JS basics sheet

A scripting language to make the web page dynamic I.e. responding to user interactions.

There are 3 ways to insert the JS code into your project:

1. Inline script e.g.:

   <button onclick="**alert('hello world')**"> click me </button>

2. Internal script: using script tag in your Html document. It is more effective when you have multiple things as buttons...etc. Having the same method e.g.:

   <button onclick="do()"> click me </button>

   <script> do() : void{.....} </script>

3. External script in external document but you will only need to link the Html and js files through script tag e.g.

   <button onclick="do()"> click me </button>

   <script src = "./jsFile.js"> </script>

If you are studying this sheet to get into a framework such as React, Angular, or ...etc., we will use the third way and fortunately there will be no need to link the JS and html files as the framework will do it itself!

Now let's start from scratch …

- ## Printing statements

There are multiple ways to print a statement In JS

1. document.write("the statement to print")

   It prints statements as paragraphs into the html document connected to. Usually, used for debugging purposes

2. alert('the statement to print') or window.alert('statement')

   it displays a small window on the top of the screen with the statement to print. Usually, used as a warning

3. Console.log("the statement to print")

   It prints the statement in the browser console not in the html document. That is why it is the most powerful for the debugging purposes. It can also be used to print many variables together, just like the following :

   ```
   Console.log(name, length, address)
   ```

   Notes:

   - In JS, no need for **; semicolons.** Use it when you need many statements on the same line
   - In JS, strings can be written between **" "** or **' '**. in case you want to print sth. In "" use enclosing ' ' and vice versa, or using **\"**, **\'** as in java, C, ...etc.

- ## Comments

  You can comment a line using **//** or multiple lines using **/***

- ## Variables

  There are 2 ways to declare a variable. One using let keyword and the other using var keyword. The **only difference** is the **scope** e.g.

  ```
  let length = 10   // local to its {...} scope

  var shape = 'circle';  //global out of its {...} scope except in case of fn.

  let x; x = 'a'
  ```

  Notes:

  - no need to mention the datatype and it can be changed at runtime I.e. **loosly typed language**
  - JS is case sensitive
  - A variable without **let** or **var** is **global out of its scope**

- Note the following 2 cases: where ref. are resolved from nearest to furthest scopes
  - ✓
```
var x = 6
If(true){
        console.log(x) // 6
        var x = 5
        console.log(x) //5
}
console.log(x) //5
```
  - ✓
```
var x = 5
if(true){
   const x = 6;
   console.log(x + 1) // 7
}
console.log(x); //5
```

Constants

Declared with const keyword in any part of code and it is **local to {…} scope**

…Now, let's see **some datatypes** with **their supported operations**

- **String** changes here are by value as it is a primitive data type
  1. Concatenation done by **+ sign** or **template lateral ``**

```
let cost = 25
Console.log("it costs" + cost + "pounds")
        Or
let cost = 25
Console.log(`it costs ${cost} pounds') ${value} called interpolation
```

  2. typeof used with **ALL datatypes** to determine the type e.g.

```
Console.log(typeof 'text'); --> string
```

  3. . length to get the length of the string
  4. . toLowerCase() and . toLocaleLowerCase() usually have the same result, but the latter depends on the **Locale** set in the browser For example, if locale is set to Turkey language, the result of "INTEGER".toLowerCase() will be "ınteger" instead of "integer"
  5. . charAt() to get character at specific index

6. . indexOf() to get the index of a character or -1 if not found
7. . trim() to remove the white spaces from both ends
8. . startsWith(string),. includes(string) return true or false
9. . slice(v1, v2)return a substring from index v1 to index v2 – 1, in case you give it only one argument, +ve -> slice(arg, size) ,and -ve -> slice(size - arg, size)
10. There are many other methods just use dot operator and investigate them

- **number** changes here are by value as it is a primitive data type
  1. **+, -, *, /, =, /=, *=, -=, +=, %, ++, --, Math Object functions**

Notes: the Implicit type conversion, used in many cases such as subtracting two numeric strings **"20" - "15",** the 2 numbers will be converted to two numbers, subtraction done, and the result will remain as number. Some cases of implicit type conversion:

- string **-, *, /** string … result will be Number
- string + number … concatenation
- number + null = number + 0

- **boolean** changes here are by value as it is a primitive data type
  - **conditional Statements:** relative and logical operators similar to other languages but there are more operators as === and !== . these two are not only checking the equality of values but also equality of types
  - not necessary to be explicitly **true** or **false;** as in JS, all values are either **true** or **false** e.g.: **all** the following represent **false**:
    "", **''**, **``**, **0**, **-0**, **null**, **undefined**, **Nan**, **false**
  - **ternary operator** does **not require** preceding **assignment statement**
- **null** the programmer sets it as a value
  changes here are by value as it is a primitive data type
- **undefined** the variable or constant without value
  changes here are by value as it is a primitive data type
- **Symbol**

- **Object : Arrays, functions, objects**

  changes here are by reference as it is not a primitive data type

  - **Arrays**
    - Declared as: const, let,or var arr **=[ ]** or **=[v1, v2, ..]**
    - Can include **different-type** values
    - 0-indexed, accessed via **[ ],** indices are **not necessarily integers**
    - Can be printed as whole in one comsole.log: console.log(arr)
    - Accessing invalid index gives **undefined**
    - **.length** is used to get the array's length
    - **.concat(array)** is used add elements of an array before the current array elements
    - **.reverse()** to reverse the order of elements within an array
    - **.unshift(value)** to insert a value at the beginning of the array
    - **.shift()** to remove a value from the beginning of the array
    - **.push(value)** to insert a value at the end of the array
    - **.pop()** to remove a value from the end of the array
    - **.splice(start, size, optional value)** it *removes* **size** items from the **start** index and *insert* the **optional value.** It mutates the array
    - There are many other functions. Just use the **.** to explore them
    - Some other functions are related to **Functional Programming** such as : ***map, forEach, filter, find, reduce*** .they are also accessed using **. operator** and will be discussed later in the sheet
  - **Functions**
    - Declared as:

```
function fn_name (parameter_name /*optional*/){
                ////// logic
                return …; /// not used in case of void
}
```

    - Passing **more parameters than needed**, will cause the **extra parameters to be ignored**

- Passing **less parameters than needed**, will make the **rest undefined**
- Can be stored in **constants** since logic will not be changed, in this case, you can give a name to the function, and you can neglect this step making the function anonymous
  const fun = function optional name (parameters){logic …}
- Can be written in a form called arrow fn., **also Anonymous.**

  fun = (parameters) => {logic..}

  Notes
  - ✓ Parenthesis can be removed in case of single parameter
  - ✓ In case of one command-logic remove both **return statement** ,and **{}**
- **Higher Order Functions**
  - ✓ Process of passing a function as an argument **called *call-back function*** or returning a function from another one as if it is a first order citizen
  - ✓ It can be passed as an argument as in this example:

```
function morning(name){
    return `Good Morning ${name}`
}
function greet(name, time){
    console.log(`${time(name)}`)
}
```

  - ✓ It can be returned from a function as in this example

```
function add(x){
    function addY(y){      notice accessing x by
        return x + y       closure
    }
    return addY
}
add(5)(4) /// 9
```

  - ✓ Such concept , Higher order functions, will be used in some JS utilities such as *map(), filter(), …etc.* that will be discussed later in the sheet

Note: JS uses **Deep Binding**

- **Objects**
  - a collection of fields called **keys** , and it can have functions declared within
  - Can be declared as :

(1) const object_name = { similar to JSON but with fns.

```
Key1:  value1,
Key2:  value2,
. ,
. ,
Key n: function optional name (parameters) {…}
}
```

Note: Keys must be separated using comma **,**

(2) let obj_name = new Object()

```
obj_name.key1 = value1
.
obj_name.keyn = valuen
```

note the dynamic addition of key-value pairs ; where Object is the default type

Note: the previous two ways may not be suitable in case of complex hierarchies or multiple objects of the same type due to redundancies

(3) function Obj_name(optional parameters){ //called constructor way

```
this.key1 = value1
..
}
```

```
let instance1_name = new obj_name(arguments)
let instance2_name = new obj_name
```

Note:

- the instantiation in last line used if the fn. accepts no parameters
- use function_name.call(this, parameters) to call the object fns

(4) let obj_name{
key1 : value1,
key2 : value2
.
}

let instance1_name = Object.create(obj_name)

Note: this way is based on **prototypes**, prototypes are just pointers to objects to resolve from, OOP of JS are based on them see this link , and this link

Notes :

- Values of keys are accessed using dot operator or square bracket operator specially if the key is a number or a string
- Keys can be deleted using delete obj_name.key_name

- **Control statements**
  - **if statement** and **switch-case** are similar to these in **Java** and **C**
  - **while-loop**, **do-while-loop**, **for-loop** are similar to this in **Java** and **C**, but remember to use variables not constants. It is also preferred to use **let** to make the scope local
- **Some JS utilities related to Functional Programming and Higher Order Functions**
  - **forEach(***call-back fn.***) :** it performs the call-back fn. logic on each item in the array. **But** it does **not return new array** so, **changes are in the original array e.g.**

```
function modify(p){p.age = 10}

let arr = [{name: 'lol', age : 22}, [{name: 'lo', age : 23}]

arr.forEach(modify)

/// arr = [{name: 'lol', age : 10}, [{name: 'lo', age: 10}]]
```

Notes

- you can pass **anonymous** functions or **arrow functions** to **all of these utilities not only *forEach***
- *foreach* can **modify reference variables at certain array index** but to **reassign a variable at certain array index** as whole either **reference or value**, it is **necessary to pass the index together with the item** to the callback fn. e.g.

  let arr = [1, 2, 3]

  arr.forEach(increment, function(num, i){arr[i] = num + 1;})
  //arr = [1, 2, 3]
- the **return value** of the callback fn. is **not received** e.g

  let arr = [1, 2, 3]

  arr.forEach(increment, function(num){return n+1;}) //arr = [1, 2, 3]

- **map(**callback fn**) :**it returns a new array of the same size of the original one and with call-back fn.'s logic performed e.g.
  let arr2=[{name: "lol", age: 29}, {name: "lo", age: 28}]
  let arr3 = arr2.map(function(p){return p.age}); // arr3 = [29, 28]
  Note: the **return value** of the callback fn. is **received** in the corresponding index of the new array
- **filter(**call-back fn.**):** it **reduces** the original array into a **new** one containing **only those satisfying** the **condition given by the call-back fn.**. e.g.

  let arr = [1,2,3,4,5,6]

  let arrF = arr.filter(function(p){return p > 3}) arrF = [4, 5, 6]

- **find(**callback fn.**):** it returns a **single instance** that has satisfied, the **first occurrence**, the **condition given by the call back fn.** if **no item satisfy** the condition, it returns **undefined**

  let arr = [1,2,3,4,5,6]

  let num = arr.find(function(p){return p > 3}) num = 4

so, we can say that: **find(..) == filter(..)[0]**

- **reduce(**callback fn., optional initial value as default is arr[0]**):** this function is so powerful that you can do many functionalities using it. But there are some constraints on the callback fn. such as:
    - ✓ it must receive **2 parameters**, the first is called **accumulator**, the second is called is called **current** …
      current: current item in the original array
      accumulator: will keep the result of calculations within
    - ✓ it must return **accumulator**; as it works recursively

  we can use it for:

    - ✓ summing the array items
    - ✓ getting the min or max item of the array
    - ✓ flattening array of arrays
    - ✓ other uses check this link

1. summing array items

```
let arr = [1,2,3,4]
let result = arr.reduce(function(sum, num){
        sum += num
        return sum;
}) // == 10
```

2. getting array max

```
let arr = [1,2,3,4]
let max = arr.reduce(function(maxi, curr){
        maxi = curr > mxi ? curr  : maxi
        return maxi
}, Number. MIN_VALUE) // 4
```

3. flattening array of arrays

```
let arr = [[1,2,3,4], [5,6,7,8]]
let arrF = arr.reduce(function(flattened, curr){
        flattened = flattened.concat(curr)
        return flattened
}, []) //[1,2,3,4,5,6,7,8]
```

- ## Some JS utilities related global JS objects
  - **Math Object utilities**
    e.g. Math.ceil(num), Math.floor(num), Math.sqrt(num), Math.random(num)//0 ~ 1, …etc
  - **Date Object utilities**
    e.g.
    - ✓ let date = new Date(); //date will be including from milliseconds to the year number
    - ✓ let month = date.getMonth() // index of the month 0 ~ 11
    - ✓ let day = date.getDay() // index of the day 0 ~ 6
    - ✓ let num = date.getDate() // the day number within the month
    - ✓ let year = date.getFullYear() // the year number

    Notes:

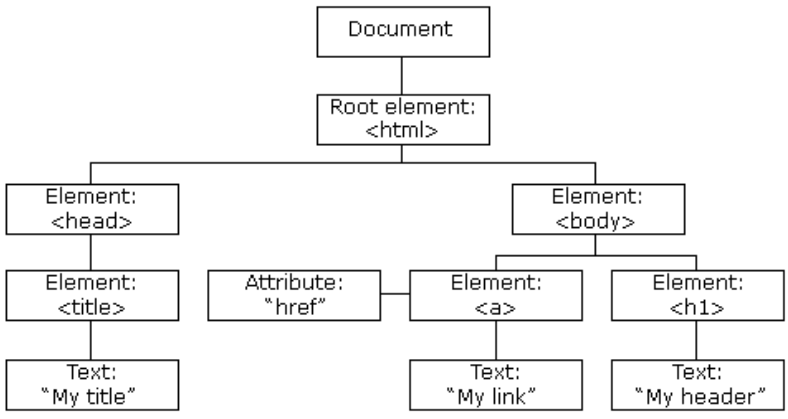
    - since most of these methods return indices, you can use an array to get the real names
    - in new Date(), you can pass a string containing the date as follows `${month_No._1_based} / ${date} / ${year}` inside the () of Date()

- ## DOM: Document Object Model

  Briefly, it enables the developer to deal with the Html and CSS of the web page through responding to interactions, modifying elements, getting elements'values, adding new classes, removing new classes…etc.

  now let's discuss DOM in more details…

  at first, **what is a document?** We can start by the following command; you can try it using developer tools in your browser.
  console.log(window)



such line of code will print the **window object** of your browser containing all info. About the current window**(tab)** such as: toolbar, history button, …etc. **one of the fields of the window object** is the **document object**, that includes the **Html components** of the **current web page** with its **associated CSS.** Such components are called **nodes.** And using **DOM,** you will be able to access these nodes, modify them, get values from them, and respond to interactions

Note:

- Html elements like Html tags are specific type of nodes; where nodes include elements, comment nodes, text nodes, …etc. You can consider them as similar definition, but you must know that there is a difference
- For more clarification, console.dir(object) will make the object more clear than console.log(); displaying its properties well-organized

Some methods to access the Html nodes using DOM:

- **getElementById(‘**element ID**’):** to get/select an element by its ID and decide what to do with it e.g.
  let we have this Html tag : <h1 id=”title”> text < /h1>. To change the color of the text displayed we can write the following JS code

```
let text = document.getElementById(‘title’)
text.style.color = ‘blue’
```

- **getElementByTagName(‘**element tag**’):** similar to **getElementById,** but it returns a node-list, not an array but can be converted, of all elements having the given tag
- **getElementByClassName(‘**element class **’):** similar to **getElementByTagName,** but requires the class name from the Html tag
- **querySelector(‘**tag name**,** class name, or id**’):** but to pass a class put a dot just behind the class name, and to use an ID, put a # just behind the ID. It returns the element in the first index of the array
- **querySelectorAll(‘**tag name**,** class name, or id**’):** it returns an **array** of all elements of a given ID or a given class-name

- **Some methods to navigate the DOM tree:**

  each node in the DOM tree has children as shown in the last figure

  - **childNodes:** returns all the children of a given node including text nodes as whitespaces in the html tags
  - **children:** returns the html child elements of a given node
  - **firstChild:** returns the first child element of a given node (usually a text node)
  - **lastChild:** returns the last child element of a given node (usually a text node)
  - **parentElement:** returns the parent node of a given node
  - **previousSibling:** returns the previous sibling node of a given node
    .previousSibling.previousSibling = .previousElementSibling
  - **nextSibling:** returns the next sibling node of a given node
    .nextSibling.nextSibling = .nextElementSibling

  e.g. let we have an unordered list in the html section of ID = "lista". And we want to access all the children and make their color red

  ```
  let lista = document.querySelector('#lista')

  lista.children.style.color = 'red'
  ```

- **Some methods to access node's contents and properties**
  - **nodeValue:** to get the value of a text node **directly**
  - **textContent:** to get the value of text node that is a **child of a given node**
  - **getAttribute(**'attribute_name'**):** to get the value of a certain attribute from a node
  - **setAttribute(**'attribute_name', 'attribute_value'**):** to set a value to a certain attribute of a node
  - **className:** to get and access the className of a certain node

    ```
    let btn = querySelector('button')
    btn.className('button1')
    btn.className('button2')   /// override all the previous!!
    ```

Note: if you do not want to override the previous className assignments you can set them as follows to combine the effect of the classes or use classList which is more dynamic

```
btn.className('button1  button2  button3  …')
```

- **classList:** using its methods: add(), contain(), and remove(), you can add or remove classNames to your node at any time
  Note: these methods can add, or remove multiple classNames in one line. Just separate the class names by commas

```
btn.classList.add('button1', 'button2', 'button3')
```

- **Adding new element dynamically to the document**
  There are different methods to do so:
  - **createElement(**'element'**):** creates an element but will not render it
  - **createTextNode(**'text content'**):** creates a text node, like a paragraph but will not render it
  - **element.appendChild(**childElement**):** to append the created elements in the document structure to be rendered
    let we have a div of id called oldDiv containing a <h1> tag of ID = head, and we want to add another elements to it like

```
let old = document.querySelector('oldDiv')
let new = document.createElement('h1')
let text = document.createTextNode('hello')
new.appendChild(text)
old.appendChild(new)
```

  - **element.prepend(**childElement**):** to insert before the first child
  - **element.insertBefore(**childElement, nextChildElement**):** similar to the previous one, but will insert the element before another one in a certain location in the DOM tree

```
document.body.insertBefore(new, old)
```

  - **element.replaceChild(**childElement, replaced ChildElement**)**: to replace an element with a new one in a given location in the DOM tree
  - **element.innerText:** to access the inner text of an element. used **instead of** creating textNode then appending it

- **element.innerHtml:** to access the inner Html of an element. Used to set children of certain element

– **Removing an element dynamically from the document**
  - **element.remove():** to remove the element itself though it would be still found statically in the Html
  - **element.removeChild(**childElement**):** to remove a child element of a certain element

- ## Event Handlers

You can listen to the events occurring on html elements in 2 ways

1. **add the handler in the html section (inline)** … not a good approach as in the next example, if you write onclick='doStar()' the on-Click is overridden to perform the new function doStar() and **not** do()

```
<button onclick='do()'> click here </button> /// html
```

2. **add the handler using JS using** … repeating the next line will not override but will do all the functions **not only the last**

element.addEventListener(event,function(normal fn/anonymous fn/**=>** fn))

```
<button> click here </button> ///html
let button = document.querySelector('button') //JS
button.addEventListener('click', do())   //JS
```

– **Some Mouse Events**
  - ✓ **click:** when you click on sth.
  - ✓ **mousedown:** when the mouse is pressed on sth.
  - ✓ **mouseup:** when the mouse is released after being pressed
  - ✓ **mouseenter:** when you move the mouse to enter some region
  - ✓ **mouseleave:** when you move the mouse out of certain region
  - ✓ **submit:** when you click the submit button of a form

    Notes:
    - click = mousedown + mouseup
    - ehover = mouseenter + mouseleave

– **Some Key Events**
  - ✓ **keypress**: key down + key up
  - ✓ **keydown:** when the key is down
  - ✓ **keyup**: when the key is up

Notes: on dealing with text-input areas in Html, you will find the text typed in element.value

- **Some Event Properties**
  - ✓ **event.currentTarget:** to get the element on which the event handler is attached. You can use **this** keyword instead
    this == event.currentTarget
  - ✓ **event.target:** to get the element on which event occur
    Note: the difference between these two methods will be clear in case of **compound element such as a div with a h1 inside**. try clicking the h1 tag and inside the div with target and currentTarget
  - ✓ **event.type:** to get the type of the event occurred
  - ✓ **event.preventDefault():** useful in case of forms where sometimes, you do not want the submit button to go to the document specified by the default link in the html
  - ✓ **event.keyCode:** to get the ascii code of the key pressed in key events
  - ✓ **event.stopPropagation():** discussed below
- **There are 3 ways in which event are processed**
  - **Event bubbling (default):** the event is handled at the target up in the hierarchy, till it reach the handler at the window
    Note: to stop propagation at any handler use **event.stopPropagation()** at the handler you think the event is fully handled
  - **Event Capturing:** the event is handled first at higher hierarchies down to the specified target. Set as:
    element.addEventListener(event, fn., {capture : true})

- **JSON Objects in JS**

  Defined in a way similar to the 1st way of defining objects but each key is surrounded by "" as well as the value of this key
  - There are some methods to deal with JSON objects
    - **JSON.parse(**json object**):** convert JSON object to JS object
    - **JSON.stringify(**json object**):** convert JS object to JSON object