Basics in JavaScript

Index

Lexical Structure: Difference between let and var

- var properties:
  - It has function scope (this means that, you can reference a variable outside the block where it has been defined (supposing you're inside the same function))
  - Variable declaration will be hoisted, initialized as undefined (this means, you can reference a variable before declare it and you will not get any error)
  - It is possible to re-declare the variable in the same scope (this means, you can redeclare a variable with same name)
- let properties:
  - It has block scope
  - Hoisted but not initialized
  - It is not possible to re-declare the variable

Lexical Structure: Difference between let and var

```
function userDetails(username) {
  if (username) {
    console.log(salary); // undefined due to hoisting
    console.log(age); // ReferenceError: Cannot access 'age' before initialization
    let age = 30;
    var salary = 10000;
  }
  console.log(salary); //10000 (accessible due to function scope)
  console.log(age); //error: age is not defined(due to block scope)
}
userDetails("John");
```

Lexical Structure: Strict mode and Script mode

- Strict mode is useful to write "secure" JavaScript by notifying "bad syntax" into real errors. For example, it eliminates accidentally creating a global variable by throwing an error and also throws an error for assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object.
- The strict mode is declared by adding "use strict"; to the beginning of a script or a function. If declared at the beginning of a script, it has global scope.

Lexical Structure: Double exclamation

- The double exclamation or negation (!!) ensures the resulting type is a boolean. It's the two time application of the negation operator

```
!!false // -> false
!!true // -> true
!!0 // -> true
!!1 // -> true
!!NaN // -> false
!!'hello' // -> true
```

Lexical Structure: Temporal Dead Zone

- Temporal Dead Zone refers to the period between the entering of a scope and the actual declaration of a variable using let or const
- This will rise an undefined value, if the variable is declared as var
- This will rise a ReferenceError, if the variable is declared as let
- Solution: declare all variable at the beginning of the scope, as in C-style

General Structure: Primitive Types and Object types

In JavaScript there are two types of values:

- Primitive Types: String, Number, BigInt, Boolean, Symbol, Undefined, Null
- Object type: list of properties (key-value pair), it's mutable and it is created by literals, classes and functions: Object, Array, Function, Map, Date, RegExp, Set, Custom Class Instance
- primitive types are passed by value, object type are passed by reference

General Structure: Numbers

- Integer Literals
- Floating-Point Literals
- Arithmetic in JavaScript
- Binary Floating-Point and Rounding Errors

General Structure: Dates and Times

General Structure: Text

- String Literals
- Escape Sequences in String Literals
- Pattern Matching

General Structure: Boolean Values

General Structure: null and undefined

General Structure: The Global Object

General Structure: Wrapper Objects

General Structure: Immutable Primitive Values and Mutable Object Refereces

General Structure: Type Conversions

- Conversions and Equality
- Explicit Conversions
- Object to Primitive Conversions

General Structure: Variable Declaration

- Repeated and Omitted Declarations

General Structure: Variable Scope

- global variables has global scope
- variabled declared within a function are defined only within the body of the function
- Function parameters also are local variables
- Local variable hides global variables with the same name

General Structure: Var, Let and Const

- var: Declares variables with function or global scope and allow re-declaration and updates within the same scope (to be avoided)
- let: Declares variable with block scope, allowing updated but not re-declaration within the same block
- const: Declares block-scoped variables that cannot be reassigned after their initial assignment

General Structure: Literals

- A literal is a data value that appears directly in a program

```
// These are all literals
var a = 12;
var s = "hello world";
var o = {x:1, y:2};
var ar = [1,2,3,4]
```

General Structure: Variable Declaration

- var variable declaration
-

Regular Expression in JavaScript

Regular Expression: Base syntax

```
const regex = /pattern/flags;
const regex = new RegExp("pattern", "flags");
```

Regular Expression: Flag

- "g": global flag: find all matches in text
- "i": ignore case: find all matches ignoring the case
- "m": Multiline: find all matches for every line and the character $ and ^ are applied to every line, not at beginning or the end of the text

Regular Expression: Basic Methods

- test(): return true or false if there is a match or not
- match(): retun the match found in the string
- replace(): find and substitute the matching text
- split(): return an array obtained splitting the text by matches

Regular Expression: Common Special Characters

- ".": any chars
- "^": start of the string
- "$": end of the string
- "*": zero or more time
- "+": one time or more
- "?": zero or one time
- "{n}": exactly n times characters

Regular Expression: Common Special Characters (2)

- \d: any digit
- [abc]: one of the characters
- [^abc]: none of the
- \d: any digit
- \w: any word
- \s: space, tab or newline
- : escape character

Regular Expression: Generic Example

```
const email = "test@email.com";
const regex = /^[\w.-]+@[\w.-]+\.\w+$/;
console.log(regex.test(email)); // true
```

Array in JavaScript

1. forEach, Map, Filter
2. Destructuring Array
3. Rest Operator

Array: Adding and Deleting Array Elements

- to add an element use the push function:

```
a = [];              // Start with an empty array
a.push("zero");      // add a value at the end. a = ["zero"]
```

- to delete an element use the delete keyword

```
a = [1,2,3];
delete a[1];    // a now has no element at index 1
1 in a          // => false: no array index 1 is defined
a.length        // => 3: delete does not affect array length<>
```

Array: Adding and Deleting Array Elements (2)

- You can also remove an element by reducing the length property
- pop() (push()) method return the element deleted
- shift() (unshift()) remove an element from the beginning of an array

Multidimensional Array

```
// One example
var table = new Array(10);
table[0] = new Array(10);
// In this case, the table is not necessary a matrix, every row can have different length

// Initialize the array
for (var row = 0; row < table.length; row++) {
    for (col = 0; col < table[row].length; col++) {
        table[row][col] = row*col;
    }
}
```

Array: Useful methods

- join()
- reverse()
- sort()
- concat()
- slice()
- splice()
- push() and pop()
- unshift() and shift()
- toString() and toLocaleString()

ECMAScript 5 Array Methods

- forEach()
- map()
- filter()
- every() and some(): true if every element satisfy the condition, true if some of these
- reduce(), reduceRight(), start from the beginning or start from the last element
- indexOf() and lastIndexOf()

Array: Array Type

- Useful function to determine if a variable is an array or not

```
Array.isArray([]) // => true
Array.isArray({}) // => false
// The instance of operator works in simple cases:
[] instanceof Array // => true
({}) instanceof Array // => false
// The best way to determine this remains the isArray() function
```

Array: Array-Like Objects

- sometimes is useful create an object to be used as an array, e.g. :

```
var a = {"0":"a", "1":"b", "2":"c", length:3};

// Now iterate through it as if it were  a real array
for (let j = 0; j < a.lenght; j++)
    console.log(a[j]);
```

- In client-side JavaScript, a number of DOM methods, such as document.getElementsByTagName(), return array-like objects

Array: Strings As Arrays

- In ECMAScript 5 strings behave like read-only arrays. Instead of accessing individual chracters with the charAt() method tou can use square bracket:

```
var s = test;
s.charAt(0) // => "t"
s[1]        // => "e"
```

- the typeof operator still returns "string" for strings
- the Array.isArray() method returns false if you pass as string
- the benefit of using String as Array is to use charAt with square bracket and the possibility to use generic array functions on String
- But string are immutable and also strings as array behave in this way. So push, pop, sort, reverse and splice doesn't work and they don't returns an errors, simply fail silently

Array: Destructuring Array

```
const numbers = [10,20,30];

// Destructuring
const [a,b,c] = numbers;

console.log(a); // 10
console.log(b); // 20
console.log(c); // 30
```

Array: Destructuring Array (2)

```
const numbers = [10,20,30];

// Destructuring
const [first, , third] = numbers;

console.log(first); // 10
console.log(third); // 30
```

Array: Destructuring Array (3)

```
const number = [5];
const [x,y = 99] = number;

console.log(x); // 5
console.log(y); // 99
```

Array: Rest Operator and Spread Operator

Rest operator is used to collect data into arrays: Values -> Array

```
function sumRest (...numbers) {
    console.log(numbers);
}
sumRest(1,2,3);
```

Spread operator is used to spred data from array: Array -> Values

```
function sumSpread (numbers) {
    console.log(...numbers);
}
sumSpread([1, 2, 3]);
```

Array: common methods on Array

- join
- reverse
- sort
- concat
- slice
- splice
- push and pop
- shift and unshift
- toString and toLocaleString

Map in JavaScript

```
const map = new Map()

map.set("a", 1);
map.set("b", 2);
map.set("c", 3);

console.log(map.get("a"));
console.log(map.size);
map.delete("b");
console.log(map.size);
```

Object in JavaScript

1. Basic feature of an object
2. Creating an Object
3. Destructuring an Object

Object in JavaScript: Spread Operator and Rest Operator

```
const user = {
    name:"john",
    surname:"doe",
    age:30,
};
const user_acces { access_time: [], access_ops: []; }
// copy (deep) w/ spread op
const copy_user = {...user};
// merge w/ spread op
const user_data = {user, user_data};
// overwrite props
const updated_user = {...user, age:31};

// Destructuring with rest op
const {nome, ...data} = user;
```

Object in JavaScript: Constructor vs Function

Function and Constructor are very similar in JavaScript, take a look at this example

```
function Person(name) {
    this.name = name;
    this.sayHello = function() {
        console.log("Hi, I'm " + this.name);
    };
}
const p1 = new Person("Anna");
p1.sayHello();
```

Main differences from a fuction: Capitalized name, this keyword, called with new

Object Section: How to Create an Object

- Create it as a Object Literal
- Call a pseudoclassical constructor with the `new` operator
- Call Object.create method on a prototype object
- Call a functional constructor

Object: Enumerating Properties

- to know about all of the properties inside an object, one can iterate over them with a for/in loop
- there exists also other utility function that can be useful, like:
  - extend(o,p): copy props from p to o
  - merge(o,p): copy props from p to o
  - restrict: filter out some properties of o
  - subtract: for each property of p, delete the property with the same name from o
  - union: return a new object that holds the properties of both o and p
  - intersection: return a new object that holds only the props of o that also appear in p
  - keys: return an array that hold the names of the enumerabl own props of o

Object: Serializing Objects (JSON)

- JSON.stringify: transform an object to a json string

```
s = JSON.stringify(o)
```

- JSON.parse: parse a string into an object

```
p = JSON.parse(s);
```

- JSON ops apply only to enumerable properties of an object, not to all of that

Generator in JavaScript

Iterator in JavaScript

Class and OOP in JavaScript

1. Class
2. Prototype
3. Constructors
4. Types
5. Subclasses
6. Modules
7. Augmenting Classes

Class and OOP: Class

- Class is not an object, it is a template for object
- Constructors work as in Java

Abstract Example of JavaScript Class

```
class ClassName {
    constructor() {...}
    method_1() {...}
    method_2() {...}
    method_3() {...}
}
```

Concrete Example of JavScript Class

```
class Car {
    constructor(name, year) {
        this.name = name;
        this.year = year;
    }

    age(x) {
        return x - this.year;
    }
}

const myCar = new Car("Ford", 2014);
```

Classes are always executed in "strict mode", never in script mode.

Class and OOP: Inheritance

- Inheritance is based on prototype chain (prototype-based inheritance)
- Every object has its own prototype
- To find the method to be executed on a certain object, explore the prototype chain to the null element
- Class is sintactic sugar for prototype. Behind a class definition there is always a prototype

Class and OOP: Prototype

- actual version of the class definition
- Every object is linked to a prototype object from which it can inherit properties
- All object literals are linked to Object.prototype
- When you make a new object, you can select the object that should be its prototype
- Prototypes create the prototype chain, on which is applied the delegation process

Class and OOP: Reflection and Enumeration

- The `for in` statement can loop over all of the property names in an object
- The `for in` loop will include all of the properties and funcitons of the object and of the prototypes
- to filter out some property and/or function use the hasOwnProperty Method and typeof (=== 'function')

```
const person = {
    name: "Tom",
    age: 30
}
console.log(person.hasOwnProperty("name")); // true
```

Class and OOP: Delete

```
delete another_stooge.nickname;
```

This will remove the nickname only for the object on which is invoked, not on to the prototype

Class and OOP: Global Abatement

Don't use global variables, instead put it into a map

```
var MYAPP = {};
MYAPP.stooge {
    "first-name": "Jow",
    "last-name": "Howard"
};

MYAPP.flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
```

Class and OOP: Subclass Usage

- Inside the constructor you can call super()
- You can use extends keyword to extend one class
- You can use this keyword to referencing the attribute and the method of the class

Class and OOP: Modules

- JavaScript modules allow you to break up your code into separate files
- Modules are imported from external files with the import statement
- You can export (and then import) functions and variables in two ways: Named Exports and Default Exports

Class and OOP: Modules (Named Exports)

- An example of Named Exports:

```
export const name = "Jesse";
export const age = 40;
```

or all at once:

```
const name = "Jesse";
const age = 40;
export {name, age};
```

Class and OOP: Modules (Default Exports)

- An example of Default export:

```
const message = () => {
    const name = "Jesse";
    const age = 40;
    return name + ' is ' + age + 'years old.';
};
export default message;
```

Class and OOP: Modules (Import)

- Importing a named export modules:

```
import {name, age} from "./person.js"
```

- Importing from a default export:

```
import message from "./message.js"
```

Class and OOP: Augmenting Classes

- An object inherits properties from its prototype, even if the prototype changes after the object is created
- Augment JavaScript classes simply by adding new methods to their prototype objects
- The prototype object of built-in JavaScript classes is "open", which means that we can add methods to numbers, strings, arrays, functions, and so on

Class and OOP: Classes and Types

- typeof: operator that allow to distinguish among built-in types (null, undefined, boolean, number, string, function and object)
- classof(): access to the class attribute of Object
- class attribute of an object is not modifiable and for your own custom class is always `'Object'`, so
- classof() doesn't work for own-defined class, in these case use one of the following methods: instanceof, constructor property, constructor function name, duck-typing philosophy

Class and OOP: Classes and Types (instanceof op)

- the expression o instanceof c evaluates to true if o inherits from c.prototype
- it doesn't work with primitive type
- One shortcoming is: by instanceof we can test if an object is instance of a certain class, but we cannot derive the class of the object by itself

Class and OOP: Classes and Types (Constructor)

- Another way to identify the class of an object is to simply use the constructor property, that is the public face of the class
- One shortcoming is: JavaScript does not require that every object have a constructor property, sometimes it is accidentally omit, sometimes intentionally
- Sometimes can be useful get the name of the construct instead of the type but not all object have a constructor name defined and not all object have a constructor function with a nam

Class and OOP: Classes and Type (Duck-Typing)

- not ask "what is the class of this object?", instead try asking "what can this object do?"
- The general idea is: look if a certain object have a certain method or a certain property, without knowing if it is of a certain type
- this is the same concept of implementing an interface (e.g. implementing a functionality) instead of extending a class

Class and OOP: Object-Oriented Techniques in JavaScript

- Encapsulation: private attribute with `#`
- Inheritance: with the extends keyword
- Polimorphism: overloading method
- Abstract: throwing exception

Class and OOP: Object-Oriented Techniques (Polymorphism)

```
class Animal {
    name = undefined;
    constructor(name) {
        this.name = name;
    }
    makeSound() {
        console.log("Need to be implemented!");
    }
}

class Tiger {
    constructor(name) {
        super(name);
    }
    makeSound() {
        console.log("Roar!");
    }
```

Class and OOP: Object-Oriented Techniques (Abstract class and)

```
class Animal {
    constructor() {
        if (new.target === Animal) {
            throw new Error("Animal class is abstract!");
        }
    }

    makeSound() {
        throw new Error("Method to be implemented!");
    }
}

class Tiger {
    constructor() {
    }

    makeSound() {
        console.log("Roar!");
```

Class and OOP: Object-Oriented Techniques (Encapsulation)

```
class Person {
    #name = undefined;
    _surname = undefined;

    constructor (name, surname) {
        this.#name = name;
        this._surname = surname;
    }
}
```

The _underscore notation makes the attribute private by convention, but publicly accessible. The #hashtag notation makes the attribute purely private as other OOP language.

Class and OOP: Object-Oriented Techniques (static)

- Example of static method

```
class MathUtils {
    static somma (a, b) {
        return a + b;
    }
}
console.log(MathUtils.somma(2, 3)); // 5
```

Class and OOP: Object Specifiers

Better way of writing constructors:

```
// One way
var myObject = maker(f,l,m,c,s);
// Better way
var myObject = maker({
    first: f,
    last: l,
    state: s,
    city: c
})
```

Class and OOP: Standard Conversion Methods

- overwrite the toString method always
- overwrite also the toLocaleString(), if you do the toString() method
- valueOf() convert an object to a primitive value: do it only if a natural representation of the object is possible
- toJSON() serializes the object in a JSON file

Class and OOP: Comparison Methods

Class and OOP: Borrowing Methods

Class and OOP: Private State

Class and OOP: Constructor Overloading and Factory Methods

Class and OOP: Composition Versus Subclassing

Class and OOP: Class Hierarchies and Abstract Classes

Function

Function

1. Function Properties, Method and Constructor
2. Recursion
3. Scope
4. Closure
5. Callbacks
6. IIFE
7. Functional Programming

Function: Hoisting

- Hoisting is the mechanism by which you can referred to the function before its declaration

Function: Function Properties, Method and Constructor

- We say that in JavaScript Function are value, and so they have Constructor, Properties and Method
- Constructor: (not popular)

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

const myFunc = new Function('a','b','return a + b');
```

Function: Function Properties, Method and Constructor (2)

- Properties:
  - .length: number of parameters
  - .name: name of the function (void if anonymous)
  - .arguments: array of the arguments of the function
- Custom properties: you can add properties to a function

```
function greet() {}
greet.customProp = "Hello!";
console.log(greet.customProp); // "Hello!"
```

Function: Function Properties, Method and Constructor (3)

- Method of the function:
  - call: call a function on an object
  - apply: as call, but arguments in the array
  - bind: a function to an object
  - toString: return the body of the function

Function: Function Properties, Method and Constructor (Call and Apply)

```
f.call(o, 1, 2);
f.apply(o, [1,2]);

// does the same thing to:
o.m = f;
o.m([1,2]); // o.m(1,2) in the case of call
delete o.m;
```

Function: Function Properties, Method and Constructor (Bind)

- The bind method bind a function f to an object o
- When you invoke bind on function, it will return a new function f, method of o
- Any arguments you pass to the new function are passed to the original function, but performing partial application

```
function sayHi(name) {
    console.log(`Hi, ${name}`);
}

sayHi.call(null,"Jack"); // "Hi, Jack"
sayHi.apply(null,["John"]); // "Hi, John"

const bound = sayHi.bind(null, "Tom");
bound() // "Hi, Tom"
```

Function: Function Properties, Method and Constructor (4)

- The arrow function are lightweight function:
  - they don't have `this`, `arguments`, `super` nor `new.target`
  - they cannot be use as `constructor`
  - the don't have `prototype`

Function: Function Properties, Method and Constructor (5)

- new.target is an attribute that is `true` if the function has been called with the `new` keyword, false otherwise
- this attribute can be used to ensure a function is called with the `new` keyword, as a constructor. This is the right way of realize a constructor by function
- this works also with class constructor

```
function MyObject() {
    if (!new.target) {
        throw new Error ("This function cannot be called without `new`!");
    }
    return myObject();
}
new MyObject(); // the correct way
MyObject(); // the wrong way
```

Function: Augmenting Types

- JavaScripts allows the basic tyes of the language to be augmented
- Adding a method to Object.prototype makes that method available to all objects
- This also works for functions, arrays, strings, numbers, regular expressions and booleans

Function: Encode or decode URI

- encodeURI() function is used to encode an URL. This function requires a URL string as a parameter and return that encoded string. decodeURI() function is used to decode an URL. This function requires an encoded URL string as parameter and return that decoded string.

```
let uri = "employeeDetails?name=john&occupation=manager";
let encoded_uri = encodeURI(uri);
let decoded_uri = decodeURI(encoded_uri);
```

Function: Exception

General Structure of the exception

```
try {
  // codice che può causare errori
} catch (err) {
  // gestisci l'errore
} finally {
  // (opzionale) codice che viene eseguito sempre
}
```

Function: Type of Exception

- SyntaxError
- ReferenceError
- TypeError
- RangeError
- ...
- Custom Error

Function: Custom Error

Create a custom error:

```
class MyCustomError extends Error {
  constructor(message) {
    super(message);
    this.name = "MyCustomError";
  }
}
throw new MyCustomError("My Custom Error!");
```

Function: Exception in Asynchronous Programming

- use reject inside Promise
- you can pass an Error object inside the reject

```
const p = new Promise ( (resolve, reject) => {
reject(new Error('promise failed!'));
});
p.catch(
    err => {
        console.log(err);
    }
);
```

- use throw if you wanna create custom error
- you can use throw in a Promise
- you can use throw without Promise

Function: Exception in Reject or Throw (2)

- throw insisde a callback function will not be recognized by a catch block, and in this case use reject
- if throw is encountered, the flow is immediately interrupted, meanwhile the reject end the block and then goes on error
- you cannot create custom error with reject
- if you create a reject error, should always be a catch block of the element

Functional Programming: Function on Arrays

- forEach, Map, Filter, Reduce

Functional Programming: Function are First-Class Citizen

- In JavaScript, first-class citizens mean that functions are treated like any other variable:
- They can be assigned to a variable
- They can be passed as argument to another function
- They can be returned by another function
- This capability enables powerful patterns like callbacks, higher order functions, event handling and functional programming in JavaScript.

Functional Programming: Higher-Order Function

- A higher-order function is a function that operates on function, taking one or more function as arguments and returning a new function

An example:

```
function not(f) {
    return function() {
        var result = f.apply(this, arguments);
        return !result;
    }
}
```

Functional Programming: Unary Function

- A unary function (also known as a monadic function) is a function that accepts exactly one argument. The term "unary" simply refers to the function's arity - the number of arguments it takes

Functional Programming: Pure Function

- A pure function is a function whose output depends only on its input arguments and produces no side effects. This means that given the same inputs, a pure function will always return the same output, and it does not modify any external state or data, e.g. it has not side effects.

Functional Programming: Partial Application of Functions

- The bind() method of a function f returns a new function that invokes f in a specified context and with a specified set of arguments. We say that it binds the function to an object and partially applies the arguments.

Functional Programming: Currying

- Currying is the technique in which a function that takes multiple arguments is broken into several function that take only one argument
- This technique is typically used in functional programming

Functional Programming: Memoization

- Memoization is the technique in which you memoize results for application of a slow function in the way you can return it in constant time when the same function is re-applied to the same input

Client Side Javascript

- Handling Events
- Scripting DOM
- Web Workers
- Asynchronous Programming
- Client-Side Storage
- External Libraries
- Testing
- Server-Sent Events

Handling Events

- Event Bubbling
- Event Capturing

Handling Events: Common Events

- Click of the mouse
- Web page has loaded
- Mouse moves over an element
- Input field is changed
- HTML form is submitted
- User strokes a key

Handling Events: Assign Events

- Event attributes

```
<button onclick="displayDate()">Try it</button>
```

- Event Listener

```
document.getElementById("myBtn").addEventListener("click", displayDate);
```

Handling Events: Remove Event Listener()

You can also remove the event listener attached to a particular element

```
element.removeEventListener("mousemove", myFunction);
```

HTML DOM Manipulation

- DOM stands for Document Object Model and is constructed as a tree of Objects
- HTML DOM can be accessed with JavaScript
- A property is a value that you can get or set (like changing the content of an HTML element)
- A method is an action you can do (like add or deleting an HTML element)

HTML DOM Manipulation: API

- Finding HTML Elements
  - document.getElementById(id)
  - document.getElementsByTagName(name)
  - document.getElementsByClassName(className)
  - document.getElementsByClassName(name)
- Changing HTML Elements
  - element.innerHTML = new html content
  - element.attribute = new value
  - element.style.property = new style (adding css style property)
  - element.setAttribute(attribute, value)

HTML DOM Manipulation: API (2)

Adding and Deleting Elements

- document.createElement(element): create an HTML element
- document.removeChild(element): remove an HTML element
- document.appendChild(element): add an HTML element
- document.replaceChild(new, old): replace an HTML element
- document.write(text): write into the HTML output stream

Adding Events Handlers

- document.getElementById(id).onclick = function(){...}

HTML DOM Manipulation: JavaScript Forms

- HTML form validation can be done by JavaScript
  As example:

```
<form name="myForm" action="/action_page.php" onsubmit="return validateForm()" method="post">
Name: <input type="text" name="fname">
<input type="submit" value="Submit">
</form>
```

- Data validation is the process of ensuring that user input is clean, correct and useful
- Data validation can be done server side (using HTML, CSS or JavaScript) or client side (after calling the backend services)

HTML DOM Manipulation: Changing CSS

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

```
document.getElementById(id).style.property = new style
```

- The style of an HTML element can also be changed when an event occurs, for example on click of a button or on when the page is loaded.
- Using events and changing CSS style of the element, you can also create animation

HTML DOM Manipulation: DOM Events

- HTML DOM allows JavaScript to react to HTML events
- A JavaScript function can be executed when an event occurs, like when a user click on an HTML element

HTML DOM Manipulation: DOM Events (2)

- To assign events to HTML elements you can use event attributes

```
<button onclick="displayDate()">Try it</button>
```

- Assign an onclick event to a button element

```
document.getElementById("myBtn").onclick = displayDate;
```

where displayDate is a function, assigned to an HTML element

- The third way is to assign an event listener

```
document.document.getElementById("myBtn").addEventListener("click", function() { ... })
```

HTML DOM Manipulation: DOM Events (3)

A small list of event

- onload and onunload events
- oninput events
- onchange event
- onmouseover and onmouseout events
- onmousedown, onmouseup and onclick events
- ...

HTML DOM Manipulation: EventListener

- You can attach an event listener to an element:

```
document.getElementById("myBtn").addEventListener("click", displayDate);
```

- You can also attach many event Handlers to the same element

```
element.addEventListener("click", myFunction);
element.addEventListener("click", mySecondFunction);
```

HTML DOM Manipulation: Event Listener to the window Object

- The addEventListener() method allows you to add event listeners on any HTML DOM object such as HTML elements, the HTML document, the window object, or other objects that support events, like the xmlHttpRequest object

```
window.addEventListener("resize", function(){
  document.getElementById("demo").innerHTML = sometext;
});
```

HTML DOM Manipulation: Event Bubbling or Event Capturing

- Event propagation is a way of defining the element order when an event occurs.
- In bubbling the inner most element's event is handled first and then the outer
- In capturing the outer most element's event is handled first and then the inner

```
addEventListener(event, function, useCapture);
```

- The default value is false, which will use the bubbling propagation, when the value is set to true, the event uses the capturing propagation.

HTML DOM Manipulation: removeEventListener method

- The removeEventListener() method removes event handlers that have been attached with the addEventListener() method

```
element.removeEventListener("mousemove", myFunction);
```

HTML DOM Navigation

- With the HTML DOM, you can navigate the node tree using node relationship
- With the HTML DOM, all noes in the node tree can be acessed by Javascript
- New nodes can be created, and all nodes can be modified or deleted

HTML DOM Navigation: Node Relationships

- html is the root node, it has no parents, the first child is head and second and last child is body
- then head has one child, title, and title has a text node
- body has two children ... and so on and so forth
- InnerHTML property let you retrieve the content of an HTML element

HTML DOM Navigation: DOM Root Nodes

- document.body - The body of the document
- document.documentElement - The full document

HTML DOM Navigation: nodeName Property

- the nodeName property specify the name of a node

```
<h1 id="id01">My First Page</h1>
<p id="id02"></p>

<script>
document.getElementById("id02").innerHTML = document.getElementById("id01").nodeName;
</script>
```

HTML DOM Navigation: nodeValue Property

- The nodeValue property specifies the value of a node
- The nodeType Property is read only. It returns the type of a node.

HTML DOM Navigation: Creating New HTML Elements

An example of adding a new HTML elements

```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script>
const para = document.createElement("p");
const node = document.createTextNode("This is new.");
para.appendChild(node);

const element = document.getElementById("div1");
element.appendChild(para);
</script>
```

HTML DOM Navigation: Removing Existing HTML Elements

An example of removing existing node:

```
<div>
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script>
const elmnt = document.getElementById("p1"); elmnt.remove();
</script>
```

- You can also do remove child (parent.removeChild(child)), instead of removing the current node
- Or also replace child with parent.replaceChild(para, child);

HTML DOM Navigation: HTMLCollection Object

- The getElementsByTagName() method return an HTMLCollection object.
- An HTMLCollection object is an array-like list (collection) of HTML elements

```
const myCollection = document.getElementsByTagName("p");
```

- Then you can apply any kind of operation you know on array

HTML DOM Navigation: HTML NodeList Object

- A NodeList object is a list (collection) of nodes extracted from a document
- A NodeList object is almost the same as an HTMLCollection object
- Some (older) browsers return a NodeList object instead of an HTMLCollection for methdos like getElementsByClassName or querySelectorAll()
- The difference from HTMLCollection: NodeList is a collection of document nodes
- HTMLCollection is always a mutable collection, instead NodeList is most often a static collection.

HTML BOM: Browser Object Model

- The Browser Object Model (BOM) allows JavaScript to "talk to" the browser
- The window object is supported by all browser. It represnts the browser's window.

HTML BOM: Window Methods

Two properties can be used to determine the size of the browser window.

- window.innerHeight - the inner height of the browser window (in pixels)
- window.innerWidth - the inner width of the browser window (in pixels)
- Some other methods:
    - window.open() - open a new window
    - window.close() - close the current window
    - window.moveTo() - move the current window
    - window.resizeTo() - resize the current window

HTML BOM: Window Screen

- The window.screen object contains information about the user's screen
- Properties are:
  - screen.width
  - screen.height
  - screen.availWidth
  - screen.availHeight
  - screen.colorDepth
  - screen.pixelDepth

HTML BOM: Window Location

- The window.location object can be used to get the current page address (URL) and to redirect the browser to a new page.
- properties available:
  - window.location.href returns the href (URL) of the current page
  - window.location.hostname return the domain name of the web host
  - window.location.pathname returns the path and filename of the current page
  - window.location.protocol returns the web protocol used
  - window.location.assign() loads a new document

HTML BOM: Window History

- The window.history object contains the browser history
- Some methods:
  - history.back() - same as clicking back in the browser
  - history.forward() - same as clicking forward in the browser

HTML BOM: JavaScript Popup Boxes

- JavaScript has three kind of popup boxes: Alert box, Confirm box, and Prompt box.

An example of Alert Box

```
window.alert("sometext");
```

An example of Confirm Box

```
window.confirm("sometext");
```

An example of Prompt Box

```
window.prompt("sometext","defaultText");
```

Web Geolocation API

- The HTML Geolocation API is used to get the geographical position of a user
- The getCurrentPosition() method is used to return the user's position.
- You can display the results in a map, but you need access to a map service, like Google Maps or OpenStreetMap

```
console.log(navigator.geolocation.getCurrentPosition(showPosition);)
```

Web Geolocation API: Showing the result in a map

An example of snippet:

```
function showPosition(position) {
  let latlon = position.coords.latitude + "," + position.coords.longitude;

  let img_url = "https://maps.googleapis.com/maps/api/staticmap?center=
  "+latlon+"&zoom=14&size=400x300&sensor=false&key=YOUR_KEY";

  document.getElementById("mapholder").innerHTML = "<img src='"+img_url+"'>";
}
```

This snippet is given from W3School tutorial.
(https://www.w3schools.com/js/js_api_geolocation.asp)

- You can also get other information from getCurrentPosition() like latitude, longitude, accuracy, altitude...

Asynchronous Programming: Promise Introduction

- "Producing code" is code that can take some time
- "Consuming code" is code that must wait for the result
- A Promise is an Object that links Producing code and Consuming code, in particular: A Promise contains both the producing code and calls to the consuming code

Asynchronous Programming: Promise usage

For example:

```
let myPromise = new Promise(function(myResolve, myReject){
    // Producing code
    myResolve();
    myReject();
});

myPromise.then(
    function(value) {...},
    function(error) {...}
);
```

Asynchronous Programming: Promise Props

- A Promise object can be "pending" (working), the result is undefined
- A Promise object can be "fulfilled", the result is a value
- A Promise object can be "rejected", the result is an error object

Asynchronous Programming: Promise Args

- Promise take a callback for success and another for failure
- Promise take a error, for failure

Asynchrnous Programming: Async & Await

- Async and Await are syntactic sugar, making promises easier to write

For example:

```
async function myFunction() {
    return "Hello";
}
```

Is the same as:

```
function myFunction() {
    return Promise.resolve("Hello");
}
```

Asynchronous Programming: Async Usage

```
async function myFunction() {
    return "Hello";
}
myFunction().then (
    function(value) {myDisplayer(value);}
    function(error) {myDisplayer(error);}
)
```

Asynchronous Programming: Await

- The await keyword makes the function ppause the execution and wait for a resolved promise before it continues
- Await can be used only inside an async function

```
async function myDisplay() {
    let myPromise = new Promise(function(resolve) {
        setTimeout(function() {resolve("Hello world");}, 3000)
    });
    document.getElementById("demo").innerHTML = await myPromise;
}

myDisplay();
```

Client-Side Storage

- Web Storage: sessionStorage and localStorage
- Cookies
- IE User Data
- Offline Web Application
- Web Databases
- Filesystem API

Web Storage: localStorage and sessionStorage

- Storage objects work much like regular JavaScript objects: simply set a property of the object to a string, and the browser will store that string for you.
- The difference between localStorage and sessioneStorage has to do with lifetime and scope: how long the data is saved for and who the is accessible to

Web Storage: localStorage and sessionStorage (2)

As usage, are the same:

```
var name = localStorage.username;
name = localStorage["username"];

for (var name in localStorage) {
    var value = localStorage[name]
}

sessionStorage.setItem("name","Jack");
let sessionName = sessionStorage.getItem("name")
```

Web Storage: localStorage and sessioneStorage (persistence)

- In local storage data will NOT expire after closing the browser (or the window)
- In session storage data will be lost after closing the browser (or the window)
- the session storage is valid only for session of the window, meanwhile the localStorage have no kind of restriction

Web storage: localStorage and sessionStorage (scope)

- localStorage is share among the all opened windows
- every window has its own sessionStorage
- both of localStorage and sessionStorage are accessible only from the same domain

Web storage: cookie

- A cookie is a piece of data that is stored on your computer to be accessed by your browser. Cookies are saved as key/value pairs
- Cookie are useful to remember information about the user profile (such as username). It basically involves two steps, when a user visits a web page, the user profile can be stored in a cookie and next time the user visits the page, the cookie remembers the user profile.
- To set cookie in a webpage, you'll have:

```
document.cookie = "username=JohnDoe; path=/services";
```

- To delete a cookie, you can set the expiration date as passed:

```
document.cookie = "username=; expires=Fri, 07 Jun 2019 00:00:00 UTC; path=/;";
```

Testing

1. Basics
2. Jasmine (Jest, Mocha)

External Libraries

External Libraries: Topics

1. Ajax
2. jQuery
3. Fetch
4. Axios
5. Superagent
6. Prototype
7. Node HTTP

Server Side JavaScript