JavaScript 1

**Kalob & Andrei 2**

With DOM

**2.1 Functions**

* Function Declaration:

**function** function\_name(param1, param2, .. ){

//codes using param1, pram2

}

* Function Invocation: The code inside the function will execute when "*something*" *invokes (calls)* the *function*:

1. When an *event* *occurs* (when a user *clicks* a button)
2. When it is *invoked* (called) from *JavaScript code*
3. *Automatically* (self invoked)

* Calling a function: Using **()** with function name will invoke the function immediately. If we want to invoke the function later we should use only the function name.
* **function\_name()** invokes immediately.
* **function\_name** can be used in statements like a variable.

function **func\_1**(){

**alert**("My name is NOT Frank Zappa");

}

/\* Calling/invoking a function \*/

**func\_1**();

function **func\_2**(name1, name2){

**alert**(name1 + "is the first name\n" + name2 + " is the second name");

}

**func\_2**("Plaw", "Student");

* Handling parameter exception: We ca use if condition as follows if some parameters are not present:

// Handling parameter exceptions

function **func\_3**(name1, name2){

    if (name1==undefined || name2==undefined) {

        var message = "Missing one parameter";

    } else{

        var message = (name1 + "is the first name\n" + name2 + " is the second name");

    }

**alert**(message);

}

**func\_3**("Plaw", "Student");

* Or, we can use var-args.
* Returning from a function: If we want to use the result of execution of a function, after the function's scope ends, we should use the *return* *statement*.

// Returning from a function

function **myFunction**(a, b) {

  return a \* b;             // Function returns the product of a and b

}

var x = **myFunction**(4, 3);   // Function is called, returned value will end up in x

**alert**(**myFunction**(4, 3));    // returned value will be aleretd

document.**write**(**myFunction**(4, 3));   // returned value will be written in Html, documnet

console.**log**(**myFunction**(4, 3));      // returned value will be console-logged

* **return** will *jumps out* from the *scope*. No code will executed which appears after "**return**" statement in the same scope.

// Returning from a function

function **myFunction**(a, b) {

    return a \* b;             // Function returns the product of a and b

**alert**(a+b); // This never executes

  }

* We can return any strings or values using "**return**" statement.
* We can also use "**break**" statement to **escape** the **scope**.

// Returning different values

function **NameOrAge**(type, value){

    type = type.**toLowerCase**();

    if(type=="age"){

        return value + value; //  addition arithmetic; doubles your age.

    } else if(type=="name"){

        //to conjugate, create a new variable first.

        // Then return the variable

        var newString = "Your name is " + value;

        return newString;

    } else{

        // You can return full strings. Remember to escape quotes inside quotes,

        return "Type did not match \"age\" or \"name\"";

    }

}

**2.2 JavaScript Events**

An event is something that happens when an action is taken on a web page. We have a few to go through;

onClick,

onDblClick.

onFocus.

onBlur,

onMouseDown,

onMouseUp.

onMouseMove,

onMouseOut,

onMouseOver.

onKeyDown,

onKeyPress, and

onKeyUp

* Some of these are pretty similar to each other but each event has it's own use. These are global attributes and can be applied to almost all HTML elements.
* Example: consider the following example:

        <script type="text/javascript">

            function **plihi**(){

**alert**("Plihi");

            }

        </script>

        <span onclick="**plihi**()">click to Plihi</span>

**2.3 DOM\_1 :Selecting HTML elements using** getElementById()

|  |  |
| --- | --- |
| * Selection in DOM:   document.firstElementChild;  document.lastElementChild;  document.firstElementChild.lastElementChild;   * Manipulation: Once you've successfully selected the object that you're interested in inside the DOM then you can manipulate it. So I can simply save it inside a variable. For example:   var heading\_1 = document.firstElementChild.lastElementChild.firstElementChild;  heading\_1.innerHTML = "There you go"; |  |

* Important selector methods:

// ----------------    Selection elements in DOM    ---------------------

// -------    Tag, Class and Id selection    ----------

​

//notice **plural** form and it returns an array

document.getElementsByTagName("li");

​

​

//following changes color of **bullet** **only**. To change text color we need to be **specific**

document.getElementsByTagName("li")[0].style.color = "red";

​

//following **not works** because of being **array/collection**

document.getElementsByTagName("li").style.color = "red";

​

​

//collections also has **length** as array.

​

document.getElementsByTagName("li").length;

​

document.getElementsByClassName("item");

​

document.getElementsByClassName("li").length;

​

document.getElementsByClassName("item").length;

​

//notice following method is **not plural**

document.getElementById("title");

document.getElementById("title").style.color = "green";

​

​

​

//------- CSS-fashioned selector QUERY-Selector--------

​

//following only select first matched element (not all). Notice "# for id" or ". for class"

​

document.querySelector("#title").style.color = "pink";

document.querySelector(".item").style.color = "orange";

​

//following select all matched elements. But gives ERROR Notice "# for id" or ". for class"

document.querySelectorAll(".item").style.color = "orange";

document.querySelectorAll(".item"); //this works

​

//slecting specific element by combining,

document.querySelectorAll(".item")[2].style.color = "orange";

​

//Notice the index is used even if there is only one item

document.querySelectorAll("li:nth-of-type(3)")[0].style.color = "red";  //affects the text also.

document.querySelectorAll(".list a")[0].style.color = "orange";  //affects the text also.

​

​

​

//querySelectorAll mosty used. And it is easy to specify the elements.

**2.4 DOM in CSS**

In this case Camel case is used instead of dashes "-". And the values are represented as strings.

// ------------  DOM and CSS  --------------

document.querySelector(".btn").style.backgroundColor = "yellow";

* However we usually don’t use this kind of code. We separate these interactive css-properties into *separate classes* in our *.css* file and TOGGLE them using JS.

**2.5 The Separation of Concerns Structure vs Style vs Behavior**

|  |  |  |
| --- | --- | --- |
| **classList**  **classList.add**  **classList.remove**  **classList.toggle** | Manipulating Style: For example we toggle following class  .big{      font-size: 10rem;      color: aqua;      font-weight: bold;  }  Using: document.querySelector("h1").classList.toggle("big"); | |
| **2.6 Text and Attribute manipulation** | | |
| * Text Manipulation: | | innerHtml:    textContent: |

* Attribute manipulation (getAttribute, setAttribute):

|  |  |
| --- | --- |
|  |  |

// Text and Attribute manipulation

document.querySelector("h1").innerHTML = "BaBA <em>Dock</em>";

​

document.querySelector("h1").textContent = "BaBA <em>Dock</em>";

​

document.querySelector("a").getAttribute("href");

​

* Changing attribute values:

document.querySelector("a").setAttribute("href", "www.bing.com");

​

document.querySelector("a").attributes;

* Example: Another Example of DOM:

        <script>

            function **$**(id) {

                return document.**getElementByld**(id);

            }

            function **ChangeMe**() {

**$**('TestlD').style.border = "1px solid red";

**$**('TestlD').innerHTML = "Added a border and this text!";

**$**('Button').value = "Changed!";

**$**('Button').disabled = true;

            }

        </script>

**2.7 Escape sequences and Escape-Method**

* As before we used \n, \t, \', \" .
* However we can also use ***escape()*** method.

var EscapeMe = "The cow jumped over the moon!?";

document.**write**(**escape**(EscapeMe));

// deprecated — A legacy feature for browser compatibility

// param string — A string value

* Unescape: It does the reverse.

/\* Unescape  \*/

var un\_escape = "The%20cow%20jumped%20over%20the%20moon%21%3F";

console.**log**(**unescape**(un\_escape));

// The cow jumped over the moon!?

* The ***escape()*** function is deprecated for URLs.
* Use ***encodeURI()*** or ***encodeURIComponent()*** instead.

**2.8 Arrays**

* Create array and then elements:

/\* Strings as array \*/

var name\_1 = "Anton";

// Create empty array and then add values

var MyArray = new **Array**();

MyArray[0]= "Bobdv";

MyArray[1]= "Andi";

MyArray[2]= "Serov";

console.**log**(MyArray);

* Literal array:

/\* Literal array \*/

var MyArray\_2 = ["Bevvy", "Ben", "Richie", "Penny wise"];

* Multi Dimensional array:

/\* Multi Dimensional array \*/

var Multi\_Arr = [["Bevvy", "Ben", "Richie", "Penny wise"], ["Ron", "Hermione", "Harry", "Malfoy"]]

**alert**(Multi\_Arr[1][2])

* JavaScript supports different kinds of loops:
* ***for*** - loops through a block of code a number of times
* ***for/in*** - loops through the properties of an object
* ***for/of*** - loops through the values of an iterable object
* ***while*** - loops through a block of code while a specified condition is true
* ***do/while*** - also loops through a block of code while a specified condition is true

**2.9 For Loops**

* For loop with increment: The for loop has the following syntax:

for (statement 1; statement 2; statement 3) {

// code block to be executed

}

*Statement 1* is executed (one time) before the execution of the code block.

*Statement 2* defines the condition for executing the code block.

*Statement 3* is executed (every time) after the code block has been executed.

// For loop with increment

for (var i = 0; i<=5; i++ ){

    document.**write**("My Nae is Khan");

    document.**write**(" Looped "+i+" Times <br>");

}

document.**write**("My Nae is <strong>Khan</strong>");

* For in loop:

// For in loop

var MyArray\_2 = ["Bevvy", "Ben", "Richie", "Penny wise"];

for (nm in MyArray\_2){

    // Returns the index number

    console.**log**(nm);

    console.**log**(MyArray\_2[nm]);

}

* For of loop:

// For of loop

let cars = ["BMW", "Volvo", "Mini"];

let text = "";

for (let x of cars) {

  text += x + "  ";

}

* ***Array.forEach():*** The ***forEach()*** method calls a function (a callback function) once for each array element.

// For each loop

var txt = "";

var numbers = [45, 4, 9, 16, 25];

numbers.**forEach**(myFunction);

function **myFunction**(value) {

  txt = txt + value + "<br>";

}

console.**log**(txt);

**2.10 While Loops**

The ***while*** loop loops through a block of code as long as a specified condition is true.

// While loop

while (condition) {

  // code block to be executed

}

* Example:

// While loop

while (condition) {

  // code block to be executed

}

i = 1;

text = '';

while (i < 10) {

  text += "The number is " + i;

  i++;

}

console.**log**(text);

* Example:

var cars = ["BMW", "Volvo", "Saab", "Ford"];

var i = 0;

var text = "";

while (cars[i]) {

  text += cars[i] + "<br>";

  i++;

}

console.**log**(text);

* Do-While loop: The do/while loop is a variant of the while loop. This loop will *execute* the code block *once*, *before checking* if the condition is true, then it will repeat the loop as long as the condition is true.

// Do While loop

do {

  // code block to be executed

}

while (condition);

* Example:

i = 5;

text = '';

do {

  text += "The number is " + i;

  i++;

}

while (i < 10);

console.**log**(text);

**If you forget to** increase **the** variable **used in the condition, the** loop **will** never end**. This will** crash **your** browser**.**

**2.11 Break-Continue & label:** Breaking Out Of Loops and Skipping A Loop Cycle

* The *break* statement "*jumps out*" of a loop.
* The *continue* statement "*jumps over*" *one iteration* in the loop.

text = '';

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

  if (i === 3) { break; }

  text += "The number is " + i + "<br>";

}

console.**log**(text);

* Continue: The continue statement *breaks* *one iteration* (in the loop), if a specified *condition* occurs, and *continues* with the *next iteration* in the loop.
* This example skips the value of 3:

text = '';

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

  if (i === 3) { continue; }

  text += "The number is " + i + "<br>";

}

console.**log**(text);

* Label: To label JavaScript statements you precede the statements with a label name and a colon:

**label:**

**statements**

* The break and the continue statements are the only JavaScript statements that can "jump out of" a *code block*.

**break** labelname;

**continue** labelname;

* The ***continue*** statement (with or without a label reference) can only be used to skip *one loop iteration*.
* The ***break*** statement, without a label reference, can only be used to *jump out of a loop* or a *switch*.
* With a ***label*** reference, the break statement can be used to *jump out of any* code block:

// Label statement

text = '';

var cars = ["BMW", "Volvo", "Saab", "Ford"];

list: {

  text += cars[0] + "<br>";

  text += cars[1] + "<br>";

  break list;

  text += cars[2] + "<br>";

  text += cars[3] + "<br>";

}

document.**write**(text);

**2.12 String operation: split & substring**

* **split():** A string can be converted to an array with the ***split()*** method:

var myString = "The quick brown fox jumps over the lazy dog";

var AmSplit = myString.**split**(" ");

document.**write**("Here is the array: <br />");

for(index in AmSplit) {

    document.**write**(AmSplit[index] + ", ");

}

* Specifying separator:

**split**(seperator, limit);

// seperator, limit both are optional.

// If the separator is omitted, "split()" returned array will contain the whole string in index [0].

// If the separator is "", 'split("")' returned array will be an array of single characters

* **substr():** substr() is how we can extract parts of a string through character position. Every letter, number and space takes up one character position.

substr(starting\_character, length\_to\_extract)

* ***starting\_character*** is the *numeric value* of where to *start* the extraction, required ***length\_to\_extract*** is *how much* of the string to take out. If empty, the rest of the string will be extracted optional.

/\* substr, substring & slice \*/

var myString = "I love this website!";

document.**write**(myString.**substr**(0, 7) + "The Complete Web Dev");

* slice(): ***slice()*** extracts a part of a string and returns the extracted part in a new string.
* The method takes 2 parameters: the *start position*, and the *end position* (end not included). This example slices out a portion of a string from position 7 to position 12 (13-1):

Example

var str = "Apple, Banana, Kiwi";

var res = str.**slice**(7, 13);

// The result of res will be: Banana

* If a parameter is *negative*, the position is *counted* *from* the *end* of the string.

**2.13 Exception handling:** Try..Catch And Throw

try{

**allert**("This is typed bad!"); //notice "ällert" is incorrect spelling

}

catch(e){ // e for error

    e = "This error occurred: \n" + e

**alert**(e);

}

* Throw: is user created exception.

// Using throw

var Name\_1 = **prompt**("Enter your name:");

try{

    var leng = Name\_1.length; // Length of user input

    if(leng >5) {

        throw "Long Name";

    } else if(leng < 5) {

        throw "Short Name";

    } else {

        throw "5 letter name";

    }

}

catch(e){

**alert**(e);

}

**2.14 Date and Time**

|  |  |
| --- | --- |
| var d = new **Date**();  document.**write**(d);   * We can also use different properties of **Date**() object. * Remember "0" means "Sunday". |  |

* Some other useful **Date()** properties.

d.**getDay**(); //Returns current date

d.**getDay**(); //Returns week-day, 0=sunday

d.**getFullYear**(); //Returns 4-digit year

d.**getTime**(); //Returns time from midnight 1970, 1 Jan in milliseconds

d.**getMonth**(); //Returns Current Month, 0=Jan

d.**getSeconds**(); //Returns current seconds

d.**getMinutes**(); //Returns current minutes

d.**getHours**(); //Returns current hour

* Example: To return Week-days in "String".

// Using day string

var d = new **Date**();

var day = d.**getDay**();

var weeKDays = new **Array** ("Sunday", "Monday", "Tuesday", "Wednesday" , "Thursday", "Friday", "Saturday");

document.**write**(weeKDays[day]);

* Creating Date Objects: Date objects are created with the ***new Date()*** constructor. ***new Date()*** creates a new date object with the current date and time.
* There are 4 ways to create a new date object:

/\* Ways to create Date() object \*/

new **Date**()

new **Date**(year, month, day, hours, minutes, seconds, milliseconds)

new **Date**(milliseconds)

new **Date**(date string)

**2.15 Advanced math**

a\_1 = Math.**round**(1.555); // rounds the value

console.**log**(a\_1);

a\_2 = Math.**pow**(2, 5); //gets 2^5

console.**log**(a\_2);

a\_3 = Math.PI; //gets pi's value

console.**log**(a\_3);

a\_4 = Math.**ceil**(1.555); //gets ceilling value

console.**log**(a\_4);

a\_5 = Math.**floor**(1.555); //gets floor value

console.**log**(a\_5);

/\* Min and Max  \*/

console.**log**("Minimum of: [1, 5, 8, 0.5, 6] :"+ Math.**min**(1, 5, 8, 0.5, 6));

console.**log**("Maximum of: [1, 5, 8, 0.5, 6] :"+ Math.**max**(1, 5, 8, 0.5, 6));

// Other functions

Math.**cos**(30);

Math.**sin**(30);

Math.**tan**(30);

Math.**sqrt**(30);

Math.**log**(30);

// BEDMAS rule: Breces, Exponents, Division/Multiplication, Addition/Subtraction

console.**log**(Math.**pow**(((19+1)\*5), 2))

**2.16 Regular Expressions: RegExp**

* RegExp: A regular expression is a way to define a pattern and seek out matches based on this specific pattern. You can
* search,
* serach and replace and
* test your Regular Expressions.
* *Regular Expressions*, or *RegExp*, is not an easy thing to learn. And to be honest, most web developers don't know very much RegExp - just because it can be so confusing.
* It can also be very' precise and useful.
* A regular expression is a *sequence of characters* that forms a *search pattern*. The search pattern can be used for *text search* and *text replace* operations.
* A regular expression can be a single character, or a more complicated pattern.
* Regular expressions can be used to perform all types of

1. text search and
2. text replace operations.

Syntax:

**/pattern/modifiers;**

* Example:

var patt = /w3schools/i;

***w3schools*** is a pattern (to be used in a search).

***i*** is a ***modifier*** (modifies the search to be case-insensitive).

**Objectives:**

* Defining a new RegExp
* The Pattern Symbols
* Modifiers
* Testing and Using
* Defining New RegExp: Like how we defined a **new array** or **new Date()**; , we use the same type of declaration i.e. we can create an object.

/\* RegExp object declaration \*/

var MyRegExp = new **RegExp**(pattern, modifiers);

* Often you will see this written differently.. more like this:

// We can also use following version

var MyRegExp = /pattern/modifiers;

* Where the ***pattern*** goes inside the slashes "/" followed by your ***modifiers***.
* A ***modifier*** is a ***specific rule*** the pattern should stick to. Like being ***case-sensitive*** or to be ***global***. Modifiers can also be called "***flags***". Notice in the following code, how we used the RegExp: /[a-z0-9]/.

        if (/[a-z0-9]/.test(char)) {    //Using RegExp

            if (obj[char] > 0) {

                obj[char]++;

            } else {

                obj[char] = 1;

                };

        }

* Using String Methods: In JavaScript, regular expressions are often used with the two string methods: *search()* and *replace()*.
* The ***search()*** method uses an expression to search for a match, and returns the position of the match.
* The ***replace()*** method returns a modified string where the pattern is replaced.

|  |  |
| --- | --- |
| RegExp | Without RegExp |
| // Using String search() With a Regular Expression  var str = "Visit W3Schools";  var n = str.**search**(/w3schools/i); | // Using String search() With a String  var str = "Visit W3Schools!";  var n = str.**search**("W3Schools"); |
| // Use String replace() With a Regular Expression  var str = "Visit Microsoft!";  var res = str.**replace**(/microsoft/i, "W3Schools"); | // Using String replace() With a String  var str = "Visit Microsoft!";  var res = str.**replace**("Microsoft", "W3Schools"); |

* Regular Expression Modifiers:

Modifiers can be used to perform *case-insensitive* more global *searches*:

|  |  |
| --- | --- |
| Modifier | Description |
| **i** | Perform *case-insensitive* matching |
| **g** | Perform a *global match* (find all matches rather than stopping after the first match) |
| **m** | Perform *multiline matching* |

/\* Example \*/

var myExp = /Rubinn/g;

var test = myExp.**test**("My name is Rubinn");

**alert** (test);

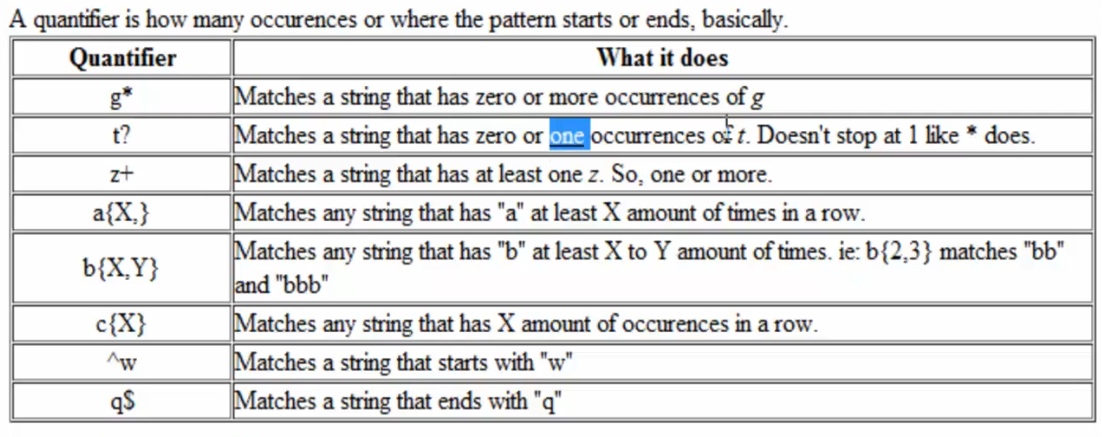
* The Pattern Symbols:
* Brackets are used to find a range of characters:

|  |  |
| --- | --- |
| Expression | Description |
| **[abc]** | Find any of the characters between the brackets |
| **[0-9]** | Find any of the digits between the brackets |
| **(x|y)** | Find any of the alternatives separated with **|**, **|** means **OR** |

* Similarly we can use **[a-z], [A-Z], [A-z], [0-9]**
* **[^xyz]** here ^ means NOT, i.e. find any characters NOT in the brackets.

* Quantifiers define quantities:

|  |  |
| --- | --- |
| Quantifier | Description |
| **n+** | Matches any string that contains at least one n |
| **n\*** | Matches any string that contains zero or more occurrences of n |
| **n?** | Matches any string that contains zero or one occurrences of n |



/\* Example Modifiers & Quantifiers \*/

var myExp = /n{2}/g;

var test = myExp.**test**("My name is Rubinn");

**alert** (test);

* Metacharacters: Metacharacters are characters with a special meaning:

|  |  |
| --- | --- |
| Metacharacter | Description |
| **\d** | Find a digit |
| **\s** | Find a whitespace character |
| **\b** | Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b |
| **\uxxxx** | Find the Unicode character specified by the hexadecimal number xxxx |

/\* Matchers \*/

var myExp = /Rubinn*\n*/g;

var tst =myExp.**test**("My name is Rubinn\n test");

**alert**(tst);

|  |  |
| --- | --- |
| Metacharacter | Description |
| . | Find a single character, except newline or line terminator |
| \w | Find a word character |
| \W | Find a non-word character |
| \d | Find a digit |
| \D | Find a non-digit character |
| \s | Find a whitespace character |
| \S | Find a non-whitespace character |
| \b | Find a match at the beginning/end of a word, beginning like this: \bHI, end like this: HI\b |
| \B | Find a match, but not at the beginning/end of a word |
| \0 | Find a NULL character |
| \n | Find a new line character |
| \f | Find a form feed character |
| \r | Find a carriage return character |
| \t | Find a tab character |
| \v | Find a vertical tab character |
| \xxx | Find the character specified by an octal number xxx |
| \xdd | Find the character specified by a hexadecimal number dd |
| \udddd | Find the Unicode character specified by a hexadecimal number dddd |

* Using different methods:
* test

/\* .test() If you want to test if your RegExp works\*/

yourRegExp.**test**("your string");

* replace

/\* replace \*/

// string.replace("string or RegExp", "replacement");

var myExp = /sentence/gi;

var str = "Change this sentence";

document.**getElementById**("demo6").innerHTML = str.**replace**(myExp, "str");

* match

/\* match \*/

var myRegExp = /string/g;

var str = "This is my test string string string".**match**(myRegExp);

for(v in str) {

**alert**(str[v]);

}

var matcs = str.**length**();

**alert**(matcs)

**2.17 Get the users browser's info**

* Window Navigator: The window.navigator object contains information about the visitor's browser.
* The ***window.navigator*** object can be written without the window prefix.

Some examples: navigator.appName

navigator.appCodeName

navigator.platform

* Example: Following represents different properties of "navigator"

<body>

    <p id="demo"></p>

    <script>

        // Browser Cookies: The cookieEnabled property returns true if cookies are enabled, otherwise false:

        document.**getElementById**("demo").innerHTML = "cookiesEnabled is " + navigator.cookieEnabled + "<br>";

        // Browser Application Name: The appName property returns the application name of the browser:

        // "Netscape" is the application name for both IE11, Chrome, Firefox, and Safari.

        document.**getElementById**("demo").innerHTML += "navigator.appName is " + navigator.appName + "<br>";

        // Browser Application Code Name: The appCodeName property returns the application code name of the browser:

        // "Mozilla" is the application code name for both Chrome, Firefox, IE, Safari, and Opera.

        document.**getElementById**("demo").innerHTML += "navigator.appCodeName is " + navigator.appCodeName + "<br>";

        // The Browser Engine: The product property returns the product name of the browser engine:

        // Do not rely on this. Most browsers returns "Gecko" as product name !!

        document.**getElementById**("demo").innerHTML += "navigator.product is " + navigator.product + "<br>";

        // The Browser Version: The appVersion property returns version information about the browser:

        document.**getElementById**("demo").innerHTML += navigator.appVersion + "<br>";

        // The Browser Agent: The userAgent property returns the user-agent header sent by the browser to the server:

        //     Warning !!! The information from the navigator object can often be misleading, and should not be used to detect browser versions because:

        //         Different browsers can use the same name

        //         The navigator data can be changed by the browser owner

        //         Some browsers misidentify themselves to bypass site tests

        //         Browsers cannot report new operating systems, released later than the browser

        document.**getElementById**("demo").innerHTML += navigator.userAgent + "<br>";

        // The Browser Platform: The platform property returns the browser platform (operating system):

        document.**getElementById**("demo").innerHTML += navigator.platform + "<br>";

        // The Browser Language: The language property returns the browser's language:

        document.**getElementById**("demo").innerHTML += navigator.language + "<br>";

        // Is The Browser Online?: The onLine property returns true if the browser is online:

        document.**getElementById**("demo").innerHTML += navigator.onLine + "<br>";

        // Is Java Enabled?: The javaEnabled() method returns true if Java is enabled:

        document.**getElementById**("demo").innerHTML += navigator.**javaEnabled**() + "<br>";

        </script>

</body>

Output:

cookiesEnabled is true

navigator.appName is Netscape

navigator.appCodeName is Mozilla

navigator.product is Gecko

5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/106.0.0.0 Safari/537.36

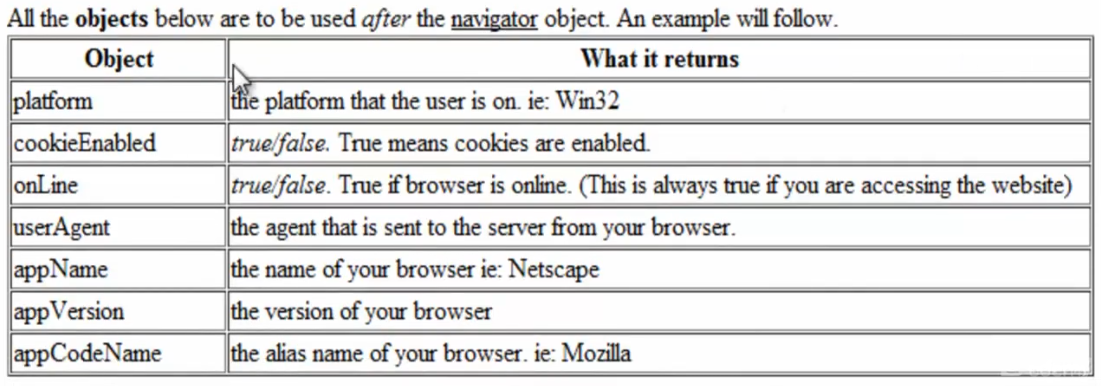
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/106.0.0.0 Safari/537.36

Win32

en-US

false

false



**2.18 JavaScript Timing Events :** setlnterval **&** setTimeout

* Timing Events: The window object allows execution of code at specified time intervals. These time intervals are called timing events.
* The two key methods to use with JavaScript are:

**setTimeout**(function, **milliseconds**);

**setInterval**(function, **milliseconds**);

* The ***setTimeout()*** and ***setInterval()*** are both methods of the ***HTML DOM*** Window ***object***.
* **setTimeout(function, milliseconds):** *Executes a function*, after *waiting* a specified number of milliseconds.
* ***function*** is our function to run.
* ***millisecond*** delay is the delay time. 1000 = 1 second

// setTimeout Example: an alert will appear after 5 second delay

var delayedAlert = **setTimeout**(function(){**alert**("Delayed!");}, 5000)

* **setInterval(function, milliseconds):** Same as setTimeout(), but *repeats the execution* of the function *continuously*.
* ***function*** is the code we want to run.
* ***millisecond*** delay is how long we "wait" between each interval. 1000 = 1 second.

Here's an example of the **setInterval**.

function **tIMe**(){

    var d = new **Date**();

    var hour = d.**getHours**();

    var minute = d.**getMinutes**();

    var second = d.**getSeconds**();

    var str = hour +":"+ minute +":"+ second;

    document.**getElementById**("demo").innerHTML = str;

}

var initiate = **setInterval**(function(){**tIMe**()}, 1000);

**2.19 JS Objects**

Objects are *variables* too. But *objects* can contain *many values*. This code assigns many values (Fiat, 500, white) to a variable named car, we use ***var*** to declare an object:

// This code assigns many values (Fiat, 500, white) to a variable named car:

var car = {

    type:"Fiat",

    model:"500",

    color:"white"

};

var person = {

    firstName: "John",

    lastName : "Doe",

    id       : 5566,

**fullName** : function() {

      return this.firstName + " " + this.lastName;

    }

  };

* The this Keyword: In a function definition, ***this*** refers to the "***owner***" of the function. In other words, ***this.firstName*** means the ***firstName*** property of this ***object***.

// This code assigns many values (Fiat, 500, white) to a variable named car:

var car = {

    type:"Fiat",

    model:"500",

    color:"white"

};

// Accessing Object Properties

// You can access object properties in two ways:

objectName.propertyName

car.color

objectName["propertyName"]

car[model]

// Object Methods: Objects can also have methods. Methods are actions that can be performed on objects.

// Methods are stored in properties as function definitions.

var person = {

    firstName: "John",

    lastName : "Doe",

    id       : 5566,

**fullName** : function() {

      return this.firstName + " " + this.lastName;

    }

  };

//   In a function definition, this refers to the "owner" of the function. In other words, this.firstName means the firstName property of this object.

// Accessing Object Methods: You access an object method with the following syntax:

objectName.**methodName**()

// Example

nAme = person.**fullName**();

// If you access a method without the () parentheses, it will return the function definition:

// Example

nAme\_2 = person.fullName;

* Example: Notice the following example, how we access and change the object properties

var MyWallet = {

    cards: {

        visa:   1,

        mastercard: 1,

        arnex:  0,

        debit:  1

    },

    cash: {

        Canadian:   1500,

        American:   2200.5,

        Euros:  "No use"

    },

    id: "Yes"

}

console.**log**(MyWallet.cash.Canadian);

// Changing object propertie value

MyWallet.cash.Canadian = 5000;

console.**log**(MyWallet.cash.Canadian);

MyWallet['cash']['American'] = 10000;

console.**log**(MyWallet.cash.American);

MyWallet['cash'].Euros= "Lost";

console.**log**(MyWallet.cash['Euros']);

/\* Do not use "." with [] \*/

// Wrong: MyWallet['cash'].['American'] = 10000;

// Wrong: console.log(MyWallet.cash.['Euros']);