at web development.

- False
- True

🕮 1.1.3

JavaScript is the default scripting language in HTML.

What you can do with *JavaScript*:

- Add and remove content from the web page.
- Access and change element attributes, including source and class.
- Insert markup into a web page using function *innerHTML*.
- Change style attribute.

🕮 1.1.4

The primary importance of *JavaScript* is in the area of website creation. JavaScript is inserted into the HTML of the web page. There are two approaches to adding JavaScript to HTML

- Create a **JavaScript source code** in a separate file and then create in the HTML code a **link** to JavaScript.
- Create an **area** in the HMTL code for a JavaScript embedding

📖 1.1.5

For inserting JavaScript code into HTML, JavaScript code is inserted into element **<script>** between **<script>** and **</script>**.

It is able to insert scripts into elements **<head>** or **<body>**.

Both options have their own advantages:

Most of the external scripts which are inserted into web pages, for example, Bootstrap, JQuery and others are inserted into element **<head>**. In the case of inserting custom scripts into **<head>**, we can have an overview of all scripts in one element.

Nowadays is common to use the inserting of code JavaScipt at the end. **<body>**. The reason is the effort for faster loading of web pages. Since the compilation of scripts slows down the showing of pages, it is better to load the content of pages for users and subsequently load scripts.

📝 1.1.6

Which HTML element can contain a program from JavaScript?

- <|javascript>
- <|img>
- <|code>
- <|script>
- <|p>

📝 1.1.7

In HTML code insert tags into element <body> for a Javascript code.

```
<!doctype html>
<html>
  <head>
    <title>First JavaScript</title>
  </head>
    <body>

      _____
      //JavaScript code

      _____
    </body>
```

```
</html>
```

📖 1.1.8

The next option for writing JavaScript code is in a separate file (saved with suffix .js), then to create a link in an HTML code for the file and open it with a browser. The placement of scripts into external files logically divides HTML code from JavaScript and simplifies reading and additional editing of the script.

Alike as inserting Javascript into pages, in case of external file we can insert element **<script>** into element **<head>** or **<body>**. An important attribute is **src**, in which we specify where the file is saved and how it is named.

For example:

```
<script src="my_script.js"></script>
```

If the file with JavaScript is available on a web address, it is possible to load a file from the web by stating a correct URL address into attribute src, for example:

```
<script src="https://www.my_page.com/js/my.js"></script>
```

📝 1.1.9

Which parameter is used for inserting Javascript link in an element **<script>**?

- src
- link
- script
- js
- href

📝 1.1.10

In HTML code insert a link into element **<body>** on Javascript stored in a file **test.js**

```
<!doctype html>
<html>
 <head>
    <title>First JavaScript</title>
 </head>
 <body>
    <script src="_____"></script>
```

```
 </body>
</html>
```

📝 1.1.11

In HTML code insert a link into element **<body>** on Javascript stored in a url address http://www.mypage.eu/js/test.js

```
<!doctype html>
<html>
 <head>
    <title>First JavaScript</title>
 </head>
 <body>
    <script _____="http://www.mypage.eu/js/test.js"></script>
 </body>
</html>
```

1.2 Outputs and comments

🕮 1.2.1

The first step with the coding language JavaScript is to add comments. Comments are commonly used by programmers, being part of the code, even though the computer (JavaScript interpreter) ignores them. Their purpose is to save comments and notes for programmers.

By using comments, the code is more efficient, when you return to rewrite it. Programming is in larger projects a cooperative activity, and that is the reason why commenting in source codes is compulsory for larger teams.

A comment can explain ideas of program creator, backup instructions for other developers inside of the code or add other important notes.

🕮 1.2.2

There are two types of JavaScipt comments:

1. A one-lined code. It is designated with characters **//**, behind which is the comment text, for example:

```
// my comment starting at the beginning of the line
```

The end of the one-lined comment isn't designated, the end of the line is meant to be the end

```
function my_f(input) // my second comment
```

2. A multiline comment. It usually includes large pieces of the text, probably for multiple lines. This type of comment is inserted between characters /* and */.

Example:

```
/*
This whole text
will be
understood by the computer as a comment
*/
```

We can even use this type of commenting inside the line of the code, for example:

```
function my_f(/* can be even zero*/ input) //my own function
```

1.2.3

Insert into Javascript code one-liner code comment with a text: I am excited about Javascript.

```
<!doctype html>
<html>
 <head>
    <title>First JavaScript</title>
 </head>
 <body>
   <script>
      _____ I am excited about Javascript
  </script>
 </body>
</html>
```

1.2.4

Insert into Javascript code a multiline comment of 3 lines beginning with "Lecture 1" and ending with "I am excited, what is next."

```
<!doctype html>
<html>
 <head>
    <title>First JavaScript</title>
 </head>
 <body>
   <script>

     _____
      Lecture 1:
      Comments
      I am excited, what is next.

     _____
  </script>
 </body>
</html>
```

1.2.5

Insert correct tags for comment into Javascript code:

1. Mark the text beginning with „If you were“ and ending with “Then they would exclaim:” into a multiline comment.
2. Insert the sentence „Oh, what a pretty house that is!” into a one-liner comment.

```
<!doctype html>
<html>
 <head>
    <title>First JavaScript</title>
 </head>
 <body>
   <script>

     _____
      If you were to say to the grown-ups:

      "I saw a beautiful house made of rosy brick, with
geraniums in the windows and doves on the roof," they would
not be able to get any idea of that house at all.
      You would have to say to them:

     "I saw a house that cost $20,000."
     Then they would exclaim:

     _____
```

```
    _____ "Oh, what a pretty house that is!"

  </script>
 </body>
</html>
```

📖 1.2.6

The first function we will use is **alert()**. The function shows alerts with a message (text) and a button OK. It is often used as a warning message for the user of a web application.

It isn't recommended to use the function too often, as it prevents the user access to some parts of the web page, while the window with a message will not be closed.

In our course, we will use the function only for controlling the correctness of our code. Therefore most of our scripts will be ended with a function **alert()** for final results output.

The important part of a function **alert()** is a parameter of a function. It is used to address an input value into a function. Input parameter will be a text which we will use in a shown window to the user, for example:

```
alert("Hello world!");
```

Similar to other coding languages, even in JavaScript it is important to end the line or a function (at the end of the line) by inserting a semicolon ; (or line feed). It separates individual commands from each other.

Theoretically, we could write the whole Javascript code into a one-liner. Semicolons are important for the interpreter to identify the end of the command. The division of the code into single lines is only for transparency and the readability of the source code.

📝 1.2.7

The majority of messages are text. If we want to work with a text in JavaScript (we will also use numbers), it is important to insert the text into quotes.

Sometimes apostrophes are used, their purpose will be mentioned later.

Insert the function **alert** into JavaScript code and show a message "First message from programmer!" to the user. Don`t forget the semicolon at the end of the line!

```
<!doctype html>
<html>
 <head>
    <title>First JavaScript</title>
 </head>
 <body>
   <script>
     _____("First message from programmer!")_____
   </script>
 </body>
</html>
```

1.2.8

Insert functions for showing two messages into a JavaScipt code. The first message will include the text "Welcome to my web page" and the second one "Feel free to look around".

Don`t forget semicolons at the end of the line!

```
<!doctype html>
<html>
 <head>
    <title>First JavaScript</title>
 </head>
 <body>
   <script>
       _____("Welcome to my web page")_____
       _____("Feel free to look around")_____
   </script>
 </body>
</html>
```

Variables

Chapter 2

2.1 Variables

📖 2.1.1

Variables are used for saving the information we need. They allow us to manipulate data and use it to get desired results. The simplest is to imagine them as areas within the memory of the computer; each area is a variable that holds information. This data can be moved, copied or even changed during the run time of your program.

It is easier to consider variables as containers that can store any information we want. The data can be used later on in your program.

Declaring and initializing variables in the *JavaScript* language:

```
var x = "the first word";
```

Explaining individual parts:

- *var* is the keyword for initializing a variable. It lets the browser (and the rest of the code) know of its existence, name and any value we assign to it. Creating a variable is called declaration and giving it value is called initialization.
- *x* is the name of the variable,
- *=* is an assignment operator. As the name suggests, it assigns a value to a variable
- *"the first word"* the value stored in our variable.

The variables with text should be put in quotation marks (apostrophes can be used in some cases).

📝 2.1.2

Create a variable **car** in code and assign a value "**MPV**".

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
  </head>
  <body>
     <_____>
        _____ _____="MPV";
     <_____>
  </body>
```

```
</html>
```

- code
- variable
- car
- script
- /code
- /script
- var

2.1.3

The advantage of variables is in case of the need we can edit its value. For a repeatable use of variables, we don`t use keywords **var**. We can view a value of the variable for the user with a function **alert()**, in which the parameter sets the name of the variable.

Assign into a variable **car** new value "**SUV**".

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
  </head>
  <body>
     <script>
        var car = "MPV";
        _____ = "_____";
        alert(car);
     </script>
  </body>
</html>
```

2.1.4

We can set a value of a variable into another variable. Example:

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
  </head>
  <body>
```

```
    <script>
       var car = "Cabrio";
       var my_car = car;
       alert(my_car);
    </script>
  </body>
</html>
```

📖 2.1.5

Linking variables with a text context are realized with the operator **+**. In the next example, we will see the linking of three input variables with the text context into one output variable which we will be shown to the user with a function **alert()**.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
  </head>
  <body>
     <script>
       var part1 = "Good";
       var part2 = " ";
       var part3 = "morning";
       var together = part1 + part2 + part3;
       alert(together);
    </script>
  </body>
 </html>
```

📝 2.1.6

Which operator is used for linking text strings?

- + (plus)
- . (dot)
- , (comma)
- - (dash)

2.1.7

From the variables **x**, **y** and **z** we can create the sentence "**I can't wait what's next**". Save the sentence into a variable **result** which will be shown to the user by a function **alert()**.

It is also needed to realize that we don't need to insert the text with the space, as the text begins with the space in the variable **y** and **z**.

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
  </head>
  <body>
     <script>
          var x = "I can't";
          var y = " wait what's";
          var z = " next";
          var result = "";
          result = _____ + _____ + _____;
          _____(_____)_____
    </script>
  </body>
</html>
```

- y
- warning
- x
- alert
- concate(x,y,z)
- print
- result
- x.y.z
- ;
- z

2.1.8

What will be shown in a window by the function alert()?

```
<!doctype html>
<html>
  <head>
```

```
    <title>JavaScript Example</title>
  </head>
  <body>
    <script>
        var planet1 = "Mercury";
        var planet2 = "Venus";
        var planet3 = "Earth";
        var planet4 = "Mars";
        alert(planet3);
    </script>
  </body>
</html>
```

- Earth
- Mercury
- Venus
- Mars

2.1.9

What will be the output when using the function alert() in the next code?

```
<!doctype html>
<html>
  <head>
    <title>JavaScript Example</title>
  </head>
  <body>
    <script>
        var planet1 = "Mercury";
        var planet2 = "Venus";

        planet1 = "Earth";
        planet2 = "Mars";
        alert(planet1);
    </script>
  </body>
</html>
```

- Earth
- Mercury
- Venus
- Mars

📝 2.1.10

What will be the output of the code using the function alert()?

```
<!doctype html>
<html>
  <head>
     <title>JavaScript Example</title>
  </head>
  <body>
     <script>
         var planet1 = "Mercury";
         // planet1 = "Venus";
         planet1 = "Earth";
         /*
         planet1 = "Mars";
         planet1 = "Jupyter";
         */
         alert(planet1);
    </script>
  </body>
</html>
```

- Earth
- Mercury
- Venus
- Mars

2.2 Numeric variables

🕮 2.2.1

The programming language is not limited to writing simple texts, it can also make calculations. We use numeric variables for this purpose.

```
var a = 10;
var b = 20;
var sum = a + b; // 30
```

So the result of a + b, which is actually 10 + 20, is assigned into the variable *sum*. First, the entire calculation is performed at the right side of the "=" and the result is assigned to the variable after its completion.

- addition (+)

- subtraction (-)
- multiplication (*)
- division (/)

If more than one operation is used in the calculation (commonly referred to as the expression), the standard policy applies: **multiplication and division take precedence over addition and subtraction**. If they are in brackets, the expression in them is evaluated first. If they have the same priority, the calculations are performed from left to right.

📝 2.2.2

What is the output of this part of the program:

```
var a = 10;
var b = 20;
var sum = 2 * a + b;
alert(sum);
```

🕮 2.2.3

Just as we could calculate the value with the output, we can do it with variables:

```
var a = 10;
var b = 5;
alert(a * b); // 50
```

In the output, the calculation is performed first - instead of the variables, the values they contain are put in - and the result obtained is written.

📝 2.2.4

What is the output of this part of the program:

```
var a = 15;
var b = 20;
alert(a + b - 2 * (a - b));
```

📖 2.2.5

The variable may have potentially any name, but we have to follow the following rules:

- the name of the variable must begin with a letter, or "_" (or $, but it is not used)
- other characters may be letters, numbers, or underscores
- names are case sensitive
- the name of the variable must not be either commands or keywords of the language

📝 2.2.6

Which of the following can be used as the variable name:

- my_var
- _cool
- cat
- d_og
- var
- woof-woof
- c 12 3

📖 2.2.7

Numbers offer a special operation that returns the remainder after division. For its calculation is used the operator %.

E.g.:

- 10 % 3 = 1
- 10 % 2 = 0
- 15 % 7 = 1
- 20 % 7 = 6
- 10 % 0 – division by zero = error

📝 2.2.8

What is the output of the following code?

```
var a = 27;
```

```
var b = 4;
var c = a % b;
alert(c);
```

📖 2.2.9

Change of the value of a variable by 1 is done using the incremental and decremental operator that replaces the "long" notation that serves to increase or decrease the value of the variable by 1.

Instead of long notation:

```
a = a + 1;
```

we can use the incremental operator ++:

```
a++;
```

Instead of long notation:

```
b = b - 1;
```

we can use the decremental operator --:

```
b--;
```

📝 2.2.10

What is saved in the variable **a** after the execution of the following commands?

```
var a = 12;
a++;
a = a - 3;
a++;
a--;
```

2.3 Data types

📖 2.3.1

We used data types number and string. Every type has different behaviour.

For:

```
var a = "word1"
var b = "word2"
```

is the result of expression *a + b* concatenation of string: *word1word2.*

For:

```
var a = 30
var b = 20
```

is the result of expression *a + b* sum of numbers: *50.*

If we use a combination of these types and add a number and a string, *JavaScript* will treat the number as a string.

For:

```
var a = 30
var b = "years"
```

is the result of expression *a + b* sum of numbers: *30years.*

📝 2.3.2

What is saved in the variable *c* at the end of the program?

```
var a = 3
var b = "winter"
var c = b + " " + a + " degrees"
```

🕮 2.3.3

The combination of various types of variables brings various behaviour of operations. The operation is executed as a numeric operation if both values are of number type. If one of the values is textual type second value is converted to a string and the operation is executed with string values.

E.g.

```
var a = 10;
var b = 15;
var c = "my result: ";
alert(c + a + b); // my result: 1015
alert(c + (a + b)); // my result: 25
```

In a second case is preferred operation *a + b* where both values are numeric and the "+" is used as numeric addition.

2.3.4

What is saved in the variable d at the end of the program?

```
var a = 3
var b = 7
var c = "value: "
var d = c + (a + b)
```

2.3.5

The type of variable can be changed. It depends on the value that is assigned to the variable, e.g.:

```
var a;          // a is undefined
a = 100;        // a is a Number
a = "Anna";     // a is a String
```

The information about the current variable type is available using *typeof*:

```
var a = 10;
alert(typeof(a)); // number
var b = "hello";
alert(typeof(b)); // string
var c;
alert(typeof(c)); // undefined
```

2.3.6

Pair variables and types of their content.

```
var a = "" + 10;    alert(typeof(a)); // _____
var b = 20 + 40.3;  alert(typeof(b)); // _____
var c = 17;         alert(typeof(c)); // _____
var d = "";         alert(typeof(d)); // _____
var e;              alert(typeof(e)); // _____
```

- number
- number

- string
- number
- undefined
- number
- undefined
- string
- undefined

📖 2.3.7

We are working often also with logic values that can have the value *true* or *false*. Data type *boolean* is used to save this kind of value.

Often it is the result of comparison or evaluation of a condition.

E.g.:

```
var a = 10;
var b = 15;
var c = a > b; // false
c = a < b;     // true
```

If we want to check if the values **are** identical we can use the operator "==".

```
var a = "winter";
var b = "winter";
var c = (a == b); // true
```

If we want to find out if the values **are not** identical we can use the operator "!=".

```
var a = "winter";
var b = "winter";
var c = (a != b); // false
```

📝 2.3.8

Pair variables and their value.

```
var a = 10;
var b = 15;
var c = 20;
var d = "Paris";
var e = "London";
var f = "London";
```

```
var r1 = (a > b);  // _____
var r2 = (a < b);  // _____
var r3 = (a == b); // _____
var r4 = ((a + b) == c); // _____
var r5 = (e == f); // _____
var r6 = (a != b); // _____
var r7 = (d == f); // _____
```

- true
- true
- false
- false
- false
- false
- false
- false
- true
- true
- true
- false
- true
- true

If Statement

Chapter 3

3.1 If statement I.

📖 3.1.1

Like other programming languages, *JavaScript* can use a non-sequential command sequence. When executing a program, it is often necessary to decide which commands the program has to run. The decision is made based on the result of the condition.

The ability to decide and execute other commands based on executing or not executing conditions is referred to as branching. It consists of a condition and commands to be running in the event of execution or not execution of the condition

For example, The program will detect the age of the user. If the user is less than 25 years old, it will say "Hello", if the user is more than 25, it will say "Good morning/afternoon". The condition will be to determine if the user is under 25. The condition must always be evaluable, i.e. we can always determine whether or not it is valid.

In JavaScript, branching is done using the **if** statement. We use this command to determine a part of the code that is executed when the condition is valid. The command looks like this:

```
if (condition) {

  // part of the code, i.e. a command or group of commands to
execute when a condition is valid

 }
```

When using the **if** statement, the following must be observed:

- The condition is always written in brackets - **()**.
- The part of the code to be executed is enclosed in brackets - **{}**.

📝 3.1.2

Fill in the missing parts of the code:

```
if _____condition_____ _____
   // commands
_____
```

-)
- {

- (
- }

📖 3.1.3

The conditions use comparison operators for comparing values and/or variables:

- **>** - is bigger, e.g. **a > b**
- **>=** - is bigger or equal, e.g. **a >= b**
- **<** - is less, e.g. **a < b**
- **<=** - is less or equal, e.g. **a <= b**
- **==** - is equal, e.g. **a == b**
- **!=** - is not equal, e.g. **a != b**

Using symbols in the wrong order will cause an error (e.g.: **=>**, or **<>**).

📝 3.1.4

Choose the correct comparison operators:

- ==
- <|=
- >=
- !=
- <|
- >
- =
- =>
- =<|
- !

📝 3.1.5

Enter the correct characters for comparison in the code:

```
input = -5;
if (a _____ 0){
   alert("zero value");
}

if (a _____ 0) {
```

```
   alert("the number is greater than or equal to zero");
}

if (a _____ 0) {
   alert("the number is negative");
}
```

📖 3.1.6

The first example of the program is about identification, whether the user with the input age is a child.

```
var age = 16;
if (age < 18) {
   alert("This is a child");
}
```

The program writes the information if the value of the variable **age** is less than **18**, but in the opposite case, i.e. if the value of **age** is **18** or older, we will not receive any information.

To handle the situation, i.e. to specify the commands to be executed when the condition does not apply, is used **else** statement. This specifies one or a group of commands to be executed if the condition is not valid.

The general registration shall take the form of:

```
if (condition) {
   // commands when the condition is valid;
} else {
   // commands when the condition is NOT valid;
}
```

Our initial program would have the form of:

```
var age = 16;
if (age < 18) {
   alert("This is a child");
} else {
   alert("This is an adult");
}
```

3.1.7

What pair of commands is used to execute so-called full branching?

- if – else
- if – then
- then – else
- condition – then

3.1.8

Complete the program, which executes the number in the variable *a* whether it is a positive or non-positive number:

```
var a = -5;
if _____a > __________ {
   alert("positive");
 } _____ _____
   alert("non-positive");
}
```

- >
- 1
- }
-)
- else
- <|
- (
- 0
- {
- {

3.1.9

The command groups that are executed when a condition is valid or not are called **branches**.

The positive branch is represented by the commands in the section that is executed when the condition is valid. The negative branch consists of the commands located in the branch after the else command.

In the following program, we will ensure that if the wage is lower than the average, its value increases by 100 and we report the increase.

```
var average = 1010;
var wage = 950;
if (wage < average) {
   wage = wage + 100;
   alert("The wage was increased");
}
alert("Your wage this month: " + wage);
```

3.1.10

Complete the program that will write the absolute value of the number in the variable *input*. Fill in to indicate whether the input was a positive or negative number.

```
var input = -5;
if (_____ < _____){
   alert("_____");
   input = -input;
} _____ {
   alert("_____");
}

alert("absolute value: " + _____);
```

- positive
- -input
- 1
- input
- esle
- input
- input
- else
- 0
- negative

3.1.11

Complete a program that greets the user according to his/her age. If the user is less than 25 years old, he will say hello, if more, he will say Good morning/afternoon.

```
_____ age = 19;
if(age _____ 25){
```

```
     alert("Hello");
} _____ {
     alert("Good morning/afternoon");
}
```

3.1.12

What is the value is written by command **alert()** after execution of commands:

```
var a = 10;
if ( a < 0 ){
   a = a + 1;
} else {
   a = a - 1;
}
alert(a);
```

- 9
- 10
- 11
- 0

3.2 If statement II.

3.2.1

In the previous examples, we always put each branch in brackets {}. If there is only one command in the branch, parentheses may not be used. However, this is not the case in real programs.

The following entry can be used for one command per branch:

```
var age = 19;
if (age < 25)
     alert("Hi");
else
     alert("Hello");
```

This is identical to the following code:

```
var age = 19;
if (age < 25) {
     alert("Hi");
```

```
} else {
     alert("Hello");
}
```

3.2.2

Fill the gaps in the program to compare two numbers and show the larger one.

```
_____ a = 10;
_____ b = 15;
if _____a > b_____
   _____(a);
else
   _____(b);
```

- int
- print
- }
- alert
- else
- alert
- var
- (
-)
- int
- print
- var
- {

3.2.3

The result of the comparison can be used also in conditions so that we will get the result of the expression and then use it in condition, e.g.

```
var a = 10;
var b = 5;
var res = a == b;
if (res == true)
  alert("Values are equal");
else
  alert("Values are different");
```

Notation

```
if (res == true)
```

can be usually written following

```
if (res)
```

because the result of the condition **res == true** is dependent on the value of the variable **res**.

📝 3.2.4

Declare the variable **x** to save the result of the comparison of variable **y** and the value 5 for equality.

```
_____ x;
var y = 7;
x = y _____ 5;
```

🕮 3.2.5

Many times there are tasks where we have to decide whether the given value is even or odd.

When searching for a solution, we can use the fact that even numbers divided by 2 give the remainder after division 0 and odd numbers give 1.

E.g.:

- 20 % 2 = 0 – is even
- 15 % 2 = 1 – is odd

📝 3.2.6

Fill the gaps in the code to identify if the number on input is even or odd.

```
var a = 12;
if (a _____ 2 == 0) {
  alert("_____");
} _____ {
  alert("_____");
}
```

- else
- **
- odd
- -
- even
- /
- %

📖 3.2.7

Often you combine in codes many conditions that can be in different relations. Mostly we are in the following situations:

- all of the conditions have to be met at the same time,
- it is enough that only one of the conditions is met.

Based on the given age of the employee decide whether he/she is in productive age - between 18 and 70 years old.

The task can be solved following:

```
var age = 22;
if (age >= 18) // first condition is met
 // we verify whether the age is also less then the upper
boundary
 if (age <= 70) // both of the conditions are met
   alert("he/she is in productive age");
```

A simple notation makes it possible to write both notations into one complex condition. We use a logical connector AND (we use & in Java) to secure that both conditions have to be met at the same time.

```
if ((age >= 18) && (age <= 70))
```

We put into the brackets each condition as well as the whole expression.

📝 3.2.8

Fill in the expression so that it is true if both conditions are met at the same time:

```
var month = 7;
if ((month >= 6_____ _____ (month <= 9_____
 alert("summer");
```

📖 3.2.9

In some cases, it is necessary that only one condition needs to be met. In that case, is used logical connector OR written using the symbol ||.

```
if ((a>0) || (b<0))
```

Evaluation of the expression is true if at least one of the conditions is met, i.e. it is enough if a > 0 or b < 0.

If both conditions are met, the expression is also true.

> Except the || operator can be used the alternative operator |. Between the | and || operators is the difference that || will end the evaluation of the logical expression in the moment it finds out that the condition is true and the following evaluation does not have effect on the result, where | evaluates till the end.
>
> The same rules apply to & and &&.

📝 3.2.10

Fill in the code so that it prints whether the time (defined in hours 0-23) corresponds to day or night. The day is from 6 till 18.

```
var time = 16;
if (time _____ 6) _____ (time _____ 18))
 alert("_____")

_____
 alert("_____")
```

- <=
- >=
- night
- day
- ||
- else

Loops

Chapter 4

4.1 For loops

📖 4.1.1

If we need to repeat a command or sequence of commands multiple times, we can use a **loop**. For example, to display a 5-times message with the text "Hello" it will look like this:

```
alert("Hello");
alert("Hello");
alert("Hello");
alert("Hello");
alert("Hello");
```

But a much more logical option is to do this through a loop:

```
for(var i = 1; i <= 5; i++){
  alert("Hello");
}
```

📖 4.1.2

The loop is defined by the **for** statement as follows:

```
for(var i = 1; i <= 10; i = i + 1) {  // loop control
  command; // loop body, list of commands to be executed
}
```

The individual parts of the **for** loop are as follows:

- *for(...)* – a loop statement defining a loop with a known number of repetitions
- *i* – control variable
- *i = 1* – set the initial value for the variable
- *i <= 10* – condition until which the loop will be executed; as long as the condition is true, the statement is executed in the body of the loop; if the condition is not true, the cycle ends
- *i = i + 1* – loop step, after executing the body of the loop, the value of the control variable will always change according to the entry, i.e. it is incremented by 1. Typically, the replacement is shorter: i++.

The run of the loop is controlled by an integer control variable, which is initially set to the default value and changes at each step of the loop. If the condition of a loop is not valid as a result of a variable change, execution of commands after the cycle is continued.

4.1.3

Complete the program so that the text "Hello" is displayed 3 times.

```
_____(var i = 1; i <= _____; i++){
   alert("Hello");
}
```

4.1.4

The value contained in the control variable can be used in the loop for any operation.

As an example, we use a program for listing numbers from 1 to 10. We also used the control variable **i** in the body of the loop and included it in the variable **output**, which we report using the **alert()** function at the end of the loop.

```
var output = "";
for(var i = 1; i <= 10; i++){
  output = output + i + ",";
}
alert(output);
```

As you execute the loop, each step adds a value to the text variable that contains **i**, followed by a comma.

For individual **i** the content of the variable **output** will be as follows:

- 1 - 1, - to the empty variable output is added the value 1 and a comma
- 2 - 1,2, - to the content of the variable output (1,) is added the content **i**, i.e. 2 and comma
- 3 - 1,2,3, - to the content of the variable output (1,2,) is added the content **i**, i.e. 3 and comma
- 4 - 1,2,3,4 - to the content of the variable output (1,2,3) is added the content **i**, i.e. 4 and comma
- ...
- 10 - 1,2,3,4,5,6,7,8,9,10 and it ends.

4.1.5

Fill in the correct values and variables in the program so that the program lists numbers from 5 to 15.

```
var output = "";
```

```
for(var i = _____; i <= _____; i++){
    output = output + _____ + ", ";
}
alert(output);
```

4.1.6

How many times the following loop puts the word "Hello" in the variable **output***?*

```
var output = "";
for(var i = 1; i < 5; i++){
    output = output + "Hello, ";
}
alert(output);
```

- 4 times
- 5 times
- 0 times
- 6 times

4.1.7

Although most loops use a change of 1 (i.e. i ++ or i = i + 1) to change the value of the control variable, there are no limitations on its ability to change it. You can decrease the value, change it by another value, multiply it, etc.

Here are two solutions for listing all even numbers from 2 to 100.

Increasing the control variable by 2.

```
var output = "";
for(var i = 2; i <= 100; i = i + 2){
  output = output + i + ", ";
}
alert(output);
```

Multiplying the control variable by 2.

```
var output = "";
for(var i = 1; i <= 50; i = i + 1){
  output = output + (i * 2) + ", ";
}
alert(output);
```

... and other variations we could find more.

4.1.8

Complete parts of the *for* loop so that the script lists all powers of two from 2 to 1024.

```
var output = "";
for(var i = 2; i _____ 1024; i = i _____ 2){
   output = output + i + ", ";
}
alert(output);
```

4.1.9

Complete parts of the *for* loop so that the script prints numbers from 10 to 1 from largest to smallest (i.e. 10, 9, 8,..., 2,1).

```
var output = "";
for(var i = _____; i >= 1; i = i _____ 1){
   output =output + i + ", ";
}
alert(output);
```

4.1.10

What does the following code print on the screen?

```
var output = "";
for(var i = 1; i < 3 ; i = i + 1){
   output = output + i + ", ";
}
alert(output);
```

- numbers 1 and 2
- numbers 1, 2, and 3
- nothing
- numbers 2 and 3
- number 2

4.2 While and do-while loops

📖 4.2.1

Sometimes we do not know how many times the loop will have to be repeated, but we can determine the condition by which the loop should be repeated. E.g.: while you are hungry, eat a cupcake.

In this case, the execution of the loop can be ensured through the while statement and the condition that the execution of the statements in the body of the loop will be executed.

The format of a while loop is:

```
while (condition) {
 commands;
}
```

The condition must be enclosed in parentheses.

📝 4.2.2

Complete the parts of the loop with a start condition:

```
_____ _____condition_____ _____
  // commands;
_____
```

- for
- do
- {
- while
-)
- (
- if
- }

📖 4.2.3

The **while** loop will execute the commands defined in the loop body while the defined condition is evaluated to true.

For example:

```
var weight = 80;
while (weight  < 100){
   alert("Eat!");
   weight = weight + 5;
}
alert("Ooh, I gained weight");
```

... increases the value of the variable **weight** (and prints "Eat!") while weight is less than 100.

📝 4.2.4

What will be in the variable *i* after the loop ends?

```
var i = 5;
while (i < 100) {
 i = i * 2;
}
```

🕮 4.2.5

The **while** loop can solve similar problems to the **for** loop and can do the same as the **for** loop if properly written.

In this case, it displays values 1-10.

```
var i = 1;
var output = '';
while (i <= 10) {
   output = output + i + ", ";
   i++;
}
alert(output);
```

> If you forget to change (increase) the variable even used in the state, the loop will never end. This will cause the browser to crash.

📝 4.2.6

What does the following script do?

```
var i = 1;
```

```
while (i > 10) {
 alert("Hello")
 i++;
}
```

- Nothing
- Prints "Hello" 10 times
- Prints "Hello" 9 times
- Prints "Hello" once
- Prints "Hello" 11 times

📝 4.2.7

Write a script that prints the sum of numbers from 1 to 20.

```
sum = 0;
number = 1;
while (_____ <= _____) {
 sum = _____ + number;
 number_____;
}
alert("Sum of the first 20 numbers are: " + sum);
```

- 20
- sum
- ++
- +
- number
- 19
- 21
- number

📖 4.2.8

The **do-while** loop executes a block of code and then checks whether the condition is valid. Then, the code block executes the repetition as long as the condition is valid.

This kind of loop does the activity as long as the condition is valid - but in the order that it first does, then checks e.g. Eat the cake while you're hungry.

The form is:

```
do {
 command1;
  ...
 commandn;
} while (condition);
```

e.g.:

```
int i = 1;
do {
 alert(i);
 i = i + 1;
} while (i < 5);
```

The main difference from other loops is that the commands in the body of the loop are executed at least once. Only after their first execution, it is checked whether the repetition should be continued.

4.2.9

Complete the program so that it prints even numbers between 1-33.

```
var number = 1;
var end = 33;
_____ {
  if (number _____ 2 == _____)
    alert(number);
  number++;
} _____ (number <= end);
```

- %
- do
- -1
- /
- 1
- 0
- while
- if
- for

📖 4.2.10

Ensure that the 5 litre pot is filled with 1 litre jars until full.

```
var number_of_jars = 0;
do {
   alert("Pour!");
   number_of_jars++;
} while(number_of_jars < 5)
alert("Full pot");
```

📝 4.2.11

Complete the program so that the do-while loop prints numbers from 1 to 50

```
var output = '';
var i = 1;
_____{
   output = output  + i + ',';
   i++;
}
_____(i <= _____)
alert(output);
```

4.3 Loops (programs)

📝 4.3.1

Fill in the code to show the values from 5 to 9:

```
for( i = _____ ; i < _____; i = i _____ 1) {
  alert(i);
}
```

📝 4.3.2

Make sure that the loop terminates:

```
for(var i = 10 ; i >= 5; i = i _____ 1) {
 alert(i);
}
```

4.3.3

Calculate the sum of the first 1000 positive numbers.

```
var sum = 0;
for(_____ i = _____; i <= _____; i_____) {
   sum = _____ + i;
}
alert(sum);
```

4.3.4

Calculate the product between numbers stored in variables **a** and **b** where **a < b**.

```
var a = 10;
var b = 15;
var prod = _____;
for(var i = _____; i >= _____; i--) {
   prod = _____ * i;
}
alert(prod);
```

4.3.5

Fill in the code so 8 dots are printed:

```
var i = 4;
var result = "";
_____ (i <= _____) {
 result = result + ".";
 i = i + 1;
}
alert(result);
```

4.3.6

Fill the gaps in the program to get the number of divisors of the number stored in variable *num*.

```
var num = 12;
var count = _____;
for(var i = 1; i <= _____; i++) {
 if (num _____ i _____ 0)
```

```
   count_____;
}
alert(count);
```

- %
- !=
- count
- /
- num
- 0
- i
- ==
- ++
- --
- 1
- -1

4.3.7

Fill the gaps in the program to identify if the number stored in the variable *num* is a prime number.

```
var num = 47;
var count = 0;
for(var i = 1; i <= num; i++) {
 if (num _____ i _____ 0)
   count++;
}
if (count == _____)
  alert("is prime number");
else
  alert("is not prime number");
```

String

Chapter 5

5.1 String type

📖 5.1.1

A string is a basic data type in *JavaScript*. It consists of series of characters like "I am hungry". In addition to the ability to keep text, it provides options for browsing content, retrieving part of the stored text, counting characters, and more. Strings are written with quotes and you can use both: single or double-quotes.

The most simple operation is getting the number of characters of the saved content. We get it using the **length** method.

The method is separated from the name using the dot ".":

```
var data = "I am hungry";
var count = data.length;
alert(count);
```

Into the variable **count** is saved the number of characters that are contained in the variable **data**, i.e. it's 11.

📝 5.1.2

What is the result of the following code? What value is stored in the variable *len*?

```
var data = "I want dog, cat and car.";
var len = data.length;
alert(len);
```

📝 5.1.3

Complete the code below:

```
var _____ = "Jan";
alert("Your name consists of " + name._____ + " chars.");
```

📖 5.1.4

The string consists of characters. Each character has its place in the string that is defined by the index. *JavaScript* indexes start from zero.

The first character in the string is on position 0, the second is on position 1, etc. The last character is placed on the position decreased by one from the whole count of characters in the string.

E.g. for:

```
var data = "Superman";
```

are characters placed on each position following:

```
0 - S
1 - u
2 - p
3 - e
4 - r
5 - m
6 - a
7 - n
```

📝 5.1.5

Type the character on position 7 in string defined as:

```
var data = "It is cold here."
```

🕮 5.1.6

If we want to read a specific character on specific position, we use the following code:

```
var data = "It is cold here.";
var character = data[7];
```

The variable ***character*** cointains 'o'.

The position of the character we want to get is written to [].

📝 5.1.7

Type the content stored in the variable **result** after the end of the next part of the program:

```
var data = "Alphabet in computer."
```

```
var result = data[3] + data[5] + data[8] + data[13] +
data[15];
```

📖 5.1.8

If we want to compare strings, the character 'A' is not the same as the character 'a'.

Therefore, when writing the code we must be careful and write code for both situations:

Check if the first letters of strings in variable ***a*** is 'h'.

```
var a = "Hello.";
if (a[0] == 'h' || a[0] == 'H')
  alert("yes, it is");
else
  alert("no, it is not");
```

Alternatively, you can use transformation to convert text to uppercase or lowercase.

The notation takes the form:

```
var a = "Hello.";
var low_a = a.toLowerCase(); // hello.
```

If we want to edit only conditions, we can use:

```
var a = "Hello.";
if (a[0].toLowerCase() == 'h') // one character is converted
  alert("yes, it is");
else
  alert("no, it is not");
```

> Alternative **toLowerCase()** is t**oUpperCase()** working under the same rules.

📝 5.1.9

Fill the gap in code to check if the first characters in variables *a* and *b* are the same.

```
var a = "Hello.";
var b = "hello.";
if (a[_____]._____ _____ b[_____]._____)
  alert("the same");
else
```

```
 alert("different");
```

- ==
- 1
- toLowerCase()
- 1
- !<|>
- =
- toLowerCase()
- toUpperCase()
- 0
- 0

5.2 Substring

🕮 5.2.1

We often need to get from the string not only one character but a substring. To obtain the part of the string is used:

- *substr* method, where is defined the beginning position and count of characters after this position,
- *substring* method, where it defines the beginning position and the ending position of the substring. The character chosen at the ending position is not counted to the substring. The method takes into account the characters from the beginning position to the character before the ending position:

```
var str = "Sagarmatha";
var res = str.substr(2, 5); // garma 2,3,4,5,6 (5 chars)
alert(res);
var res = str.substring(2, 5); // gar 2,3,4 (chars 2 to 5-
1=4)
alert(res);
```

The substring method has also a second form. In the case when we input only one parameter it will return a substring from the given position till the end of the string.

```
var str = "My long string";
alert(str.substring(8)) // string
```

5.2.2

What is the result of the following code? What is the value of the variable *res*?

```
var str = "New York City";
var res = str.substring(4, 6);
```

5.2.3

What is the result of the following code? What is the value of the variable *res*?

```
var str = "Don Quijote de la Mancha";
var res = str.substring(12);
```

5.2.4

Fill the gaps to get the first and last character of content stored in the *str* variable.

```
var str = "My long string";
var first = str[_____];
var last = str[str._____ - _____];
alert(first + last); // Mg
```

5.2.5

The occurrence of the substring in the existing string is verified by the *indexOf()* method and returns the position where the substring is placed.

```
var text = "Jan Amos Comenius";
var pos = text.indexOf("Amos");
```

The variable *pos* will contain the value 4 because the 4th position was first found at the beginning of the searched substring.

In case that the searched substring is not found in the string, it returns the value -1. This can be used to notify the user.

```
var text = "Jan Amos Comenius";
var pos = text.indexOf("abc");
if (pos == -1)
  alert("Substring was not found.")
else
  alert("Substring beginns at position " + pos + ".");
```

5.2.6

Complete the code to find if the first string contains the second string or the second string contains the first one:

```
var first = "winter is nice";
var second = "nice";
var pos = _____.indexOf(_____);
if (pos > _____)
 alert("first contains second")
else
 alert("first doesn't contains second");

var pos2 = _____.indexOf(_____);
if (pos2 _____ -1)
 alert("second doesn't contains first")
else
 alert("second contains first");
```

- 1
- ==
- -1
- 0
- first
- >=
- first
- second
- second

5.2.7

What is the result of the following code written in alert?

```
var a = "New York";
var b = "or";
alert(b.indexOf(a));
```

5.2.8

Complete the code to find the initials of the name stored in the variable **name**. The initials always start after the space:

```
var name = "don Quijote de la Mancha";
```

```
var initials = _____; // prepare empty inicials
do { // al least one initial must be in name
   initials = initials + name[_____]; // first character is
initial
   var pos = name.indexOf(_____); // separator of names
   if (pos > _____) // if there is some space delete part to
space
     name = name._____(_____); // copy string from char after
space
} _____ (pos > _____) // while was space in name
alert(initials);
```

- substr
- while
- pos
- 1
- -1
- 0
- 0
- substring
- -1
- " "
- pos + 1
- 0
- 1
- pos - 1
- ""

📖 5.2.9

We need to transform a string into a number sometimes. The **parseInt()** function parses a string and returns an integer.

E.g.:

```
var sNum = "100";
var num = parseInt(sNum);
alert(sNum + sNum) ; // 100100
alert(num + num) ; // 200
```

If the first character cannot be converted to a number, **parseInt()** returns **NaN**. If some of next characters cannot be converted parseInt return only a converted part e.g. *parseInt("12a3")* returns *12*.

To check the result of conversion we use **isNan(variable)**.

The reverse conversion can be achieved by adding the numerical value to the string value (e.g. empty string).

```
var num = 10;
var sNum = "" + num;
```

5.2.10

Complete the following program. If the number is stored in the variable, write his double side otherwise, request it to be re-entered.

```
var a = "105";
var num = _____(a);
if (_____(num))
  alert("try again");
else
  alert(num*2);
```

5.3 String (programs)

5.3.1

Complete the code that will create a mirror image of the given text, e.g.:

```
Mother -> rehtoM
winter -> retniw
```

```
var text = "Aladin";
var res = "";
var character = "";
for(var i = 0; i < text._____; i++) {
    character = text.substring(i,_____);
    res = _____ + _____; // the char is put before string
}
alert(res);
```

5.3.2

Complete the following code and find out how many times is the digit 7 repeated in the given string.

```
var text = "676776";
var count = 0;
var character, num;
for(var i = 0; i < text._____; i++) {
   character = text._____(i,_____);
   if (character _____ "7")
     count++;
}
alert(count);
```

5.3.3

Complete the code that returns the count of the digits stored in the string variable.

```
var text = "125a4as0";
var count = _____;
var character, num;
for(var i = 0; i < text.length; i++) {
   character = text.substring(_____,_____);
   num = parseInt(character);
   if (!_____(num))
     count++;
}
alert(count);
```

5.3.4

Complete the code that returns the sum of the digits of the number you entered.

```
Input : 123
Output: 6
```

```
var text = "12548";
var sum = _____;
var character, num;
for(var i = 0; i < text._____; i++) {
    character = text._____(i,i+1);
    num = _____(character);
    sum = sum + _____;
}
alert(sum);
```

- length

- 1
- character
- num
- parseInt
- length()
- int
- substr
- substring
- 0

5.3.5

Complete the code that returns the product of the digits of the number you entered.

```
Input : 123
Output: 6
```

```
var text = "12548";
var prod = _____;
var character, num;
for(var i = 0; i < text._____; i++) {
    character = text._____(i,i+1);
    num = _____(character);
    prod = prod * _____;
}
alert(prod);
```

Arrays

Chapter 6

6.1 Arrays

📖 6.1.1

More than 90 % of applications need for their work lists. The example of lists are people, invoices, cars, measured values, etc.

The most simple list that we have already worked with is a **string** - it contains the list of characters ordered into a string that allows reading, adding, deleting, etc.

The access to specific characters of the list was secured through the index:

```
var str = "Aladin"
str[0] - A
str[1] - l
str[2] - a
str[3] - d
str[4] - i
str[5] - n
```

📝 6.1.2

What is the result of the output of the program?

```
var str = "South America";
alert(str[str.length - 2]);
```

📖 6.1.3

To create lists of data of the same type is used the data type **array**. We can create an array with values using:

```
var cars = ["Audi", "Peugeot", "BMW"];
```

or

```
var cars = new Array("Audi", "Peugeot", "BMW");
```

The access to each element is done using an index where the first value is saved at position 0, e.g.:

```
alert(cars[0]); // Audi
alert(cars[1]); // Peugeot
alert(cars[2]); // BMW
```

6.1.4

What is the result of the output of the program?

```
var cars = ["Audi", "Peugeot", "BMW"];
alert(cars[0]+cars[2]);
```

6.1.5

The last element in the array has the index **numberOfElements - 1**. We can get the number of elements using *array.length*, e.g.

```
var cars = ["Audi", "Peugeot", "BMW"];
alert(cars.length); // 3
```

During the execution of the program can be the value of the element changed following way:

```
cars[0] = "Suzuki";
```

6.1.6

What is the result of the output of the program?

```
var numbers = [1, 2, 3, 8, 9, 7, 5];
for(var i = 0; i < numbers.length; i++) {
  numbers[i] = numbers[i] + i;
}
alert(numbers[3] + numbers[5]);
```

6.1.7

To add a new element into the array we have two ways:

The simplest way is to use *push* to add a new element to the end of the array:

```
var data = [10, 20, 30, 50];
data.push(321); // [10, 20, 30, 50, 321];
```

6.1.8

Fill the gap in the code to add a new element to the end of the array and show the list of elements in alert:

```
var data = ["A", "B", "C", "D", "E"];
data._____("X"); // add new element to the end of array
var res = "";
for(var i = 0; i < data._____; i++) {
  res = res + data[i] + ",";
}
alert(res); // list of elements separated by comma
```

6.1.9

The second way allows setting the i-th element to the new value.

If we use the index immediately following the index of the last element, a new element after the last one will be added.

```
var myInputs = ["A", "B", "C", "D", "E"];
myInputs[5] = "F"; // A B C D E F
```

If we use the index not immediately following the index of the last element, the new element will be added and elements between the last and new element will be set to **undefined**.

```
var myInputs = ["A", "B", "C"];
myInputs[5] = "X"; // A B C undefined undefined X
```

6.1.10

Fill the gap in the code to add a new element after the end of the array to achieve the following result:

```
A, B, C, D, E, undefined, X
```

```
var data = ["A", "B", "C", "D", "E"];
data[_____] = "X";
var res = "";
for(var i = 0; i < data.length; i++) {
  res = res + data[i] + "_____";
}
alert(res);
```

6.2 Arrays processing

📖 6.2.1

The arrays in *JavaScript* can contain different types of values, e.g.:

```
var data = ["Paris", "France", -8000, 105, 2148271];
```

where the data represents the name, country, year of foundation, area and population.

To get data we use a familiar approach:

```
data[0] .. data[4]
```

📝 6.2.2

Fill the gap to achieve a population of Madrid.

```
var data = ["Madrid", "Spain", 900, 605, 3165541];
alert(_____[_____]);
```

📖 6.2.3

A useful method to prepare output with arrays elements is **toString()**. The method converts an array to a string of comma-separated array values.

```
var data = ["Anna", "Ivan", "Juanita", "George", "Dieter"];
alert(data.toString()); // Anna,Ivan,Juanita,George,Dieter
```

The **join()** method creates an output of all array elements, but we can specify the separator too, e.g.:

```
var data = ["Anna", "Ivan", "Juanita", "George", "Dieter"];
alert(data.join(" - ")); // Anna - Ivan - Juanita - George -
Dieter
```

📝 6.2.4

Use methods **toString()** and **join()** to achieve the following outputs:

```
var data = ["A", "B", "C", "D", "E"];
alert(data._____); // A,B,C,D,E
alert(data._____(_____)); // A, B, C, D, E
alert(data._____(_____)); // A , B , C , D , E
alert(data._____(_____)); // A - B - C - D - E
```

- join
- ","
- ","
- tostring
- ","
- toString()
- join
- toString()
- join
- "-"
- toString
- ","

📖 6.2.5

The *pop()* method removes the last element from an array:

```
var data = ["A", "B", "C", "D", "E"];
data.pop(); // A, B, C, D
```

📝 6.2.6

What is the result of the following code? What content is stored in variable **res** after its execution?

```
var data = ["A", "B", "C"];
data.pop();
data.push("X");
data.push("Y");
data.pop();
data.push("Z");
var res = data.join(",")
alert(res);
```

📖 6.2.7

Method **splice()** is used for:

- adding a new element(s) into the array to any position,
- remove existing element(s) from any position.

To add elements is simple:

```
var data = ["A", "B", "C", "D", "E"];
data.splice(3,0,"X","Y"); // A,B,C,X,Y,D,E
```

- The first parameter (3) defines the position where new elements (defined later) will be inserted.
- The second parameter (0) is used for deletion - not interesting for us now.
- Next parameters (any count) defined elements for insertion.

In code, we add new elements starting from position 2.

📝 6.2.8

Update the array to achieve results A, B, X, Y, Z, C, K, L, D, E, F.

```
var data = ["A", "B", "C", "D", "E"];
data._____(_____, 0, "X", "Y", "Z");
data._____(_____, 0, "K", "L");
data._____(_____, 0, "F");
```

📖 6.2.9

Method **splice()** can be used for removing elements:

To remove elements is simple:

```
var data = ["A", "B", "C", "D", "E"];
data.splice(1,2); // A,D,E
```

- The first parameter (1) defines the position where to start with deletion.
- The second parameter (2) defines how many elements we want to delete.
- Parameters for insertion we don't define now.

In the code, we removed 2 elements starting on position 1.

📝 6.2.10

Update the array to achieve results A, B, C, F.

```
var data = ["A", "B", "C", "D", "E", "F"];
data._____(_____, _____);
```

🕮 6.2.11

We can use a combination of removing and inserting (replacing) elements.

In the same splice() command, we can remove elements started at the defined position and later add elements placed from the same position.

The structure of command is as follow:

```
var data = ["A", "B", "C", "D", "E"];
data.splice(1, 2, "X", "Y", "Z"); // A, X, Y, Z, D, E
```

- The first parameter (1) defines the position where to start with deletion.
- The second parameter (2) defines how many elements we want to delete.
- Parameters for insertion are three and they are inserted from position 1 in the array.

In the code, we removed 2 elements and added 3 elements. Both operations started at position 1

📝 6.2.12

Update the array to achieve results A, B, X, Y, F.

```
var data = ["A", "B", "C", "D", "E", "F"];
data.splice(_____, _____, "_____", "_____");
```

6.3 Arrays (programs)

📝 6.3.1

Complete the program that prints the number of occurrences of a given value in the given array.

```
var data = [10, 5, 15, 5, 7, 11, 5];
var el = 5
```

```
var count = _____;
for(var i = 0; i < data._____; i++) {
 if (data[_____] _____ el)
  count_____;
}
alert(count);
```

6.3.2

Complete the code that prints the largest value of the given array.

```
var data = [10, 5, 15, 5, 7, 11, 5];
var max = data[0];
for(var i = _____; i < data._____; i_____) {
  if (data[i] > _____)
    max = data[_____];
}
alert(max);
```

6.3.3

Complete the code that calculates the average value of a given array.

```
var data = [10, 5, 15, 5, 7, 11, 5];
var sum = 0
for(var i = 0; i < data._____; i++) {
  sum = sum _____ data[_____];
}
var avg = sum/data._____;
alert(avg);
```

6.3.4

Complete the code that prints all the array elements divisible by a given value for the given integer array.

```
var data = [6, 24, -8, -12, 21, 7, 4, 4];
var number = 4;
var res = "";
for(var i = 0; i < data._____; i++) {
  if (data[i] _____ number _____ _____)
    res = res + data[i] + ",";
```

```
}
alert(res);
```

6.3.5

Complete the program to create a new array based on the data from the default array. Copy only items containing the specified string.

```
var data = ["Madrid", "Paris", "Stockholm", "London",
"Florida"];
var search = "ri";
var new_array = _____; // empty array definition
for(var i = 0; i < data._____; i++) {
 if (data[i]._____(search) > -1)
    new_array._____(data[i]);
}
alert(new_array.toString());
```

Functions

Chapter 7

7.1 Functions

📖 7.1.1

A function is a fragment of code (a set of statements), which can be **called** by code external to the function. Functions can also be called internally in case of recursion.

A function has a name, body, and parameters.

```
function myFunction(par1, par2) {
   // body of function
}
```

The name identifies the function within an application. Simply defining a function does not execute it; we need to call it. Calling a function actually performs the specified code statements from the function body with the indicated parameters.

```
...
myFunction(1,2)
```

Moreover, each function returns a value. The default return value is *undefined*, but you can change it by using the keyword *return*.

> There are also functions without names, we call them *anonymous functions* and they will be described in the next sections.
>
> In JavaScript, functions are objects (they are prototypes of the *Function* object), which means you can add new properties and methods to them. However, functions can be called, and this is what distinguishes them from other objects. Functions have an important role in JavaScript because they create new variable scopes.

📝 7.1.2

Complete the function structure:

```
_____ _____(_____) _____
 _____
}
```

- functionName
- function
- {
- parameter
- // function body

7.1.3

What does the below function return?

```
function foo() {
    alert('Fitped');
}
```

- undefined
- Fitped
- null

7.1.4

The first way of defining a function is **a function declaration**. The function declaration starts with the *function* keyword, followed by a function's name, a list of parameters, and code statements enclosed in curly brackets *{}*. See the basic code below:

```
function functionName(parameter1, parameter2) {
    statements;
}
```

As you can see, the list of parameters to the function is enclosed in parentheses and separated by commas. Parameters are input data, and we use them to pass values for the function's code. Parameters are also called arguments. Let's start with the declaration of a very basic summation function:

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

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

The above function adds two numbers and returns the result of this summation.

7.1.5

What is the result of this code:

```
function process(a, b) {
    return a * b + 5 - b;
}
```

```
process(10, 2); // ???
```

📖 7.1.6

The *console.log* function is a procedure that allows displaying values in the console.

> The function is available either in every modern browser (use the *F12* key to open it) or the Node.js interpreter.

Functions must be in scope when they are called, but the function declaration can be hoisted, which means you can call a function before its declaration. Note, this works only for *named functions* with functions declarations. See the example below:

```
console.log(sum(1, 2)); // 3
function sum(a, b) {
    return a + b;
}
```

The scope of a function is the function in which it is declared, or the entire program if it is declared at the top level.

📝 7.1.7

Complete the code to output the function result to the console:

```
function process(a, b) {
    return a * b;
}

_____._____(process(5, 8)); // 40
```

📖 7.1.8

The second method of defining a function is **a function expression**. A function is just a value and can be assigned to a variable:

```
let foo = function (a, b) {return a * b};
```

After a function expression has been stored in a variable, the variable can be used as a function. A function expression stores in a variable does not require a name; hence, we call it an anonymous function.

```
console.log(foo(5, 3)); // 15
```

There are several different ways that function expressions become more useful than function declarations:

- using them as closures,
- using them as arguments to other functions,
- using them as *Immediately Invoked Function Expressions* (IIFE).

📝 7.1.9

Complete the function as expression:

```
_____ sum = _____ (a, b) {_____ a + b};
```

- def
- funct
- run
- func
- let
- function
- return

7.2 Function parameters

🕮 7.2.1

Function **parameters** is the list of variable names listed in the function definition, whereas **arguments** are the real values passed to (and received by) the function.

```
function foo(parameter1, parameter2, parameter3) {
    // ...
}
```

A JavaScript function definition does not specify data types for parameters (it can be done in other languages like TypeScript) and hence does not perform type checking on the passed arguments.

JavaScript functions do not even check the number of arguments received.

If a function is called with missing arguments (less than declared), the missing values are set to: *undefined*.

```
function foo(parameter1, parameter2, parameter3) {
    console.log(parameter1);
    console.log(parameter2);
    console.log(parameter3);
}
foo(1, 2); // 1 2 undefined
```

📝 7.2.2

Does it need to specify a data type of parameters in the function header?

- no
- yes

🕮 7.2.3

In the newer version of JavaScript, it is allowed to assign default parameter values in the function declaration:

```
function foo(parameter1, parameter2, parameter3=3) {
    console.log(parameter1);
    console.log(parameter2);
    console.log(parameter3);
}
foo(1, 2); // 1 2 3
```

If a function is called with too many arguments (more than declared), these arguments can be reached using the **arguments** object. The argument object contains an array of the arguments used when the function was called (invoked):

```
function foo() {
    console.log(arguments);
}
foo(1, 2, 3); // [1, 2, 3]
```

📝 7.2.4

Given the following code, what will be the console output when *foo()* is executed?

```
function foo(a, b, c) {
    console.log(a + b + c);
};
foo('a ', 'b ');
```

- a b
- a b undefined
- a b c

7.2.5

What is the name of the special object that contains all values passed to a function?

- arguments
- variables
- parameters

7.2.6

Fill the gaps in the code to achieve output:

```
1
8
2
```

Code:

```
function foo(parameter1_____, parameter2_____,
parameter3_____) {
    console.log(_____);
    console.log(_____);
    console.log(_____);
}
foo(1, 2);
```

-
- parameter2
- parameter1
-
- = 8
- parameter3

📝 7.2.7

Fill the gaps in the following code:

```
_____ foo(a, b=_____) {
    _____ a*b;
}
console.log(foo(5)); // should be 15
```

🕮 7.2.8

Arrow functions are a more concise syntax for writing function expressions. There are several syntaxes available for declaring arrow functions:

```
let foo = (parameter1, parameter2, ..., parameterN) => {
statements }
let foo =  (parameter1, parameter2, ..., parameterN) =>
expression
// equivalent to: => { return expression; }

// Parentheses are optional when there's only one parameter
name:
let foo = (singleParameter) => { statements }
let foo = singleParameter => { statements }

// The parameter list for a function with no parameters should
be written with a pair of parentheses.
let foo = () => { statements }
```

📝 7.2.9

Fill the gaps in the following code:

```
let foo = (a, b) _____ a + b;
console.log(foo(10, 5)); // should be 15
```

Document Object Model (DOM)

Chapter 8

8.1 Introduction to DOM

📖 8.1.1

The **DOM** (*Document Object Model*) is a cross-platform and language-independent interface that is an API for HTML and XML documents. The DOM represents the structure of a document as **a logical tree**, wherein each node is an object representing a part of the document.

Additionally, the DOM is the **programming interface** for events, aborting activities, and accessing the tree, which can be used for changing the structure, style or content of the document.

The DOM model is based on the **W3C DOM standard**, see this link if you want to learn more about it.

The consecutive versions of the DOM specification are called **DOM Levels**. Each new level of the DOM adds or changes specific sets of features.

- The DOM Level 1 defines the core elements of the Document Object Model.
- The DOM Level 2 extends those elements and adds events.
- The DOM Level 3 extends the DOM Level 2 and adds more elements and events.
- The DOM Level 4 was published in 2015. It was a snapshot of the **DOM Living Standard**, which is the current standard.

📝 8.1.2

What is the meaning of DOM?

- Document Object Model
- Developer Object Model
- Development Object of Model
- Development Object Model

📖 8.1.3

DOM is just an interface and not a ready-to-use library. In this section, the implementation of DOM for web browsers will be described. It is known as the HTML DOM (the standard model for HTML documents). It defines:

- the HTML elements as **objects**,
- the **properties** of all HTML elements,

- the **methods** to access all HTML elements,
- the **events** for all HTML elements.

We can say that the HTML DOM is a standard for how to get, change, add, or delete HTML elements. Besides that, there are other standards like the *Core DOM* and *XML DOM*, which are not explained in this course.

📝 8.1.4

Which of the following sentences is correct?

- The DOM was created especially for the CSS technology.
- The DOM is a programming interface.
- The DOM uses a queue structure for the document representation.
- The DOM allows handling events through the API.
- The DOM is a binary description of a web page.

📝 8.1.5

Select the parts defined in DOM

- HTML elements
- properties of HTML elements
- methods to access HTML elements
- events for HTML elements
- links to other web pages
- targets of links defined in the webpages

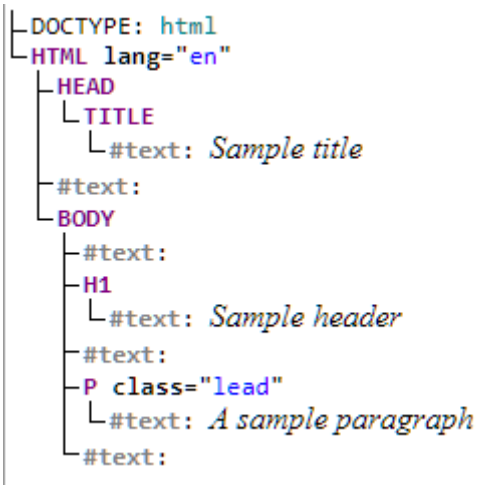
🕮 8.1.6

When a web page is loaded, the browser creates the *Document Object Model* (also known as *a DOM tree*) - the browser needs to know the entire structure before building a tree. See the example of HTML code below:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <h1>Sample header</h1>
        <p class="lead">A sample paragraph</p>
    </body>
```

```
</html>
```

The above code will produce the following DOM model:

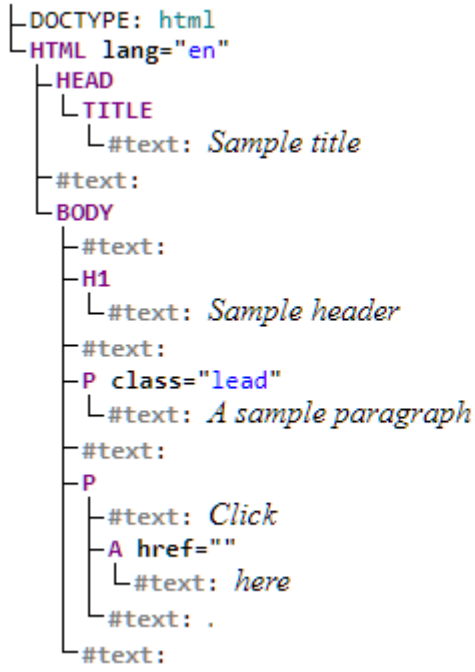


As you can see, the structure consists of all elements from the original HTML code, special #text nodes, which are in fact new lines, tabs and spaces from the source code. You can easily create more examples by using the following tool. The next cards in this section will outline more details of the DOM tree.

8.1.7

Is the following DOM tree correct according to the code?

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <h1>Sample header</h1>
        <p class="lead">A sample paragraph</p>
        <p>Click <a href="">here</a>.</p>
    </body>
</html>
```



- Yes
- No

8.1.8

Is the following DOM tree correct according to the code?

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <h1>Sample header</h1>
        <p class="lead">A sample paragraph</p>
        <p>Click <a href="">here</a>.</p>
        <p class="footer">Copyright &copy; 20xx</p>
    </body>
</html>
```

```
└DOCTYPE: html
└HTML lang="en"
  ├HEAD
  │ └TITLE
  │   └#text: Sample title
  ├#text:
  └BODY
    ├#text:
    ├H1
    │ └#text: Sample header
    ├#text:
    ├P class="lead"
    │ └#text: A sample paragraph
    ├#text:
    ├P
    │ ├#text: Click
    │ ├A href=""
    │ │ └#text: here
    │ └#text: .
    └#text:
```

- No
- Yes

📖 8.1.9

Websites are built based on three technologies: HTML (*HyperText Markup Language*), CSS (*Cascading Style Sheets*) and *JavaScript*. All of them are handled directly by a browser (e.g. Chrome or Firefox).

The implementation of the HTML DOM in web browsers is written in *JavaScript*, so with *JavaScript* code, you can:

- change all the **HTML elements** on the page,
- change all the **HTML attributes** on the page,
- change all the **CSS styles** on the page,
- **remove** existing HTML elements and attributes,
- **add** new HTML elements and attributes,
- **react** to all existing **HTML events** in the page,
- **create** new **HTML events** on the page.

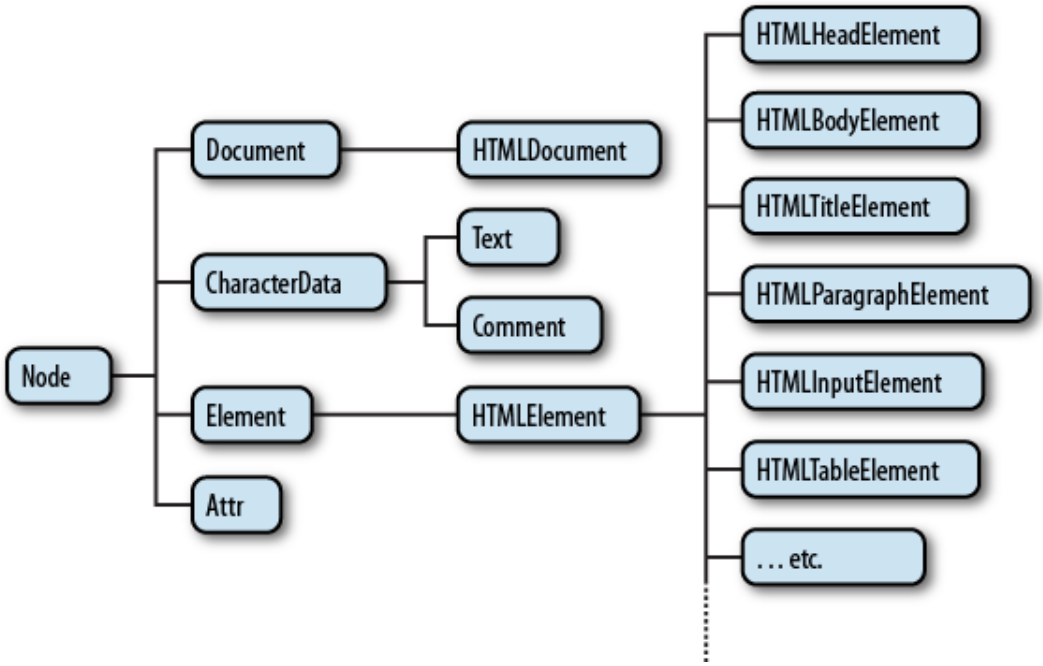
It is assumed that the reader is already familiar with the basics of HTML and CSS.

📖 8.1.10

The HTML DOM tree, wherein all HTML elements are defined as objects, can be accessed through JavaScript.

The DOM programming interface for JavaScript is just a set of properties and methods of each object. Properties of HTML elements are values that you can set or change, whereas methods are actions you can perform on HTML elements.

The object *document* is the *entry point* for accessing the DOM API. In simple terms, the *document* object is the root of the DOM tree. Each node in the tree has its own type that provides properties and methods proper for the element represented by the node. See the below image to learn more about JavaScript's DOM class hierarchy.



There are 12 node types, but in practice, we usually work with 4 of them:

- **Document** (the document object)
- **Element** (HTML elements);
- **Node** (more general type than element);
- **Text/comment**.

8.1.11

What are the most used elements of DOM?

- document
- element
- node
- text/comment

- picture
- common object
- page of web

📖 8.1.12

Look at the below example of using the DOM API. Let's have the following code:

```
<p class="lead">A sample paragraph</p>
```

We can modify the HTML structure with the API:

```
document.querySelector('.lead').id = 'test';
```

The modified version of the code after the execution of the above JavaScript code:

```
<p id="test" class="lead">A sample paragraph</p>
```

In order to use the DOM API, the logical structure of a page needs to be loaded completely. To make sure that your JavaScript code will work properly, put your code before the end of the *<body>* tag or use events (these will be explained later in this course).

You can press the F12 key to open the developer tools and depending on a browser, go to the tab called *HTML/Elements* to check the DOM tree of the current page.

📝 8.1.13

Which key opens the developer console in modern web browsers?

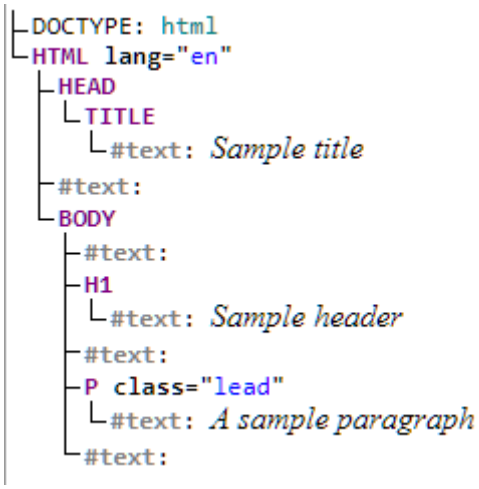
- F12
- F11
- F10

📖 8.1.14

The nodes in the DOM tree have a hierarchical relationship to each other. The terms: parent, child, and sibling are used to describe connections between the nodes. See the example below:

```
<!DOCTYPE html>
<html lang="en">
```

```
    <head><title>Sample title</title></head>
    <body>
        <h1>Sample header</h1>
        <p class="lead">A sample paragraph</p>
    </body>
</html>
```



- <html> is the **root** node.
- <html> has no parents.
- <html> is the **parent** of <head> and <body>.
- <head> is the **first child** of <html>.
- <body> is the **last child** of <html>.
- <head> has one child: <title>.
- <title> has one child (a text node): *Sample title*.
- <body> has two **children**: <h1> and <p>.
- <h1> has one child: *Sample header*.
- <p> has one child: *A sample paragraph*.
- <h1> and <p> are **siblings**.

8.1.15

Fill the gaps to describe the following DOM tree:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <h1>Sample header</h1>
        <p class="lead">A sample paragraph</p>
        <p>Click <a href="">here</a>.</p>
        <p class="footer">Copyright &copy; 20xx</p>
```

```
    </body>
</html>
```

_____ is the root node.

<body> has _____ children.

<h1> has _____ child: Sample header.

<p> is the _____ of <a>.

- parent
- <|head>
- four
- one
- click
- <|/body>
- <html>

8.2 Document properties

📖 8.2.1

To access any element in an HTML page, you always start with accessing the **document object**. The document object provides a set of properties and methods for manipulating and finding objects in the tree.

The **document object** is easily accessible with the global variable document. See the example below:

```
console.log(typeof document); // object
```

The document properties provide access to the tree's elements. For example document.body is a reference to the <body> element.

📝 8.2.2

What part of DOM provides access to the tree's elements?

- document properties
- document content
- document getters

📖 8.2.3

New nodes in the DOM tree (as well as in the structure of the document) can be created with the method *document.createElement({tag name})*.

You should provide the tag`s name that you want to create, as an argument of the method.

```
document.createElement("p"); // creates a <p> element
document.createElement("a"); // creates a <a> element
document.createElement("button"); // creates a <button>
element
```

After the element creation, use the *parent.appendChild({new element})* or *parent.insertBefore({new element}, {existing element})* method to insert it to the document. Let's have the following HTML code:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <h1>My content</h1>
        <script>
            var p = document.createElement("p");
            p.innerHTML = "Some text..."; // insert text
            document.body.appendChild(p);
            var q = document.createElement("p");
            q.innerHTML = "Some second text..."; // insert
text
            document.body.appendChild(q);
        </script>
    </body>
</html>
```

After execution, the document should be as follows:

```
MY CONTENT
Some text...
Some second text...
```

The *parent.appendChild({new element})* method appends a node as the last child of a node.

8.2.4

Order the lines on the web page after script code execution:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <script>
        <script>
            var p = document.createElement("p");
            p.innerHTML = "My first text"; // insert text
            document.body.appendChild(p);
            var q = document.createElement("p");
            q.innerHTML = "My second text"; // insert text
            document.body.appendChild(q);
        </script>
    <p>Prepared content in paragraph</p>
        <script>
            var p = document.createElement("p");
            p.innerHTML = "My third text"; // insert text
            document.body.appendChild(p);
        </script>
        </script>
    </body>
</html>
```

- My third text
- My first text
- My second text
- Prepared content in the paragraph

8.2.5

The following code:

```
var p = document.createElement("p");
p.innerHTML = "Some text...";
document.body.appendChild(p);
document.body.appendChild(p);
```

The above code **will not add** two *<p>* elements.

This is because the second call of the *appendChild* will try to add an element that is already in the tree.

The solution for creating the same object twice is method *element.cloneNode({deep})*.

The code below:

```
var p = document.createElement("p");
p.innerHTML = "Some text..."; // insert text
document.body.appendChild(p);
var clonedP = p.cloneNode(true);
document.body.appendChild(clonedP);
```

The *element.cloneNode({deep})* method creates a copy of a node, and returns the clone.

Set the *deep* parameter value to *true* if you want to clone all children, otherwise *false*.

8.2.6

Order the lines on the web page after script code execution:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <script>
            var p = document.createElement("p");
            p.innerHTML = "Start line with text"; // insert
text
            document.body.appendChild(p);
            var clone = p.cloneNode(true);
            document.body.appendChild(clone);
        </script>
    <p>Prepared content in paragraph</p>
        <script>
            var p = document.createElement("p");
            p.innerHTML = "Last line with text"
            document.body.appendChild(p);
        </script>
    </body>
</html>
```

- Last line with text
- Start line with text
- Prepared content in the paragraph
- Start line with text

📖 8.2.7

The **insertBefore**({new child node}, {subsequent child node}) method inserts the new child node before the subsequent one.

The following code inserts the element to the specific position:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <script>
            var p = document.createElement("p");
            p.innerHTML = "Some text..."; // insert text
            document.body.appendChild(p);
            var a = document.createElement("a");
            a.innerHTML = "Link";
            document.body.insertBefore(a, p);
        </script>
    </body>
</html>
```

After execution, the structure of the document should be as follows:

```
<a>Link</a>
<p>Some text...</p>
```

📝 8.2.8

Order the lines on the web page after script code execution:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <script>
            var a = document.createElement("p");
```

```
            a.innerHTML = "1111111"; // insert text
            document.body.appendChild(a);
            var b = document.createElement("p");
            b.innerHTML = "2222222";
            document.body.insertBefore(b, a);
            var c = document.createElement("p");
            c.innerHTML = "3333333";
            document.body.insertBefore(c, a);
            var d = document.createElement("a");
            d.innerHTML = "4444444";
            document.body.insertBefore(d, b);
        </script>
    </body>
</html>
```

- 4444444
- 2222222
- 1111111
- 3333333

📖 8.2.9

The *parent.appendChild({new element})* method appends a node as the last child of a node.

The *parent.insertBefore({new element}, {existing element})* inserts a new child node before a specified, existing, child node.

The creation of nodes is not limited to HTML elements, you can also create a text node:

```
var h = document.createElement("h1"); // creates a <h1>
element
var t = document.createTextNode("Sample header"); // creates a
text node
h.appendChild(t); // appends the text to <h1>
```

The above code should produce the following HTML structure:

```
<h1>Sample header</h1>
```

8.2.10

Which of the following methods create new objects?

- parent.replaceChild
- document.createElement
- parent.removeChild
- parent.appendChild
- document.createTextNode

8.2.11

Complete the code to achieve this structure:

Main header

text of paragraph

Header

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <script>
        var ha = document._____("h1");
        var ta = document._____("Main header");
        var hb = document._____("h2");
        var tb = document._____("Header");
        var hc = document._____("p");
        var tc = document._____("text of paragraph");
        ha.appendChild(ta);
        hb._____(tb);
        hc._____(tc);
        document.body.appendChild(hc);
        document.body.insertBefore(_____, _____);
        document.body.appendChild(_____);
        </script>
    </body>
</html>
```

- createTextNode

- appendChild
- hc
- createTextNode
- ha
- createElement
- createElement
- hb
- createElement
- createTextNode
- appendChild

📖 8.2.12

Sometimes, you want to add several elements at the same moment. Adding them one by one using *appendChild* is not efficient, because the DOM tree is reloaded after each *appendChild* call. You should use the *document.createDocumentFragment()* method instead.

```
var d = document.createDocumentFragment();
d.appendChild(document.createElement("h1"));
d.appendChild(document.createElement("p"));
document.body.appendChild(d);
```

The *createDocumentFragment* method creates an imaginary node, where you can change, add, or delete, some of the content. These changes do not destroy the document structure, so it can be safer to extract only parts of the document, modify them and insert them back to the document.

📝 8.2.13

Complete the code to achieve this structure using fragment document:

Main header

Header

text of paragraph

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <script>
```

```
        var ha = document.createElement("h1");
        var ta = document.createTextNode("Main header");
        var hb = document.createElement("h2");
        var tb = document.createTextNode("Header");
        var hc = document.createElement("p");
        var tc = document.createTextNode("text of paragraph");
        ha.appendChild(ta);
        hb.appendChild(tb);
        hc.appendChild(tc);
        var frag = document._____(); // create fragment
        frag._____(ha);
        frag._____(hb);
        frag._____(hc);
        document._____._____(frag);
        </script>
    </body>
</html>
```

- appendChild
- createDocumentFragment
- fragment.appendChild
- appendChild
- fragment.appendChild
- fragment.appendChild
- body
- doc.appendChild
- appendChild
- doc.appendChild
- doc.appendChild
- appendFragment
- appendChild

8.3 Accessing elements

🕮 8.3.1

To access elements of the DOM tree, you can use several methods, i.e.:

- finding elements based on a document structure,
- finding elements based on CSS classes, HTML tags, IDs, and names,
- accessing elements based on pre-defined collections.

Based on a web browser and used method, there are 3 possible outcomes:

- a *NodeList* object, which is a list (collection) of nodes (it can contain any type of nodes, like text, elements, and so on);
- an *HTMLCollection* object, which is a list of elements limited to the *Element* type;
- a *Node* or *Element* type object.

All returned collections are sorted as they appear in the source code and can be accessed by index numbers. The index starts at 0.

> It is worth stressing that collections mentioned here are not arrays. You can loop through them and refer to their nodes like an array, but you cannot use methods, like *valueOf()*, *push()*, *pop()* or *join()*.

📝 8.3.2

Is it true?

The elements of the DOM tree can be processed with some array functions. It is possible to achieve them using indexes.

- yes
- no

🕮 8.3.3

Traversing the DOM tree means finding elements based on their relation to other elements. With traversing, you can move up (ancestors), down (descendants) and sideways (siblings) in the DOM tree, starting from the selected (current) element.

Each element has build-in methods that allow accessing related objects. The *childNodes* property returns a collection of a node's child nodes, as a *NodeList* object. See the example below.

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <h1>Sample header</h1>
        <p>A sample paragraph</p>
    </body>
</html>

console.log(document.body.childNodes); // [text (new line and
spaces), h1 element, text, p element, text]
```

A similar method to the *childNodes* is the *children* property, which returns a collection of an element's child elements, as an *HTMLCollection*object, so the outcome of that method will be limited to HTML elements only.

```
console.log(document.body.children); // [h1 element, p
element]
```

We can easily access the first and last child of the specified element.
The *firstChild* property returns the first child node, whereas the*lastChild* property returns the last child node. The *firstChild* and *lastChild* properties return the relevant node as a Node object.

```
console.log(document.body.firstChild); // text
console.log(document.body.lastChild); // text
```

To return HTML elements only, use
the *firstElementChild* and *lastElementChild* property instead.

```
console.log(document.body.firstElementChild); // h1 element
console.log(document.body.lastElementChild); // p element
```

Finally, the *hasChildNodes()* method returns *true* if the specified node has any child nodes, otherwise *false*.

8.3.4

In the following HTML structure:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <h1>My article</h1>
        <p>A sample paragraph</p>
        <h2>Second level</h2>
    </body>
</html>
```

choose the correct answers:

```
console.log(document.body.firstElementChild); // _____
console.log(document.body.lastElementChild); // _____
console.log(document.body.firstChild); // _____
```

```
console.log(document.body.lastChild); // _____
```

- A sample paragraph
- text
- h1
- text
- p
- Second level
- My article
- h2

🕮 8.3.5

Let's have the following HTML structure:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <h1>Sample header</h1>
        <p>A sample paragraph</p>
    </body>
</html>
```

The *nextSibling* property returns the **node** immediately following the specified node, at the same tree level.

```
console.log(document.body.firstChild.nextSibling);
// h1 element, the first child is a text node
```

Respectively, the *nextElementSibling* property returns the **element** immediately following the specified element, in the same tree level.

```
console.log(document.body.firstChild.nextSibling.nextElementSi
bling);
// p element, not a text node
```

Correspondingly, the *previousSibling* property returns the previous **node** of the specified node and the *previousElementSibling* property returns the previous **element** of the specified element, at the same tree level.

```
console.log(document.body.lastChild.previousSibling); // p
element
console.log(document.body.lastChild.previousSibling.previousEl
ementSibling);
```

```
// h1 element
```

In order to access the parent **node** of the specified node, use the *parentNode* property, which returns a *Node* object.

```
console.log(document.body.firstChild.parentNode);
// body
```

The *parentElement* property returns the parent **element** of the specified element.

8.3.6

Let's have the following code:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <h1>Sample header</h1>
        <p class="lead">A sample paragraph</p>
    </body>
</html>
```

Which of the following methods return the *<h1>* element?

- document.firstChild
- document.firstElementChild
- document.firstChild.nextElementSibling
- document.lastChild.previousElementSibling
- document.parentNode

8.3.7

To find elements in the DOM tree, you can use identifiers like a tag name, ID attribute, and class name.

Basic HTML tag structures for these cases are:

```
<tagname id="{id attribute}" class="{class(es)
name(s)}">...</tagname>
<p class="head" id="mainhead"></p>
```

Let's have the following HTML structure:

```
<!DOCTYPE html>
```

```
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <h1 id="main-title">Sample header</h1>
        <p class="lead">A sample lead</p>
        <p>A sample paragraph</p>
    </body>
</html>
```

The method *getElementById({id})* returns the **element** that has the ID attribute with the provided value. The method returns an *Elementobject* that represents an element with the specified ID, *null* if no elements with the specified ID exists.

```
document.getElementById('main-title'); // h1 element
document.getElementById('lead'); // null
```

The method *getElementsByTagName({tag name})* returns a **collection** of all elements in the document with the specified tag name, as a *NodeList* object.

```
document.getElementsByTagName('p'); // [p element, p element]
document.getElementsByTagName('h2'); // [], an empty
collection
```

To find all elements that have a specified class name, use the *getElementsByClassName({class name})* method. It returns a **collection** of elements as a *NodeList* object.

```
document.getElementsByClassName('lead'); // [p element]
document.getElementsByClassName('footer'); // [], an empty
collection
```

The last method is *getElementsByName({name})*, which return a collection of all elements in the document with the specified name (the name attribute).

8.3.8

Let's have the following code:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <p id="xyz" class="abc">Fitped</p>
        <div class="abc">Fitped</div>
```

```
        <p name="xyz" class="abc">Fitped</p>
    </body>
</html>
```

Which of the following methods will you use to get all elements with text "Fitped"?

- document.getElementById()
- document.getElementsByClassName()
- document.getElementsByName()

📖 8.3.9

You can use CSS selectors to find elements in the DOM tree with the methods *querySelector({css selector})* and *querySelectorAll({css selector})*. In general, CSS selectors are patterns used to select elements you want to style (in CSS).

The *querySelector({css selector})* method returns the first element that matches a specified CSS selector(s) in the document, whereas the *querySelectorAll({css selector})* method returns all the matches.

Let's look at the following example:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <h1 id="main-title">Sample header</h1>
        <p class="lead">A sample lead</p>
        <p>A sample paragraph</p>
    </body>
</html>
document.querySelector('p'); // p element
document.querySelector('h1'); // h1 element
document.querySelector('h2'); // null
document.querySelector('#main-title'); // h1 element
document.querySelector('.lead'); // p element
document.querySelectorAll('p'); // [p element, p element]
document.querySelectorAll('h1'); // [h1 element]
document.querySelectorAll('h2'); // []
document.querySelectorAll('#main-title'); // [h1 element]
document.querySelectorAll('.lead'); // [p element]
```

8.3.10

You should use *document.querySelector({css selector})* method to get all elements that match the specified selector.

- False
- True

📖 8.3.11

The last way of accessing elements in the DOM tree is using pre-defined collections, which are properties of the *document* object.

We have already mentioned the *document.body* property, which returns the *<body>* element. Respectively, the *document.head* property returns the *<head>* element.

The *document.documentElement* property returns the *<html>* element and the *document.title* property returns the *<title>* element.

The *document.anchors* property returns all *<a>* elements that have a name attribute.

The *document.forms* property returns all *<form>* elements.

The *document.images* property returns all *<img>* elements.

The *document.links* property returns a collection of all links in the document.

The *document.scripts* property returns all *<script>* elements.

📝 8.3.12

Fill the gap in the following statement:

Use the document._____ property to get access to the <head> element.

8.4 Substitution and elimination of elements

📖 8.4.1

To replace an element with another use the *parent.replaceChild({new element}, {old element})* method. The new node could be an existing node in the document, or you can create a new node.

```
var h = document.createElement("h1");
document.body.appendChild(h);
var p = document.createElement("p");
document.body.replaceChild(p, h); // replaces the existing
element
```

The above code should produce the following HTML structure:

```
<p></p>
```

8.4.2

Complete the code to achieve this replacement of Header 1 by Header 3:

```
<!DOCTYPE html>
<html lang="en">
    <head><title>Sample title</title></head>
    <body>
        <script>
        var ha = document.createElement("h1");
        var ta = document.createTextNode("Main header");
        ha.appendChild(ta);
        var hb = document.createElement("h2");
        var tb = document.createTextNode("Header2");
        hb.appendChild(tb);
        document.body.appendChild(ha);
        document.body.appendChild(hb);
        var hc = document._____("h3");
        var tc = document._____("Header3");
        _____._____(tc);
        _____._____._____(_____, _____);
        </script>
    </body>
</html>
```

- hb
- document
- hc
- appendChild
- hc
- createElement

- body
- hb
- ha
- ha
- hc
- createTextNode
- appendChild
- replaceChild
- createElement

📖 8.4.3

An existing element can be removed from the DOM in two ways:

- the *element.remove()* method,
- the *parent.removeChild({element})* method.

```
var p = document.createElement("p");
document.body.appendChild(p);
p.remove();
document.body.removeChild(p); // does the same as the remove()
method from previous line
```

The *element.remove()* method returns nothing but it removes the object from memory. The *parent.removeChild({element})* method returns the removed node or *null* if the node does not exist, but it does not remove the object.

📝 8.4.4

The *element.remove()* keeps an object in memory.

- False
- True

📝 8.4.5

The *element.removeChild({child})* keeps an object in memory.

- True
- False

Manipulation with Elements

Chapter 9

9.1 Changing element style

📖 9.1.1

To change the style of an HTML element, use the following syntax:

```
element.style.property = new style
```

Where the *property* is one of the CSS properties, e.g. *backgroundColor*. All CSS properties with a dash, like *font-family*, *text-align*, *border-top-width*, etc. are represented in the camel-case form, i.e. *fontFamily*, *textAlign*, *borderTopWidth*, and so on.

📝 9.1.2

Is it possible to change the content or properties of objects placed in DOM?

- yes
- no

📖 9.1.3

Let's have the following document:

```
<!DOCTYPE html>
<html>
    <head><title></title></head>
    <body>
        <p id="par">Fitped</p>
    </body>
</html>
```

The code below should change the color of the paragraph text as well as its font size.

```
document.getElementById('par').style.color = 'red';
document.getElementById('par').style.fontSize = '18px';
```

📝 9.1.4

Let's have the following document:

```
<!DOCTYPE html>
<html>
    <head><title></title></head>
    <body>
        <p id="ppp">Fitped</p>
    </body>
</html>
```

Fill the gaps to change the color of the paragraph text to green.

```
document.getElementById('_____')._____._____ = 'green';
```

📖 9.1.5

The computed style is the style actually used for displaying the element, i.e. the values of CSS properties, which are set based on multiple sources, like internal style sheets, external style sheets, inherited styles and browser default styles.

To get the computed style object (i.e. an object containing the values of all CSS properties of the element) use the *window.getComputedStyle({element})* method.

Let's have the following HTML code:

```
<!DOCTYPE html>
<html>
    <head><title></title></head>
    <body>
        <p style="color: blue;">Fitped</p>
    </body>
</html>
```

The script below should display a value of the *color* property.

```
let style =
window.getComputedStyle(document.querySelector('p'));
console.log(style.color); // blue
```

📝 9.1.6

Fill the gap to set the font color of heading to the same color as paragraph text:

```
<!DOCTYPE html>
<html>
    <head><title></title></head>
```

```
    <body>
        <p id="para" style="color: blue; background-
color:red">Fitped</p>
        <h1 id="main-title">Heading1</h1>
    <script>
       let _style =
window.getComputedStyle(document._____('p'));
       hd = document._____('main-title');
       hd.innerText = 'Changed heading';
       hd.style._____ = _____.color;
    </script>
    </body>
</html>
```

- color
- color
- style
- backgroud-color
- getElementById
- getStyle
- _style
- querySelector
- getElementByTag

📖 9.1.7

The second method of changing HTML style is modifying the value of the element's *class* attribute.

To change a class, the *className* property can be used. The property sets or returns the class name of an element (the value of an element's class attribute).

```
<p class="lead">Fitped</p>
<h1 class="main-header black-version">Fitped</h1>
console.log(document.querySelector('p').className); // lead
console.log(document.querySelector('h1').className); // main-
header black-version
document.querySelector('h1').className = 'second-header';
console.log(document.querySelector('h1').className); //
second-header
```

9.1.8

Which of the following piece of code is the correct way of changing style?

- document.getElementById("a").style.backgroundColor = 'red';
- document.getElementById("a").style.background-color = 'red';
- document.getElementById("a").css.backgroundColor = 'red';
- document.getElementById("a").styleSheet.backgroundColor = 'red'

9.1.9

In order to get the computed style object, you should use the *window.____({element})* method.

9.1.10

Which of the following piece of code is correct?

- document.getElementById("a").classname
- document.getElementById("a").class-name
- document.getElementById("a").className
- document.getElementById("a").classNames

9.1.11

Similar to *className* is the *classList* property. This property is useful to add, remove and toggle CSS classes on an element.

```
<h1 class="main-header black-version">Fitped</h1>
```

To get the number of CSS classes of an element's *class* attribute, use the *element.classList.length* property.

```
console.log(document.querySelector('h1').classList.length); // 2
```

To add a new class to a specified element, use the *element.classList.add({class1}, {class2}, ...)* method.

```
document.querySelector('h1').classList.add('c1');
document.querySelector('h1').classList.add('c2', 'c3');
```

```
console.log(document.querySelector('h1').className); // main-
header black-version c1 c2 c3
```

The *element.classList.contains({class})* method returns a boolean value, indicating whether an element has the specified class name or not.

```
console.log(document.querySelector('h1').contains('c1')); //
true
console.log(document.querySelector('h1').contains('c4')); //
false
```

The *element.classList.item({index})* method returns the class name with a specified index number from an element. Index starts at 0. Returns *null* if the index is out of range.

```
console.log(document.querySelector('h1').item(0)); // main-
header
console.log(document.querySelector('h1').item(1)); // black-
version
console.log(document.querySelector('h1').item(5)); // null
```

The *element.classList.toggle({class})* method toggles between a class name for an element.

```
console.log(document.querySelector('h1').toggle('c2')); //
false
console.log(document.querySelector('h1').className); // main-
header black-version c1 c3
console.log(document.querySelector('h1').toggle('c2')); //
true
console.log(document.querySelector('h1').className); // main-
header black-version c1 c3 c2
```

9.1.12

Let's have the following HTML and JS code:

```
<p class="a b c"></p>
document.querySelector('p').classList.add('d');
```

After the execution, the class name of the *<p>* will be equal to:

- a b c
- a d b c
- a b d c

- a b c d

9.1.13

Let's have the following HTML and JS code:

```
<p class="a b c"></p>
document.querySelector('p').classList.toggle('b');
```

After the execution, the class name of the *<p>* will be equal to:

- a c
- a c b
- a b c
- a

9.2 innerHTML

9.2.1

Property *innerHTML* is used for changing the HTML structure dynamically. It can be used with every HTML DOM element.

In order to get a string that represents the HTML content of an element use the following syntax:

```
element.innerHTML
```

Set the *innerHTML* property:

```
element.innerHTML = "<p>Lorem ipsum...</p>"
```

9.2.2

Which of the following piece of code is correct?

- document.getElementById("a").innerHtml
- document.getElementById("a").innerHTML
- document.getElementById("a").innerhtml
- document.getElementById("a").inner_html

📖 9.2.3

Let's assume the following HTML code:

```
<div id="a"> <p id="b"> c </p> </div>
```

In the first example, the content of the *div* element and *p* element is retrieved:

```
console.log(document.getElementById("b").innerHTML); // c
console.log(document.getElementById("a").innerHTML); // <p
id="b">c</p>
```

After applying the following code:

```
document.getElementById("b").innerHTML = "d";
```

The resulting structure will be as follow:

```
<div id="a"> <p id="b"> d </p> </div>
```

You can remove all children elements in this way as well. If you apply the next code:

```
document.getElementById("a").innerHTML = "";
```

The result will be as follow:

```
<div id="a"></div>
```

📝 9.2.4

Assume that, you have the following code:

```
<!DOCTYPE html>
<html>
    <head><title></title></head>
    <body></body>
</html>
```

Fill the gap, in order to get the following result:

```
<!DOCTYPE html>
<html>
    <head><title></title></head>
    <body><p></p></body>
</html>
document.body.innerHTML = '_____';
```

9.2.5

Which of the following piece of code removes all children elements in the *<body>* element?

- document.body.innerHTML = '';
- document.body.innerHtml = '';
- document.querySelector('.body').innerHTML = '';
- document.getElementById('body').innerHTML = '';

Event-driven Programming

Chapter 10

10.1 Event-driven programming

📖 10.1.1

Event-driven programming is a programming paradigm in which the flow of the program is determined by events, like mouse clicks, key-presses, time events, and so on.

An **event handler** is a function that is called when a particular event occurs. The event handler gets an **event object** as an argument. The event object has properties and methods related to the raised event, e.g., information about which mouse button was pressed when the mouse event was triggered.

HTML DOM events allow JavaScript to register different event handlers on elements in an HTML document. Each event is limited to the certain HTML elements, e.g. the *onload* event has the following supported HTML tags: *<body>*, *<frame>*, *<iframe>*, *<img>*, *<input type="image">*, *<link>*, *<script>* and *<style>*.

📝 10.1.2

What is event-driven programming?

- programming paradigm where the program is controlled by events like mouse-click, key-press etc.
- library (of functions) supporting identification and processing of events like mouse-click, key-press etc.
- part of DOM focused on the processing of special types of events at DOM elements

📖 10.1.3

The *onclick* attribute can be used to attach an event handler to the element.

Let's have the following function:

```
function clicked() {
    console.log('Clicked!');
}
```

Now, we will *attach* the function to the *<button>* element by putting some JavaScript code into the *onclick* attribute.

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

The *clicked()* function will be called a reaction for clicking the button.

In general, *JavaScript* code is allowed to be added to HTML elements through event handler attributes.

```
<element event="some JavaScript">
```

Hence, the same effect as in the above example can be achieved with the following code:

```
<button onclick="console.log('Clicked!')">Click me!</button>
```

10.1.4

Fill the gap to call function clicked() after click to defined button.

```
<button _____="_____()">Click me!</button>
```

10.1.5

The most common events are mouse events.

The ***onclick*** event occurs when the user clicks on an element, whereas the ***oncontextmenu*** event occurs when the user right-clicks on an element to open a context menu.

Lastly, the ***ondblclick*** event occurs when the user double-clicks on an element.

The ***onmousedown*** event occurs when the user presses a mouse button over an element, and the ***onmouseup*** event occurs when a user releases a mouse button over an element.

The last group of events is related to the movement of a mouse cursor.
The ***onmouseenter*** event occurs when the pointer is moved onto an element, and respectively, the ***onmouseleave*** event occurs when the pointer is moved out of an element. The ***onmousemove*** event occurs when the pointer is moving while it is over an element. The ***onmouseover*** event occurs when the pointer is moved onto an element, or onto one of its children.

10.1.6

Choose correct mouse event:

simple click - _____

right-click - _____

pressed mouse button - _____

released mouse button - _____

the mouse pointer is moved onto an element - _____

the mouse pointer is moving over an element - _____

- onmouseright
- onmouseclick
- onmousedown
- onclick
- onmousemove
- onmouseup
- onmouseenter
- onmousechar
- onmousepress
- oncontextmenu

📖 10.1.7

As an argument, an event handler function will receive the ***MouseEvent*** object, which provides properties and methods describing mouse interactions.

```
function mouseEventHandler(event) {
    if (event.button == 0) {
        console.log('Left mouse button clicked!');
    } else if (event.button == 1) {
        console.log('Middle mouse button clicked!');
    } else if (event.button == 2) {
        console.log('Right mouse button clicked!');
    }
}
<button onmousedown="mouseEventHandler(event)">Click
me!</button>
```

As you can see in the above example, you have to pass the ***event*** object into the event handler.

10.1.8

Fill the gaps:

```
function mouseEventHandler(_____) {
    if (event.button == 0) {
        console.log('_____ mouse button clicked!');
    } else if (event.button == 1) {
        console.log('Middle mouse button clicked!');
    } else if (event.button == 2) {
        console.log('_____ mouse button clicked!');
    }
}
<button onmousedown="mouseEventHandler(_____)">Click
me!</button>
```

- Left
- event
- event
- eventHandler
- Right

10.1.9

Assume that the message should be shown after a mouse double-click, fill the gap with the right event name.

```
<button _____="showMessage())">Click me!</button>
```

10.2 More event types

10.2.1

The next popular group of events is keyboards events. The *onkeydown* event occurs when the user is pressing a key, and respectively, the *onkeyup* event occurs when the user releases a key. The *onkeypress* event occurs when the user presses a key.

As an argument, an event handler function will receive the *KeyboardEvent* object, which provides properties and methods describing keyboard interactions.

```
function keyboardEventHandler(event) {
```

```
    console.log(event.key); // a single character (like "a",
"4" or "$") or a multicharacter (like "F1" or "Enter")
}
<input onkeydown="keyboardEventHandler(event)" type="text" />
```

📝 10.2.2

Assume that the message should be shown after a key is pressed (generating a char), fill the gap with the right event name.

```
<input _____="showMessage()" />
```

🕮 10.2.3

Events that are triggered by the user interface belongs to the UI events group.

The *onload* event occurs when an object has loaded.

The *onresize* event occurs when the document view is resized.

The *onscroll* event occurs when an element's scrollbar is being scrolled.

📝 10.2.4

Choose a correct event:

The _____ event occurs when an object has loaded.

The _____ event occurs when the document view is resized.

The _____ event occurs when an element's scrollbar is being scrolled.

- onresize
- onscroll
- onmove
- onsizechange
- ondrag
- onshift
- onload

📖 10.2.5

So far, we have used the combination of an HTML element event attribute and a *JavaScript* function. Now, we attach an event handler with *JavaScript* only, see the below syntax:

```
element.onload = function(event) { ... };
```

The above code attaches a single event handler to an element without an HTML attribute. See the below examples:

```
document.body.onload = function () {
    console.log('The page is loaded.');
};
document.querySelector('p').onclick = function () {
    console.log('The paragraph was clicked.');
};
```

📝 10.2.6

Fill in the space and set the response to clicking on the paragraph:

```
document.querySelector('p')._____ = _____ () {
    console.log('The paragraph was clicked.');
};
```

- func
- onclicked
- clicking
- onClicked
- function
- funct
- onclick

📖 10.2.7

JavaScript allows the execution of code at specified time intervals. These time intervals are called timing events. The two key methods to use with JavaScript are:

- The *setTimeout({function}, {milliseconds})* method executes a function, after waiting a specified number of milliseconds.
- The *setInterval({function}, {milliseconds})* method, same as *setTimeout()*, but repeats the execution of the function continuously.

The *clearTimeout()* method stops the execution of the function specified in the *setTimeout()* method.

```
let timer = setTimeout(function () {}, 1000);
clearTimeout(timer);
```

The *clearInterval()* method stops the executions of the function specified in the *setInterval()* method.

```
let timer = setInterval(function () {}, 1000);
clearInterval(timer);
```

10.2.8

Is it true?

The *setTimeout* method repeats the execution of the event handler function continuously.

- False
- True

10.3 Event handlers

10.3.1

You can add multiple event handlers with the *element.addEventListener({event}, {function}, {useCapture})*. The method takes three arguments:

- the required *event* that specifies the name of the event;
- the required *function* that specifies the function to run when the event occurs;
- optional *useCapture*, a Boolean value that specifies whether the event should be executed in the capturing or in the bubbling phase (this will be explained in the next card).

The following code presents exemplary usage of the *addEventListener* method:

```
let btn = document.querySelector('button');
btn.addEventListener('click', function(e) {
    console.log('First mesage');
});
```

```
btn.addEventListener('click', function(e) {
    console.log('Second message');
});
```

After clicking on the button, two messages should be displayed.

Do not use the *on* the prefix in event names. For example, use *click* instead of *onclick*.

📝 10.3.2

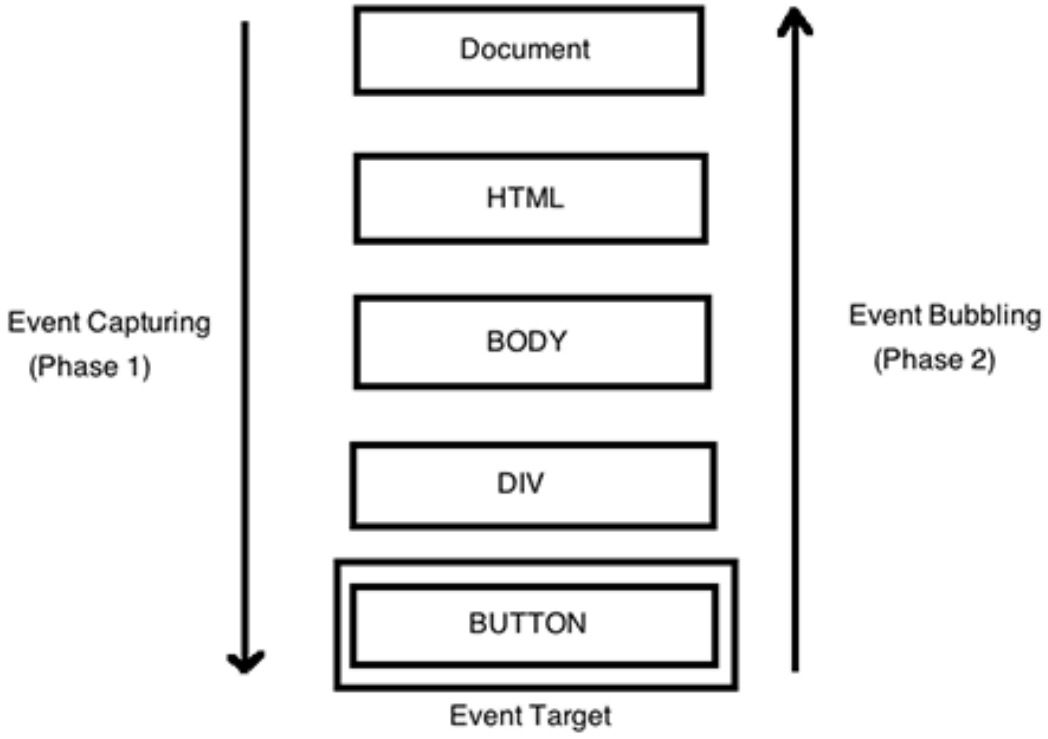
Is it possible to use the addEventListener method to attach multiple event handlers to the same element?

- yes
- no

🕮 10.3.3

Suppose you have assigned a click event handler on a *<a>* element, which is nested inside a *<p>* element. Now, if you click on that link, the handler will be executed. But, instead of the link, if you assign the click event handler to the paragraph containing the link, then even in this case, clicking the link will still trigger the handler. That's because events don't just affect the target element - they travel up and down through the DOM tree to reach their target. This is known as **event propagation**.

Event propagation is a mechanism that defines how events propagate or travel through the DOM tree to arrives at their target and what happens to it afterwards. Event propagation proceeds in three phases: **capturing**, **bubbling**, and **target** phase. Take a look at the following illustration:



Source: Medium: Event Bubbling and Event Capturing in JavaScript.

Event bubbling is the event that starts from the **deepest element or target element to its parents, then all its ancestors which are on the way from the bottom to top**, whereas event capturing is the event that starts from **the top element to the target element**. Modern browsers do not support event capturing by default, but you can turn it on with the previously mentioned *useCapture* argument. The target phase is when the event **reached the target element**.

See the example below:

```
<html>
    <head><title></title></head>
    <body>
        <p><button></button></p>
    </body>
</html>
document.querySelector('p').addEventListener('click', function
(event) {
    console.log('Second message');
});
document.querySelector('button').addEventListener('click',
function (event) {
    console.log('First message');
```

```
});
```

In the example above, after clicking on the button, both messages will be displayed.

10.3.4

What numbers are written after clicking on the image?

```
<!DOCTYPE html>
<html>
    <head><title></title></head>
    <body>
    <p><img src="image.png"></p>
    <script>
        document.querySelector('p').addEventListener('click',
function (event) {
         console.log('10');
       });
       document.querySelector('img').addEventListener('click',
function (event) {
         console.log('20');
       });
    </script>
    </body>
</html>
```

- 20, 10
- 10, 20
- 10
- 20
- nothing

10.3.5

If you want to stop the event bubbling, use the *event.stopPropagation()* method. See the example below:

```
<html>
    <head><title></title></head>
    <body>
        <p><button></button></p>
    </body>
```

```
</html>

document.querySelector('p').addEventListener('click', fucntion
(event) {
    console.log('Second message');
});
document.querySelector('button').addEventListener('click',
fucntion (event) {
    console.log('First message');
    event.stopPropagation();
});
```

In the example above, after clicking on the button, the first message will be displayed only.

You can use *event.stopImmediatePropagation()* to stop event propagation as well. The difference is that this method will also stop the rest of the attached event handlers from being executed.

📝 10.3.6

Use the *event.stopPropagation()* method to prevent attached event handlers from being executed.

- False
- True

🕮 10.3.7

The *element.removeEventListener({event}, {function}, {useCapture})* method removes an event handler that has been attached to the element. The arguments of the method are the same as in the *addEventListener* method. To remove event handlers, the function specified with the *addEventListener* method must be an external, *named* function, see the example below:

```
function clicked(event) {
    console.log('Clicked!');
}
document.querySelector('button').addEventListener('click',
clicked);
document.querySelector('button').removeEventListener('click',
clicked); // removes the event handler
```

If the event handler was attached two times, one with capturing and one bubbling, each must be removed separately.

📝 10.3.8

Fill the gap to remove event listener:

```
document.querySelector('button').addEventListener('click',
clicked);

document.querySelector('button')._____('click', clicked);
```

- removeEventListener
- deleteListener
- deleteEventListener
- removeListener

🕮 10.3.9

The *event.preventDefault()* method cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur. For example, clicking on a *Submit* button, prevent it from submitting a form or clicking on a link, prevent the link from following the URL. Not all events are cancelable. Use the *cancelable* property to find out if an event is cancelable.

```
document.querySelector('a').addEventListener('click', function
(event) {
    event.preventDefault();
});
```

📝 10.3.10

Use the event._____() to cancel the default action of an element.

🕮 10.3.11

You can use the *CustomEvent({eventName}, {detail})* for creating custom events. The *eventName* represents the name of the event. The *detail* argument is optional, and it is an event-dependent value associated with the event. See the example below:

```
let event = new CustomEvent('customEvent', {
    detail: {
        message: 'Fitped'
    }
});
```

You can attach the event name to an element.

```
document.querySelector('a').addEventListener('customEvent',
function (event) {
    console.log(event.detail.message);
});
```

To trigger the event, use the following code:

```
document.querySelector('a').dispatchEvent(event); // should
display the Fitped message
```

You can also trigger standard events like *click* or *load*, with the *Event*. See the example below:

```
const event = new Event('click');
document.querySelector('button').dispatchEvent(event);
```

10.3.12

Use the _____ to create custom events.

PRISCILLA

priscilla.fitped.eu