HTML

HTML structure

- `<!DOCTYPE html>`: The document type declaration for HTML5
- `<html>`: The root element of an HTML page
- `<head>`: Contains meta-information about the document
- `<title>`: The title of the document

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
content="width=device-width, initial-
scale=1.0">
    <link rel="stylesheet" href="styles.css">
<!-- CSS link -->
    <title>Welcome page</title>
</head>
<body>
    <h1>Hello students of the Web Technologies
course !</h1>
</body>
</html>
```

- width=device-width : the width of the page matches the screen width
- initial-scale=1.0 : the initial zoom level

Open Graph Protocol

The Open Graph protocol enables any web pages to show some additional information when shared on social media or messaging apps.

```
<meta property="og:title" content="The Rock" /
>
<meta property="og:type"
content="video.movie" />
<meta property="og:url" content="https://www.
imdb.com/title/tt0117500/" />
<meta property="og:image" content="https://ia.
media-imdb.com/images/rock.jpg" />
```

Lists

Unordered lists

Unordered lists are used to list items in no particular order. They are defined with the `<ul>` element.

```
<ul>
    <li>Item 1</li>
    <li>Item 2</li>
</ul>
```

Ordered lists

Ordered lists are used to list items in a specific order. They are defined with the `<ol>` element.

```
<ol>
    <li>Item 1</li>
    <li>Item 2</li>
</ol>
```

Hyperlink

Hyperlinks are used to link from one page to another. They are defined with the `<a>` element.

```
<a href="https://www.google.com"
title="Google" target="_blank">Google</a>
<a href="mailto:username@email.com?
subject=hello&body=world!">Hello World!</a>
```

Media elements

Images

```
<img src="image.jpg" alt="Image description">
```

Audio & Video

```
<audio src="audio.mp3"></audio>
<video src="video.mp4"></video>
```

Canvas

The canvas element allows for dynamic, scriptable rendering of 2D and 3D shapes.

```
<canvas id="myCanvas" width="200"
height="100"></canvas>
```

Tables

Tables are used to display data in a tabular format. They are defined with the `<table>` element.

```
<table>
    <!-- row -->
    <tr>
        <!-- header column -->
        <th>Student ID</th>
        <th>Grade</th>
    </tr>
    <tr>
        <!-- regular column -->
        <td>4</td>
        <td>5</td>
    </tr>
    <tr>
        <!-- merged columns -->
        <td colspan="2">6</td>
    </tr>
</table>
```

Forms

Forms are used to collect user input. They are defined with the `<form>` element.

```
<form action="/submit" method="post">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name">
    <input type="submit" value="Submit">
</form>
```

Semantic elements

```
<header>Header</header>
<nav>Navigation</nav>
<main>Main content</main>
<section>Section</section>
<article>Article</article>
<aside>Aside</aside>
<footer>Footer</footer>
```

CSS

CSS Selectors

CSS selectors are used to select the elements to which the ruleset will apply. There are several types of selectors:

Type

Type selectors select elements based on their tag name.

```
p {
  color: red;
}
```

This rule will apply to all `<p>` elements. in the html page.

Id

Id selectors select elements based on their `id` attribute.

```
#myId {
  color: red;
}
```

This rule will apply to the element with the `id="myId"` attribute.

Class

Class selectors select elements based on their `class` attribute.

```
.myClass {
  color: red;
}
```

This rule will apply to all elements with the `class="myClass"` attribute.

Universal

The universal selector * selects all elements.

```
* {
  color: red;
}
```

This rule will apply to all elements in the html page.

Grouping

Grouping selectors select multiple elements.

```
h1, h2, h3 {
  color: red;
}
```

Descendant

Descendant selectors select elements that are descendants of another element.

```
div p {
  color: red;
}
```

This rule will apply to all `<p>` elements that are descendants of a `<div>` element.

Child

Child selectors select elements that are direct children of another element.

```
div > p {
  color: red;
}
```

This rule will apply to all `<p>` elements that are direct children of a `<div>` element.

Flexbox

Flexbox is a layout model that allows elements to align and distribute space within a container.

```
<div class="container">
    <div class="item">Item A</div>
    <div class="item">Item B</div>
    <div class="item">Item C</div>
</div>
.container {
    display: flex;
    flex-direction: row;
}
.item {
    order: 1;
    flex-grow: 1
}
```

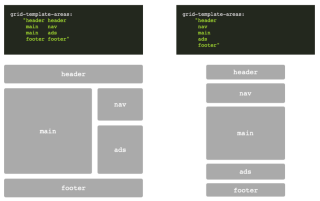
- Container (parent) properties : flex-direction, flex-wrap, flex-flow, justify-content, align-items, align-content
- Item (child) properties : order, flex-grow, flex-shrink, flex-basis, flex, align-self

Grid

Grid is a layout model that allows elements to align and distribute space in two dimensions.

```
<div class="container">
    <div class="item">Item A</div>
    <div class="item">Item
    B</div>
    <div class="item">Item C</div>
</div>
.container {
    display: grid;
    grid-template-columns: 1fr 1fr 1fr;
}
.item {
    grid-column: 1 / 3;
}
```

- Container (parent) properties : grid-template-column, grid-template-rows, grid-template-areas, grid-template, …
- Item (child) properties : grid-column-start, grid-column-end, grid-column, grid-row, grid-area, …



Media Queries

Media queries are used to apply different styles based on the device's characteristics.

```
h1 { font-size: 50px; }     /* General rule */

@media (min-width: 576px) { /* Tablet
dimensions */
    h1 { font-size: 60px; }
}

@media (min-width: 768px) { /* Desktop
dimensions */
    h1 { font-size: 70px; }
}
```

CSS variables

```
:root { /* Global variables */
    --main-color: #06c;
    --main-bg-color: #fff;
}

.my-element {
    color: var(--main-color, #06c);
    background-color: var(--main-bg-color,
#fff);
}
```

The var() function can take fallback values in case one of the variables is not defined.

JavaScript

Interpreted The interpreter reads the source code and executes it directly.
Just-in-time (JIT) compiled The interpreter compiles the hot parts of the source code into machine-code and executes it directly.
First-class functions Functions are treated like any other value. They can be stored in variables, passed as arguments to other functions, created within functions, and returned from functions.

Adding JavaScript directly to HTML

```
<script type='text/javascript'>
  console.log('Hello, World!');
  document.writeln('Hello, World!')
</script>
```

Adding JavaScript to an external file

```
<script src='script.js'></script>
```

- The defer attribute is used to defer the execution of the script until the page has been loaded.
- The async attribute is used to load the script asynchronously.

Primitive types

- **Undefined**: Unique primitive value undefined
- **Number**: Real or integer number (e.g. 3.14, 42)
- **Boolean**: true or false

- **String**: Character sequence, whose literals begin and end with single or double - quotes (e.g. “HEIG-VD”, ‘hello’)
- **BigInt**: Arbitrary-precision integers, whose literals end with an n (e.g. - 9007199254740992n)
- **Symbol**: Globally unique values usable as identifiers or keys in objects (e.g. Symbol(), Symbol(“description”))
- **Null**: Unique value null

Arrays

- Array: Ordered collection of values

The syntax for creating an array is:

```
let fruits = ['Apple', 'Banana', 'Cherry'];
```

Elements can be accessed using bracket notation:

```
console.log(fruits[0]);        // Apple
console.log(fruits.length);    // Banana
```

Methods on arrays

```
fruits.push("mango", "papaya");   // Appends new items
fruits.pop();                     // Removes and returns the last item
fruits.reverse();                 // Reverses the items' order
fruits.splice(2, 1, 'Orange');    // Replaces 1 elemnt at position 2 with 'Orange'
fruits.splice(1, 0, 'Peach');     // Inserts 'Peach' at index 1
```

Typeof operator

The typeof operator returns the type of a variable or expression.

```
console.log(typeof 42);          // number
console.log(typeof 'hello');     // string
console.log(typeof null);        // object
console.log(typeof [1, 2, 3]);   // object
```

Arithmetic operators

```
1 + 1;  // addition
1 - 1;  // subtraction
1 / 1;  // division
1 * 1;  // multiplication
1 % 1;  // modulo
1 ** 1; // exponentiation
```

String operators

```
"con" + "cat" + "e" + "nate";
`PI = ${Math.PI}`; // template literals (instead of: "PI = " + Math.PI)
```

In practice we should opt for template literals over string concatenation.

Automatic type conversion

Automatic type conversion is performed when comparing values of different types. It is at the root of many issues when using comparison operators.

```
"1" == 1 // true (!!!)
false == 0 // true
8 * null // 0
```

Strict equality

Strict equality compares two values for equality without type conversion.

```
"1" === 1 // false
"1" !== 1 // true
```

Variables

var

The var statement declares a **non-block-scoped** variable, optionally initializing it to a value. Its scope is its current execution context, i.e. either the enclosing function or, if outside any function, global. It can be re-declared.

```
var x = 1;
if (true) { var x = 2; }  // same variable
console.log(x);           // 2
```

let

The let statement declares a **block-scoped** local variable, optionally initializing it to a value. It can be re-assigned but not re-declared.

```
let x = 1;
{ let x = 2; }  // different variable (in a new scope)
console.log(x); // 1
let x = 1000;   // Error: redeclaration of let x
```

const

The const statement declares a **block-scoped** read-only named constant. It can be re-assigned but not re-declared.

```
const x = 1;
x = 2; // TypeError: Assignment to constant variable.
```

While and do-while loops

```
let num = 0;
while (num < 10) {
  console.log(num);
  num += 1;
}
let echo = "";
do {
    echo = prompt("Echo");
    console.log(echo);
} while (echo != "stop");
```

For loop

```
for (let num = 0; num < 10; num++) {
    console.log(num);
}
```

The for…in statement iterates over the enumerable properties of an object.

```
let obj = {a: 1, b: 2, c: 3};
for (let prop in obj) {
    console.log(prop, obj[prop]);
}
```

The for…of statement creates a loop iterating over iterable objects.

```
let nums = [0, 1, 2, 3, 4, 5, 6, 7, 8 , 9];
for (let num of nums) {
    console.log(num);
}
```

Declaration notation

```
function square(x) {
  return x * x;
}
// or
var square = function(x) {
    return x * x;
}
```

Arrow functions

```
var square = x => x * x
// or
var square = (x) => {
  return x * x;
}
```

Recursion

```
function factorial(n) {
  return n == 1 ? n : n * factorial(n-1);
}
console.log(factorial(5)) // 5 * 4 * 3 * 2 * 1 = 120
```

.. as long as it does not overflow the call stack.

Higher-order functions

```
function greaterThan(n) {
    return m => m > n;
}
let greaterThan10 = greaterThan(10);
console.log(greaterThan10(11)); // true
```

Regular expressions

Regular expressions are patterns used to match character combinations in strings. They are created using the RegExp constructor or a literal notation.

```
const re1 = /ab+c/;
const re2 = new RegExp(/ab+c/);
```

- `Character Classes (., \s, \d, …)` that distinguish types of chararters (resp. any, whitespace or digit)
- `Character sets ([A-Z], [a-z], [0-9], [abc], …)` that match any of the enclosed - characters (resp. uppercase letters, lowercase letters, digits, and any of a, b or c)
- `Either operator (x|y)` that match either the left or right handside values
- `Quantifiers (\*, +, ?, {n}, {n,m})` that indicate the number of times an expression matches
- `Boundaries (^, \$)` that indicate the beginnings and endings of lines and words
- `Groups ((), (?<name>), (?:))` that extracts and remember (or not) information from the input
- `Assertions (x(?=y))` that helps at defining conditional expressions

```
const emailRegex = /^[a-zA-Z]([a-zA-Z0-9._-]+)@(?:[a-zA-Z0-9]+\.)+(?:com|org|net)$/;
```

Flags

```
const re1 = /ab+c/;   // no flag
const re2 = /ab+c/g;  // global search
const re3 = /ab+c/i;  // case-insensitive search
const re4 = /ab+c/m;  // multi-line search
const re5 = /ab+c/gi  // global case-insensitive search
```

Context

```
function doTwice(f) {
    f.call(this); // bind this to the current context
    f.call(this); // bind this to the current context
}
let human = {
    age: 32,
    getOlder() {
        this.age++;
    }
}
doTwice.call(human, human.getOlder); // bind this to human
console.log(human.age); // Output will be 34
```

Array methods

- `concat()` concatenates two or more arrays and returns a new array.
- `join()` joins all elements of an array into a string.
- `pop()` removes the last element from an array and returns that element.
- `push()` adds one or more elements to the end of an array and returns the new length of the array.
- `reverse()` reverses the order of the elements of an array.
- `shift()` removes the first element from an array and returns that element.
- `slice()` selects a part of an array, and returns it as a new array.
- `sort()` sorts the elements of an array.
- `includes()` determines whether an array contains a specified element.
- `flat()` flattens an array up to the specified depth.