Data Types in JavaScript

Important Built in Data Types in Javascript:

Number: Represents numeric values. It can be integers or floating-point numbers. For example: let age = 25;

String: Represents textual data. It is enclosed in single or double quotes. For example: let name = "John";

Boolean: Represents a logical value that can be either true or false. It is often used for conditional statements and comparisons. For example: let isStudent = true;

Undefined: Represents a variable that has been declared but has not been assigned a value. When a variable is declared without initialization, it is automatically assigned the value undefined. For example: let city;

Null: Represents the intentional absence of any object value. It is often used to indicate that a variable has no value or that an object does not exist. For example: let person = null;

Object: Represents a collection of key-value pairs, where each value can be of any data type. Objects are created using curly braces {}. For example:

```
let person = {
   name: "John",
   age: 25,
   isStudent: true
};
```

Array: Represents an ordered collection of values, called elements. Arrays are created using square brackets [] and can hold values of any data type.

The array elements In javascript can be of any datatype unlike in java or c++ where arrays should have homogeneous data types.

```
let arr1 = [
   1,
   true,
   "string",
   undefined,
   {
       name: 'aravind'
   }
]
```

Arrays

Methods and properties on arrays:

```
arr1.length => Gives the number of elements in the array.

arr1.pop() => removes the element at the last index and returns the removed element.

arr.push(element1, element2 ....) => adds the elements at the lastand returns the updated length of the array.Arrays

arr.unshift() => removes the element at the starting index

arr.shift(element) => adds the element at the starting index
```

```
arr.slice(start, end) => returns sub array containing all the elements in between the indices [start, end) remember element at end index will not be included.

arr.splice(start, deleteCount, e1, e2, e3) => splice method is used to add/delete elements at required positions in the array. deletes the `deleteCount` number of elements starting at index `start` and adds e1,e2 and e3 elements at the same place.
```

forEach , map, filter & reduce

forEach method:

```
let arr = [4, 5, 6] ;
function forEach(x) {
   for(let i = 0 ; i < arr.length; i++){
       x(arr[i], i, arr);
   }
}
let something = function(element, index, list){
   console.log(element, index, list)
}
forEach(something)
```

- forEach method doesn’t return anything.

Map Method:

```
let arr = [5, 6];
function map(callback){
   let aggrigatedArray = []
   for(let i = 0 ; i < arr.length; i++){
       let result = callback(arr[i], i, arr);
       aggrigatedArray.push(result);
   }
   return aggrigatedArray;
}
let output = map(function(element, index, list){
   return element + index ;
});
console.log(output) // [5, 7]
```

- Map method always returns an array of same length.

Filter Method:

Filter method is used to filter out some elements from the original Array.

```
The Filter method always returns an array of any
length(possibly 0 as well).
```

```
let arr = [5, 6, 4, 7];
function filter(callback){
   let aggrigatedArray = [];
   for(let i = 0 ; i < arr.length; i++){
       let result = callback(arr[i], i, arr);
       if(result === true){
           aggrigatedArray.push(arr[i]);
       }
   }
   return aggrigatedArray;// [6, 4]
}

let output = filter(function(element, index, list){
   return (element % 2 === 0);
})
console.log(output) // [6, 4]
```

Reduce Method:

- The Reduce method always returns an aggregated value of any data type.

```
let arr = [5, 3] ;
function reduce(callback, intialValue) {
   // initialValue = 10
   let sum ;
   let intialIndex ;
   if(intialValue === undefined){
       sum = arr[0];
       intialIndex = 1 ;
   }
   else{
       sum = intialValue;// sum = 10
       intialIndex = 0 ;
   }
   // sum = 10 => 15 => 19
   for(let i = intialIndex ; i < arr.length; i++) {
       sum = callback(sum, arr[i], i);
   // when i = 0 ;  sum =  callback(10, 5, 0) = 15
   // when i = 1 ; sum = callback(15, 3, 1) = 19
   }
   return sum ; // 19
}
let output = reduce(function(prev, element, index){
   return prev + element + index
}, 10) ;
```

```
console.log(output) // 19
```

Objects

Scope Chain & Closures:

```
/*
Scope chain is a crucial concept in JavaScript that
determines the accessibility and visibility of variables and
functions within a program. It refers to the hierarchy of
nested scopes and how JavaScript resolves identifiers
(variable and function names) when they are referenced.

To understand the scope chain, let's consider some examples:
*/


Example: 1
function outer() {
 var x = 10;
  function inner() {
   var y = 20;
   console.log(x + y);
 }
  inner();
```

```
}

outer();

/*
* In the above example, we have two nested functions: outer and inner.
* The variable x is defined in the outer function, and the variable y is defined in the inner function.
* When we call the inner function from within the outer, it tries to access both x and y to perform the addition. JavaScript follows the scope chain to resolve the variables.
*  It first checks the local scope of inner, finds y, and then moves to the outer scope of outer, where it finds x. Thus, it successfully calculates and logs the result, which is 30.
*/
```

Example 2:

```
function outer() {
 var x = 10;
  function inner() {
   console.log(x);
 }
```

```
  return inner;
}

var closureFunc = outer();
closureFunc();

/*
* In this example, the outer function returns the inner function.
* We store the returned function in the variable closureFunc and then invoke it.
* The interesting part is that even though the outer function has already finished executing and its local variable x is technically out of scope, the inner function still has access to it. This is due to the concept of closure.
* When the inner function is returned and assigned to closureFunc, it carries along with it a reference to its parent scope, which includes the variable x. Thus, when we call closureFunc(), it logs the value of x, which is 10.
*/
```

Example 3:

```
function outer() {
 var x = 10;
  function inner() {
   var y = 20;
   function deep() {
     var z = 30;
     console.log(x + y + z);
   }
   deep();
 }
  inner();
}
outer();

/**
* This example demonstrates multiple levels of nested
functions.
```

```
* The deep function is nested within the inner function, which itself is nested within the outer function.
* Each function has its own set of locally defined variables (x, y, and z). When deep is called from within the inner, it performs the addition of x, y, and z.
* JavaScript follows the scope chain and successfully resolves all the variables, resulting in the output 60.
* Understanding the scope chain is essential for properly managing variables and functions in JavaScript.
* It ensures that the correct values are accessed and prevents naming conflicts between different scopes within a program.
*/
```

DOM

What is DOM & It's Usage

```
DOM : Document Object Model
Representation of HTML structure in the form a Tree (Object)
```

What we can do using DOM(document):

window is the global object when we run javascript in browsers.

window object contains a lot of properties and methods. ex: setTimeout, alert, document etc..

document is the object present inside the window object which manages the entire HTML, CSS of a particular web page .

Example: The title of the web page can be modified by using

```
document.title = "new title" ;
```

Important point :

Whatever the HTML elements(span, div, button etc) that you have inside a webpage behind the scenes they are javascript objects(Non-primitive), stored in the HEAP memory.

document object(API) is used to retrieve, update, delete, create HTML elements dynamically.

1. Retrieval

Extracting data/behavior of an HTML element(remember it's a javascript object stored in memory behind the scenes)

example:

i) extracting the value present inside an input element.

ii) extracting the innerText of an element.

iii) extracting color of the text etc...

2. Update

Updating/Modifying the data/behavior of an HTML element.

example:

i) Changing the innerText of an element.

ii) Changing the CSS styling of an element etc...

3. Delete

Removing/Deleting an element from the DOM tree structure(which eventually removes from the UI as well).

```
          example:
           i) Removal of a Popup/modal/dialog-box from the
UI.

4. Create
          Remember that all the HTML elements behind the
scenes are javascript objects stored in memory HEAP.Creating
a new HTML element object's reference in the HEAP and adding
it to the DOM tree.

          example:
             i) Adding the input elements conditionally.
             ii) Showing up a submit button on entering all
the fields. etc...
```

Retrieval:

- getElementById:
  Retrieves an element using its unique ID attribute.
- getElementsByClassName:
  Returns a collection of elements that have a specific class name.
- getElementsByTagName:
  Returns a collection of elements that have a specific tag name.

- querySelector:
  Returns the first element that matches a specified CSS selector.
- querySelectorAll:
  Returns a collection of elements that match a specified CSS selector.
- parentNode:
  Accesses the parent element of a given element.
- childNodes:
  Retrieves a collection of child nodes (including elements, text nodes, etc.) of a given element.
- children:
  Retrieves a collection of child elements of a given element (excluding non-element nodes).
- nextSibling:
  Accesses the next sibling element of a given element (includes text nodes).
- previousSibling:
  Accesses the previous sibling element of a given element (includes text nodes).
- nextElementSibling:
  Accesses the next sibling element of a given element.
- previousElementSibling:
  Accesses the previous sibling element of a given element.

Examples:

```
<div id="prev-sibling"></div>
   <div id="parent">
       <span>Span text</span>
       <div class="child-class" id="child">
           <p>Para</p>
           <div id="inner">
           </div>
       </div>
   </div>
   <div id="next-sibling"></div>

const parent = document.getElementById("parent");

const classElements =
document.getElementsByClassName("child-class");
     // getElementsByClassName always returns a
collection/list/array of HTML elements.

     const tagElements =
document.getElementsByTagName("div")
     // getElementsByTagName always returns a collection of
HTML elements.
```

```
console.log(document.querySelector("div:nth-child(1)"))//
div#prev-sibling


console.log(document.querySelectorAll("div:nth-child(2)"))
    // [div#parent, div#child-class, div#inner]

    console.log(parent.nextSibling); //  a Text node.

    console.log(parent.nextElementSibling) //
div#next-sibling element

    console.log(parent.previousSibling) // a Text node.

    console.log(parent.previousElementSibling) //
div#prev-sibling

    console.log(parent.children) // [span, div]

    console.log(parent.childNodes) // [text, span,text,
div, text]

    const child = parent.children[1] //
div#chuld.child-class
```

```
    console.log(child.parentNode) // div#parent
```

Promises

A Promise is an object(instance if built in Promise class/constructor function) which represents the eventual completion or failure of a task.

A promise Object has three possible states

i) pending (initial state)
ii) fulfilled (when the promise is resolved)
iii) rejected (when the promise is failed)

Every promise object also holds a data field.

Behavior of Promise class / constructor function :

1. Promise class constructor takes a function usually called as executor.
2. The executor method will be immediately invoked by the Promise constructor as soon as we create an object.
3. The Promise constructor method passes two functions(usually called as resolve and reject) to the executor function.
4. Whenever we call the resolve the state of the promise changes from pending -> fulfilled .
5. Whenever we call the reject the state of the promise changes from pending -> rejected .

Upon promise's state change maybe from pending -> fulfilled or pending -> rejected we have a feature to pass callbacks to be executed

I.e when the state changes from pending -> fulfilled we can execute a function `successCallback`
Similarly upon state change from pending -> rejected we can execute a function `errorCallback`.

Promise class internally has two methods `then` and `catch` which are used to attach the successCallback and errorCallback respectively.

```
let prom = new Promise((resolve, reject) => {
            setTimeout(() => {
                resolve({name: "Aravind"})
            }, 1000)
        })

        let successCallback = (data) => {
            console.log(data ) // {name: "Aravind"}
        }

        let errorCallback = (error) => {
            console.log(error)
        }

        prom.then(successCallback);
        prom.catch(errorCallback);
```

In the above example the promise will be resolved after 1s.

By the time the resolve method is called it pushes the `successCallback` method into the microtask queue. Same thing goes with reject & errorCallback as well.

While calling resolve/reject we can pass an argument which will be binded to the promise object as data field, that data will be captured by the successCallback/errorCallback function as parameter.

Promise chaining

In progres…

Promise.all, Promise.any, Promise.race

Async/ Await

```
async function callme() {
 console.log("before first promise")
 let data1 = await new Promise(resolve => {
   setTimeout(() => resolve({name: "aravind"}), 3000)
 })
 console.log("after first promise")
 let data2 = await new Promise(resolve => {
   setTimeout(() => resolve({name: "shubh"}), 3000)
 })
 console.log("after second promise")

 return [data1, data2]
}
let x = callme();
```

```
x.then((result) => {
 console.log("async function execution is done",
result)
})
```