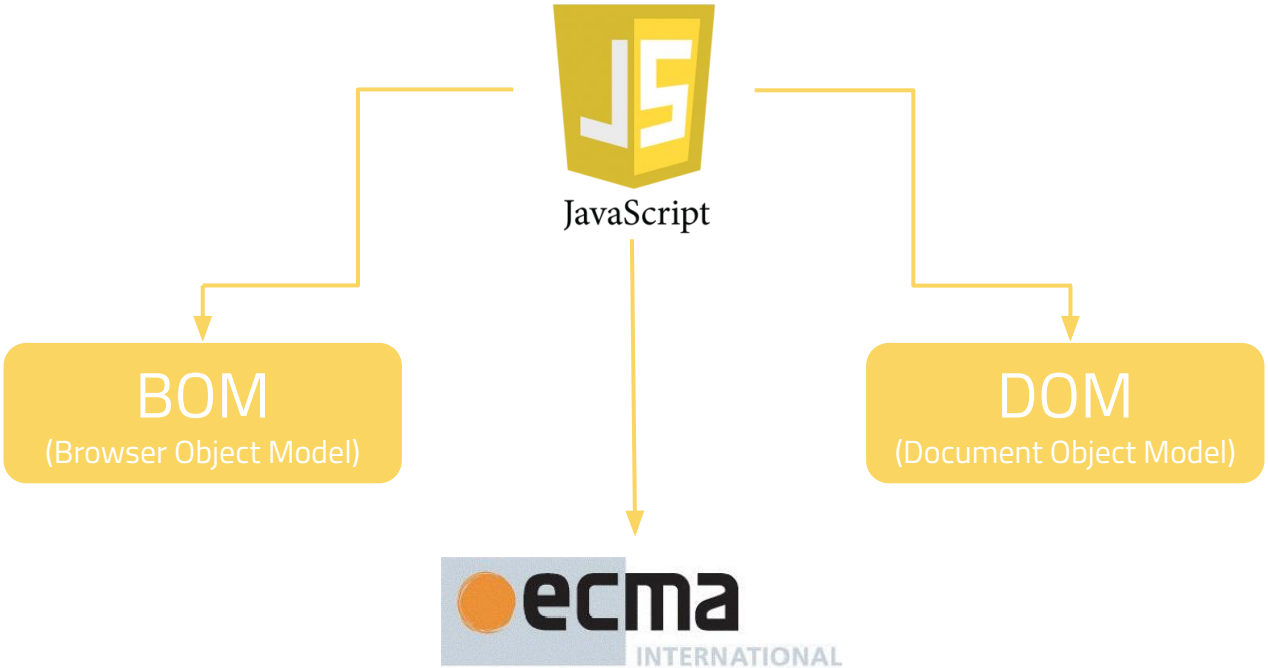

JMenuBar mnuMain = new JMenuBar();
JMenuItem
mnuNewGame = new JM
mnuInstruction = new JM
mnuExit = new JMenuIter
JButton btn1v1 =
JButton btnEmpty
JPanel pnlNew
JLabel lblTitle = new
JTextArea txtMessage = new
JOptionPane.
"your current game will be lost..." + "\n" +
"Are you sure you want to continue?",
"Quit Game?" ,JOptionPane.YES_NO_OPTION);
if(op

JavaScript

Vahram Aleksandryan
Senior JavaScript Developer(VOLO)
7+ years of experience

vahramaleksandryan@gmail.com
(+374) 93749743

What is JavaScript?



A brief history of JavaScript

1995 Brendan Eich creates the very first version of JavaScript in just 10 days. It was called Mocha, but already had many fundamental features of modern JavaScript.

1996 Mocha changes to LiveScript and then JavaScript, in order to attract Java developers. However, JavaScript has almost nothing to do with Java.

Microsoft launches IE, copyright JavaScript from Netscape and calling it JScript.

1997 With a need to standardize the language, ECMA releases ECMAScript 1, the first official standart of JavaScript.

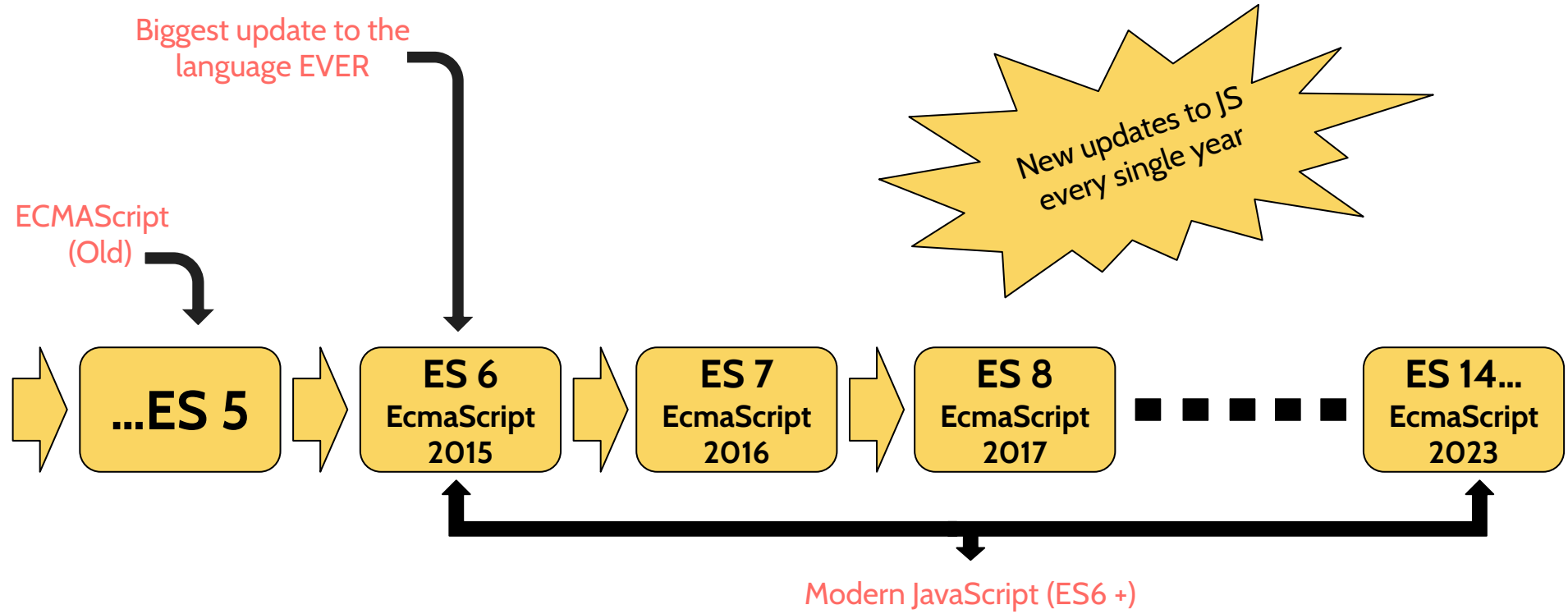
2009 ES5 (EcmaScript 5) is released with lots of great new features.

2015 ES6/ES2015 (ECMAScript 2015) was released: the biggest update to the language ever!

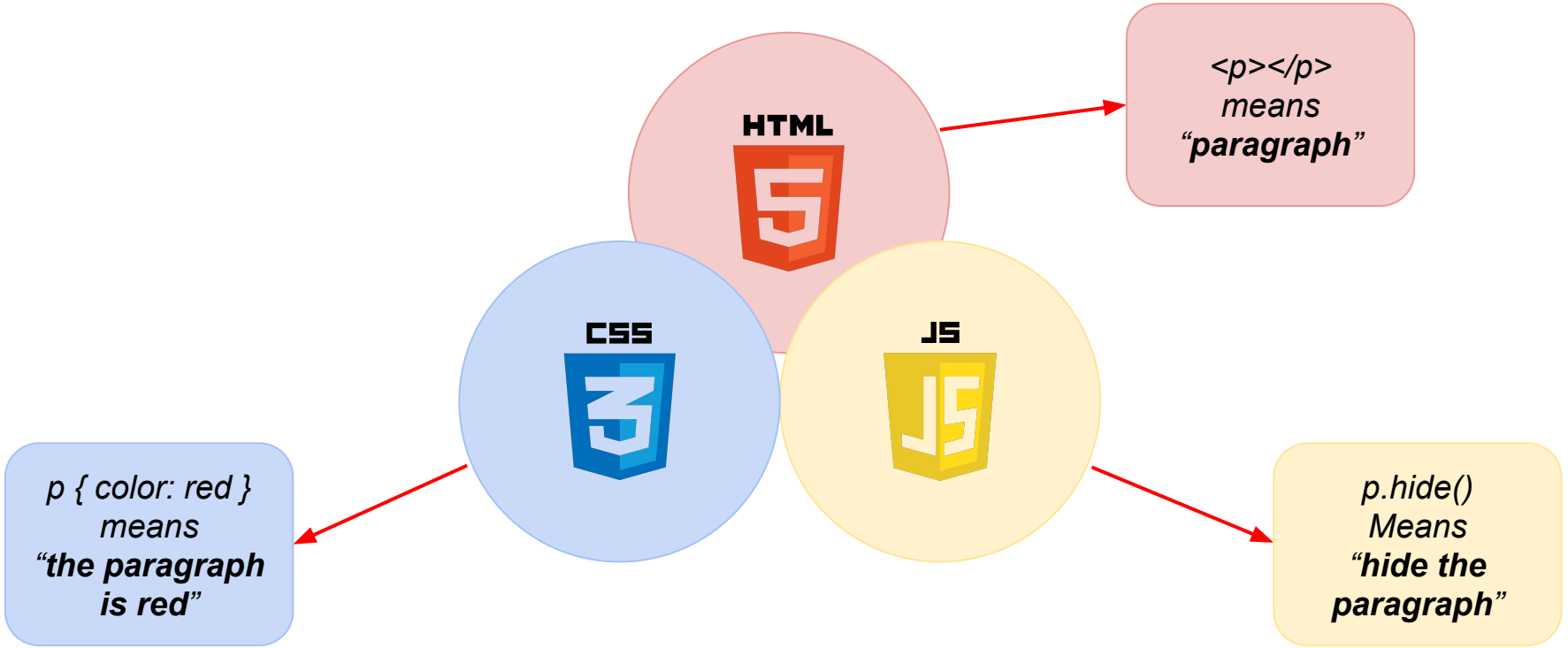
ECMAScript changes to an annual release cycle in order to ship less features per update.

2016 - ∞ Release of ES2016 / ES2017 / ES2018 / ES2019 / ES2020 / ES2021 /ES2022 ...

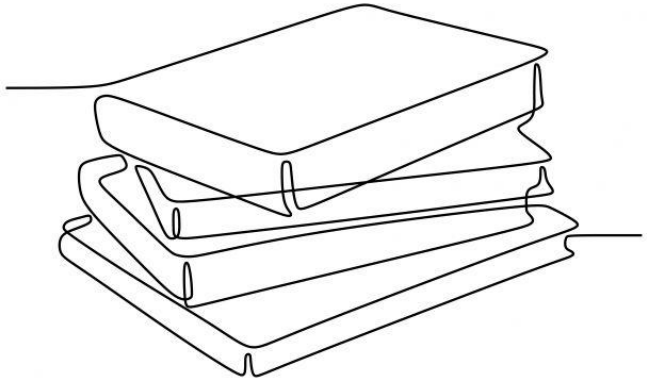
EcmaScript Releases



The role of JavaScript in web development



ECMAScript(ES 5)



01

Section

- Data types
- Operators
- Expressions and Statements
- Variables

Data Types

The set of types in the JavaScript language consists of primitive values and objects.

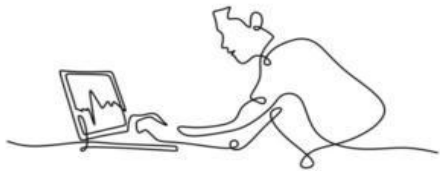
- Primitive types
  - Numeric
  - String
  - Boolean
  - Undefined
  - Null
  - Symbol (ECMAScript 6+)
- Object (reference types)

Numeric Type

ECMAScript has two built-in numeric types: **Number** and **BigInt**.

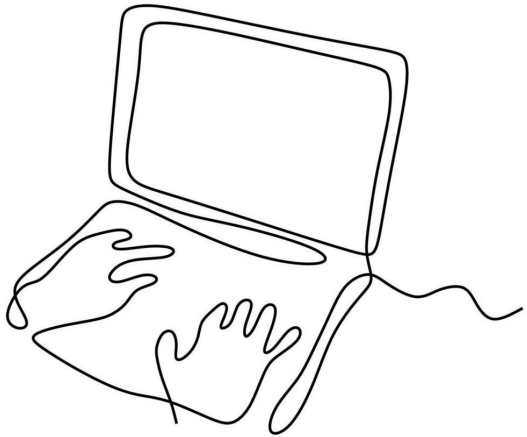
- Number
  - The JavaScript Number type is a **double-precision 64-bit binary format IEEE 754** value, like double in Java or C#. This means it can represent fractional values, but there are some **limits** to the stored number's magnitude and precision. Very briefly, an IEEE 754 double-precision number uses 64 bits to represent 3 parts:
    - 1 bit for the sign (positive or negative)
    - 11 bits for the exponent (-1022 to 1023)
    - 52 bits for the mantissa (representing a number between 0 and 1)

- BigInt (ECMAScript 6+)
  - 9007199254740991n



Mathematical Symbols And Operations

- Addition ' + '
  - 10 + 5 //15
- Subtraction ' - '
  - 10 - 5 //5
- Multiplication ' * '
  - 10 * 5 //50
- Exponentiation ' ** ' (ECMAScript 6+)
  - 10 ** 2 //100
- Division ' / '
  - 10 / 5 //2
- Modulus (Remainder) ' % '
  - 10 % 2 //0
  - 5 % 2 //1
- Expression grouping ' ( ) '
  - (7 + 4) * 2 //22



String Type

JavaScript's String type is used to represent textual data. It is a set of "elements" of 16-bit unsigned integer values. You can use single or double quotes.

- *Strings*
  - “Hello, world!”
  - ‘Hello, world!’
- *Special characters*
  - 'Hello, \n\tworld!'
  - 'Hello \'my\' world!'
  - 'format disk c:\\ on computer'
- *Concatenation*
  - 'Hello ' + ',' + ' world' + '!'
  - '5' + '9'

Boolean Type

Boolean represents a logical entity and can have two values: **true** and **false**.

- *Comparison Operators*
  - == and ===
  - < and >
  - <= and >=
- *Logical Operators*
  - ! (NOT)
  - && (AND)
  - || (OR)

Examples

- 5 && 6 //6
- 5 || 6 //5
- 6 && 0 //0
- 0 || 6 //6
- “Hello” && “test” //test
- “ ” && “hello” //hello
- “” && “hello” //””
- !true //false

Practical Work

- 2 + '2';
- 2 + '2' + 2;
- '5' + 6 + 7;
- 4 + true;
- ' ' + true;
- 'Hello' && 2 || ' ' && 5;
- ((5 >= 7) || ('javascript' != 'java')) && !(((11 * 3)== 99) && true);
- 'a' * 10; 🤔

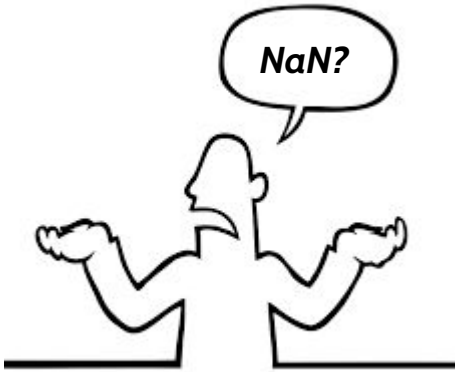
What is NaN ?

JavaScript uses **NaN** as the result of a failed operation on numbers.

- *Examples*
  - 10 * 'text' //NaN
  - 10 / 'text' //NaN
  - 0 / 0 //NaN
  - 100 + 0 / 0 //NaN

❖ The *isNaN()* determines whether a value is *NaN* or not.

- ➢ isNaN(NaN) //true
- ➢ isNaN(10 * 'text') //true
- ➢ isNaN(10) //false



Number.isNaN() (ECMAScript 6+)

typeof Operator

The **typeof** operator returns a **string** indicating the type of the **operand's value**.

- typeof 37 //'number'
- typeof 3.14 //'number'
- typeof NaN //'number'
- typeof 'text' //'string'
- typeof ' ' //'string'
- typeof true //'boolean'

Data Type Conversion

- *Number*
  - a = '12' * 1;
  - a = +'12';
  - a = parseInt('12.57');
  - a = parseFloat('12.57');
  - a = Number(true);

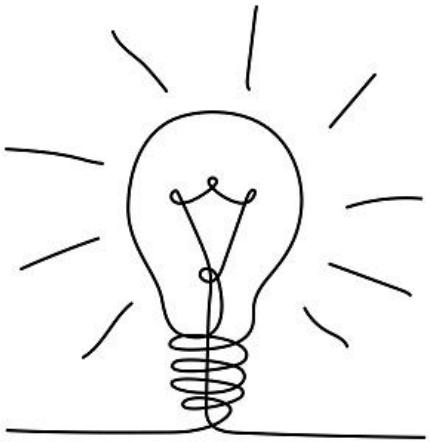
- *String*
  - a = 5 + ' ';
  - a = String(true);

- *Boolean*
  - a = !5;
  - a = Boolean(5);

What is a variable ?

A variable is a container for a value, like a number we might use in a sum, or a string that we might use as part of a sentence.

- *Declaring a variable (a, A, $, _)*
  - var name;
  - var age, email;
- *Initializing a variable*
  - var name = 'John';
  - var name = 'John', age, password = 123;
- *Updating a variable*
  - var number = 10;
    number = number + 5;
    number += 5;



null and undefined Data Types

In computer science, a **null** value represents a reference that points, generally intentionally, to a **nonexistent or invalid object or address**.

- typeof null; // 'object' (BUG)
- parseInt(null); // NaN
- parseFloat(null); // NaN
- Number(null); //0
-

undefined is a primitive value automatically assigned to variables that have just been declared.

- typeof undefined; //undefined

Difference between **null** and **undefined**.

- null == undefined //true
- null === undefined //false

Expressions and Statements

- An **expression** is a code that, after execution, returns a value.
  - 2 + 3 = 5;
  - 2 + 2 * 4 = 10;
  - (4 + 2) * (1 + 3) = 24;

- A **statement** is a single command in code that performs a specific action. Statements do not evaluate or return anything, so they are not expressions. In JavaScript, all statements can be divided into several categories:
  - flow control (if and else, switch....);
  - iterations (for, while....);
  - declaration of values (var....);
  - functions (function, return....);
  - others (debugger, import, export....);

ECMAScript Comments

ECMAScript **comments** can be used to explain code, and to make it more readable. Comments can also be used to **prevent execution**, when testing alternative code. There are two different ways to write comments in ECMAScript: on a **single-line** or on **multiple** lines.

- A **single–line** comment begins with two forward-slash characters, such as this:
    - // single line comment
- A **block** comment begins with a forward slash and asterisk **(/*)** and ends with the opposite **(*/)**, as in this example:
    - /* This is a multi-line
      comment */

JS

02

Section

- Loops and iteration (while, for, do while)
- Statements (if, else, else if)
- Break statement
- Continue statement
- Switch Case statement

Loops and Iteration (while statement)

A **while** statement executes its statements as long as a specified condition evaluates to **true**. A while statement looks as follows:

- ```
  while ( statement ) {
      //condition
  }
  ```

If the condition becomes **false**, statement within the loop stops executing and control passes to the statement following the loop.

The following while loop iterates as long as n is less than 5:

- ```
  var n = 0;
  while ( n < 5 ) {
      console.log( n );
      n += 1;
  }
  ```

Increment (++) and Decrement (--) operators

- *Post Increment/Decrement*
  - ```
    var n = 1;
    var cnt = n++;
    console.log( n );     //n 2;
    console.log( cnt ); //cnt 1;
    ```

- *Pre Increment/Decrement*
  - ```
    var n = 1;
    var cnt = ++n;
    console.log( n );     //n 2;
    console.log( cnt ); //cnt 2;
    ```

Loops and Iteration (for statement)

A **for** loop repeats until a specified condition evaluates to **false**. A for statement looks as follows:

- ```
  for ([initialExpression]; [conditionExpression]; [incrementExpression]) {
      //statement
  }
  ```

The following for loop iterates as long as ***i*** is less than **10**:

- ```
  for (var i = 1; i < 10; i++) {
      console.log( i );
  }
  ```

Loops and Iteration (do...while statement)

The **do...while** statement repeats until a specified condition evaluates to **false**.

A **do...while** statement looks as follows:

- ```
  do {
      //statement
  } while (condition);
  ```
  Statement is always executed **once before the condition is checked**.

- ```
  var n = 0;
  do {
      console.log( n ); //0; 1; 2; 3; 4;
      n++;
  } while (n < 5);
  ```

- ```
  var n = 0;
  do {
      console.log( n ); //0;
      n++;
  } while (false);
  ```
  ```

Statements (if, else, else..if)

The **if** statement executes a statement if a specified condition is **truthy**. If the condition is **falsy**, another statement in the optional **else** clause will be executed.

- ```
  if ( condition ) {
        //statement
  }
  ```

- ```
  if ( condition ) {
        //statement 1
  } else {
        //statement 2
  }
  ```

- ```
  if ( condition 1 ) {
        //statement 1
  } else if ( condition 2 ) {
        //statement 2
  //....
  } else {
        //statement N
  }
  ```
  ```

Conditional (ternary) Operator

The **conditional (ternary)** operator is the only JavaScript operator that takes three operands: a condition followed by a question mark **(?)**, then an expression to execute if the condition is **truthy** followed by a colon **(:)**, and finally the expression to execute if the condition is **falsy**.

- condition ? exprIfTrue : exprIfFalse;
    - ```
      var n = 5;
      n == 5 ? console.log(true) : console.log(false);
      ```
    - ```
      var a = 10;
      var b = (a == 5) ? true : false;
      console.log(b);
      ```

Break and Continue Statements

- The **break** statement terminates **the current loop**.
    - ```
      for ( var i = 0; i < 5; i++ ) {
          if ( i == 2 ) {
              break;
          }
          console.log( i );
      }
      ```
- The **continue** statement terminates execution of the statements in **the current iteration of the current or labeled loop**, and continues execution of the loop with the **next iteration**.
    - ```
      for ( var i = 0; i < 5; i++ ) {
          if ( i == 2 ) {
              continue;
          }
          console.log( i );
      }
      ```

Switch...Case Statement

The **switch** statement evaluates an expression, matching the expression's value against a series of **case** clauses, and executes statements after the first case clause with a matching value, until a **break** statement is encountered. The **default** clause of a switch statement will be jumped to if no case matches the expression's value.

- ```
  switch (expression) {
      case value1:
          //Statements executed when the result of expression matches value1
          break;
      case valueN:
          //Statements executed when the result of expression matches valueN
          break;
      default:
          //Statements executed when none of the values match the value of the expression
          break;
  }
  ```
  ```

JS

03

Section

- What is function?
- Defining functions
- Function arguments
- Return statement
- Function expressions
- Scope of variables, Hoisting
- Anonymous function
- Closure, Recursion

Functions

Functions are one of the **fundamental building blocks** in ECMAScript. A function in ECMAScript is similar to a procedure—a set of statements that performs a task or calculates a value, but for a procedure to qualify as a function, it should **take some input and return an output** where there is some obvious relationship between the input and the output. To use a function, you must **define** it somewhere in the scope from which you wish to **call** it.

- **Defining functions (Function declarations)**

➢
```
function sayHello() {
        console.log("Hello, World!");
}
sayHello();
```

➢
```
function sayHello(name) {
        console.log("Hello, " + name + "!");
}
sayHello("John");
```

Functions (return statement)

The **return** statement is used to return a **particular value** from the function to the function caller. The function will **stop** executing when the return statement is called.

- ```
  function sum(numOne, numTwo) {
      return numOne + numTwo;
  }

  var result = sum(5, 7);
  console.log(result);
  console.log(sum(5, 7));
  ```


  ```

Functions

- **typeof functions**
  - typeof function some () { }; //'function'
- **Function expression**
  - The function keyword can be used to define a function inside an expression.
    - ➢
```
function sayHello() {
    console.log("Hello, world!");
}
var test = sayHello;
test();
```
    - ➢
```
var sayHello = function() {
    console.log("Hello, world!");
};
sayHello();
```

Functions

- Returning a function

  *Functions can **return** a function.*

  - ```
    function outer() {
        function inner(name){
            console.log("Hello, " + name);
        }
        return inner;
    }
    var test = outer();
    test("John");
    ```


    ```

Functions

- Returning a function

  *Functions can **return** a function.*

  - ```
    function outer() {
        return function (name){
            console.log("Hello, " + name);
        };
    }
    var test = outer();
    test("John");
    ```


    ```

Scope of variables

Scope determines the **accessibility** (visibility) of variables. ECMAScript has 3 types of scope:

- *Global scope*
    - A variable declared outside a function, becomes **GLOBAL**.
    - A global variable has Global Scope: All scripts and functions on a web page can access it.
- *Local Scope*
    - Variables declared within a function, become **LOCAL** to the function.
    - Local variables have Function Scope. They can only be accessed from within the function.
- *Block Scope* (ECMAScript 6 +)
    - Variables declared inside a { } block cannot be accessed from outside the block. *(let, const)*

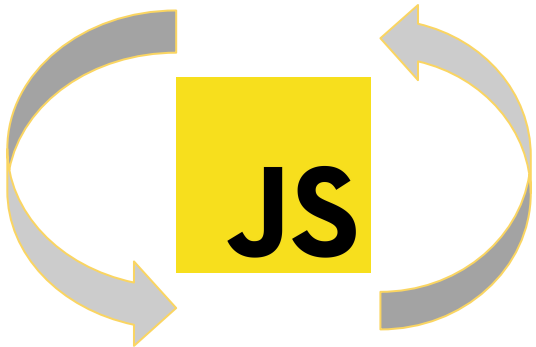
Scope of variables

-
```
if (true) {
    var name = "John";
}

console.log(name); //John
```

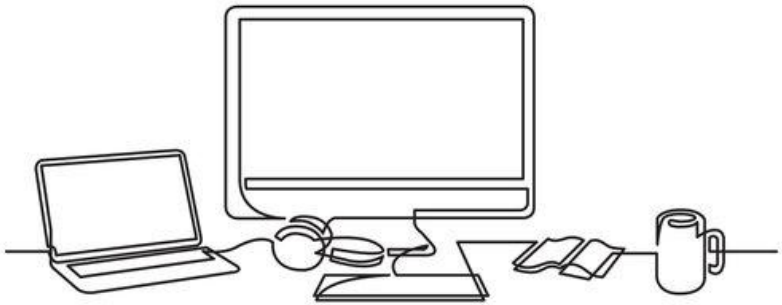
-
```
var name;

if (true) {
    name = "John";
}

console.log(name); //John
```

Scope of variables

- ```
  var num = 23;  //Global variable

  function some() {
        console.log(num); //23
  }

  some();
  console.log(num); //23
  ```


  ```

Scope of variables


- ```
  function some(){
      var firstName = "John";  // local variable
  }
  console.log(firstName);  //Uncaught ReferenceError: firstName is not defined
  ```

- ```
  function some(firstName){
      return firstName;  // local variable
  }
  console.log(firstName); //Uncaught ReferenceError: firstName is not defined
  ```

Hoisting

Hoisting refers to the process whereby the interpreter appears to **move** the declaration of **functions**, **variables** or **classes** to the **top** of their scope, prior to execution of the code.

- *Variable hoisting*
  - ```
    console.log(str);
    var str = 'some text';
    console.log(str);
    ```

- *Function hoisting*
  - ```
    greet();
    function greet() {
        console.log('Hello!');
    }
    ```

Functions (Closure)

A **closure** is the combination of a function bundled together **(enclosed)** with references to its surrounding state **(the lexical environment)**. In other words, a closure gives you access to an outer function's scope from an inner function. Closures are created every time a function is created, at function creation time.

```
function init() {
        var name = 'Mozilla'; //name is a local variable created by init function
        function displayName() {  //displayName is the inner function
                console.log(name);
        }
        return displayName;
}
var func = init();
init();
```

Functions (Recursion)

The act of a function **calling itself**, recursion is used to solve problems that contain smaller sub-problems. A recursive function can receive **two** inputs: **a base case** (ends recursion) or **a recursive case** (resumes recursion).

```
function loop(count) {
    if (count > 10) {
        return;
    }
    console.log(count);
    loop(count + 1);
}
loop(0);
```



JS

04

Section

- What is object?
- Object data type - Object
- Object - properties
- Object - methods
- this, arguments
- Function - (properties, methods)
- Object data type -Array

Object - Properties

The **Object type** represents one of ECMAScript's data types. It is used to store various keyed collections and more complex entities. Objects can be created using the Object() constructor or the **object initializer / literal syntax**.

- ```
  var user = { };
  user.name = "John";
  user.age = 25;

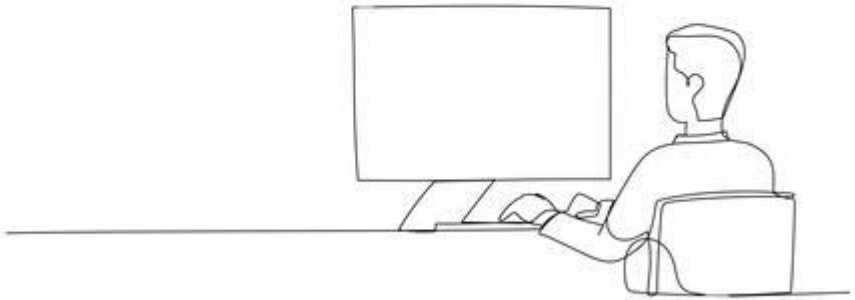
  console.log(user);
  console.log(user.name);
  console.log(user.age);

  typeof user; //object
  ```


  ```

Object - Properties

- ```
  var user = {
        name: "John",
        age: 20
  };

  console.log(user);
  console.log(user.name);
  console.log(user.age);
  ```

```
user.name = "Mike";
user.role = "Admin";

console.log(user);
```
```

Object - Properties

- ```
  var user = {
      "user name": "John",
      var: 0,
      3: true
  };

  console.log(user);
  console.log(user["user name"]);
  console.log(user["var"]);
  console.log(user[1+2]);
  ```


  ```

delete operator

The **delete** operator removes a property from an object.

```
var user = {
        name: "John",
        age: 20
};

delete user.name;  //true
console.log(user);
```

in operator

The in operator returns **true** if the specified property is in the specified object.

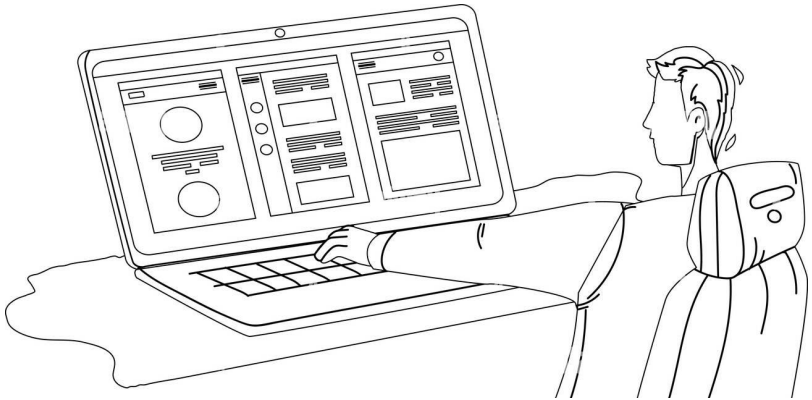
- ```
  var user = {
        name: "John",
        age: 20
  };

  console.log("name" in user); //true
  console.log("age" in user);    //true
  console.log("role" in user);   //false
  ```


  ```

for...in statement

The **for...in** statement **iterates** over all enumerable string properties of an object.

- ```
  var user = {
      0: "John",
      1: 20,
      2: "admin"
  };

  for (var i = 0; i in user; i++) {
      console.log(i + ": " + user[i]);
  };
  ```

- ```
  var user = {
      name: "John",
      age: 20,
      role: "admin"
  };

  for (var i in user) {
      console.log(i + ": " + user[i]);
  };
  ```

Object - Methods

- ```
  var user = {
        name: "John",
        age: 20,
        sayHello: function(msg){
        console.log("Hello " + msg);
        }
  };

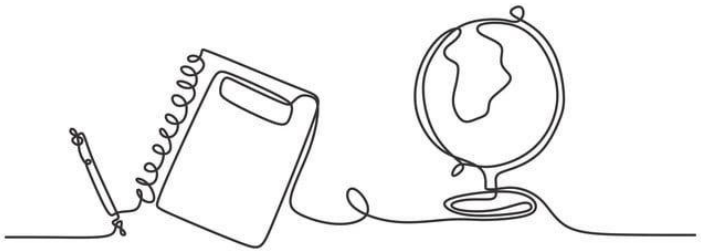
  console.log(user);

  user.sayHello("John");
  user.sayHello("Mike");
  ```


  ```

this keyword

In most cases, the value of **this** is determined by how a function is called **(runtime binding)**. It can't be set by assignment during execution, and it may be **different** each time the function is **called**.

- ```
  var user = {
        name: "John",
        age: 20,
        sayHello: function( ) {
              console.log("Hello " + this.name);
        }
  };

  user.sayHello();
  ```


  ```

this keyword

- ```
  var user = {
      name: "John",
      age: 20,
      sayHello: function( ) {
          function say( ) {
              console.log(this.age);
          }
          say();
      }
  };

  user.sayHello(); //undefined
  ```


  ```

this keyword

- ```
  var user = {
      name: "John",
      age: 20,
      sayHello: function( ) {
          var self = this;
          function say( ) {
              console.log(self.age);
          }
          say();
      }
  };

  user.sayHello(); //20
  ```


  ```

Object data type - Array

The **Array object**, as with arrays in other programming languages, enables **storing a collection** of multiple items under a single variable name, and has **members** for performing common array operations.

```
var myArr = [ ]; //empty array
var myArr = [ 10, "Mike", true, {a: "a"}, [true, 55 ]];

console.log(myArr);
console.log(myArr[0]);
console.log(myArr[3]);

myArr[15] = "John";
console.log(myArr);
```

Array - length

The **length** property **sets** or **returns** the **number** of elements **in that** array.

```
var myArr = [ 10, "Mike", true, {a: "a"}, [true, 55 ]];
console.log(myArr.length); //5

myArr.length = 60;
console.log(myArr.length); //60

myArr.length = 3;
console.log(myArr); //[ 10, "Mike", true]
```

Array - length(loop)

```
var myArr = [ 10, "Mike", true ];
myArr[5] = "John";


for(var i = 0; i <= myArr.length; i++) {
        console.log(i +": " + myArr[i]);
}

for(var i in myArr) {
        console.log(i +": " + myArr[i]);
}
```

Array - Methods (toString, join)

```
var myArr = [ 10, "Mike", true ];
```

- *toString()*
  - The **toString()** method returns a **string** representing the specified array and its elements.

```
var str = myArr.toString();
console.log(str); //"10,Mike,true"
```

- *join()*
  - The **join()** method creates and returns a **new string** by concatenating all of the elements in an array, separated by commas or a specified separator string. If the array has only one item, then that item will be returned without using the separator.

```
var str = myArr.join();
console.log(str); //"10,Mike,true"
```

```
var str = myArr.join(":");
console.log(str); //"10:Mike:true"
```

Array - Methods (concat)

```
var arrayFirst = [ 10, "Mike", true ];
var arraySecond = [ 20, "John", false ];
var arrayThird = [ 30, "Kevin", true ];
```

- *concat()*
  - The **concat()** method is used to **merge** two or more arrays. This method does **not change** the existing arrays, but instead returns a **new array**.

```
arrayFirst.concat(arraySecond); //[10, 'Mike', true, 20, 'John', false]
arrayFirst.concat(arraySecond, arrayThird);
                                //[10, 'Mike', true, 20, 'John', false, 30, 'Kevin', true]
```

  -

```
var newString = arrayFirst + arraySecond;
console.log(newString); //'10,Mike,true20,John,false'
```

Array - Methods (slice)

```
var arr = [ 10, "Mike", true, 20, "John", false ];
```

- *slice()*
  - The **slice()** method **returns a copy** of a portion of an array into a **new** array object selected from start to end **(end not included)**, where start and end represent the **index** of items in that array. The original array **will not be modified**.
    - ```
      var copyArr = arr.slice(2);
      console.log('copyArr: ', copyArr); //[ true, 20, 'John', false ]
      ```
    - ```
      var copyArr = arr.slice(2, 4);
      console.log('copyArr: ', copyArr); //[ true, 20 ]
      ```
    - ```
      var copyArr = arr.slice(-2);
      console.log('copyArr: ', copyArr); //[ 'John', false ]
      ```
    - ```
      var copyArr = arr.slice(2, -1);
      console.log('copyArr: ', copyArr); //[ true, 20, 'John']
      ```
    - ```
      var copyArr = arr.slice(3, 2);
      console.log('copyArr: ', copyArr); //[ ]
      ```
    - ```
      var copyArr = arr.slice( );
      console.log('copyArr: ', copyArr); //[ 10, "Mike", true, 20, "John", false ]
      ```

Array - Methods (push, pop)

```
var arr = [ 10, "Mike", true, 20, "John", false ];
```

- *push()*
  - The push() method **adds** one or more elements **to the end** of an array and **returns** the new **length** of the array.

```
var length = arr.push("test", 30);
console.log(arr); //[ 10, 'Mike', true, 20, 'John', false, 'test', 30 ]
console.log(length); //8
```

- *pop()*
  - The pop() method **removes** the **last element** from an array and **returns that element**. This method **changes the length** of the array.

```
var popped = arr.pop();
console.log(arr); //[ 10, 'Mike', true, 20, 'John' ]
console.log(popped); //false
```

Array - Methods (shift, unshift)

```
var arr = [ 10, "Mike", true, 20, "John", false ];
```

- *shift()*
  - The shift() method **removes the first element** from an array and **returns that removed element**. This method **changes the length** of the array.

```
var removedElement = arr.shift();
console.log(arr); //[ 'Mike', true, 20, 'John', false ]
console.log(removedElement); //10
```

- *unshift()*
  - The unshift() method **adds one or more elements** to the **beginning** of an array and **returns the new length** of the array.
  -

```
var length = arr.unshift(30, "tests");
console.log(arr); //[ 30, 'tests', 10, 'Mike', true, 20, 'John', false ]
console.log(length); //8
```

Array - Methods (splice)

```
var arr = [ 10, "Mike", true, 20, "John", false ];
```

- *splice() - toSpliced()*
  - The splice() method changes the **contents** of an array by **removing** or **replacing** existing elements **and/or adding new elements** in place.

```
var removed = arr.splice(0, 1);
console.log(arr); //[ 'Mike', true, 20, 'John', false ]
console.log(removed); //[ 10 ]

var removed = arr.splice(1, 3);
console.log(arr); //[ 10, 'John', false ]
console.log(removed); //[ 'Mike', true, 20 ]

var removed = arr.splice(1, 0, "new Item");
console.log(arr); //[ 10, 'new Item', 'Mike', true, 20, 'John', false ]
console.log(removed); //[ ]
```

Array - Methods (at)

```
var arr = [ 10, "Mike", true, 20, "John", false ];
```

- *at()*
  - The at() method takes an **integer** value and **returns the item at that index**, allowing for **positive** and **negative** integers. Negative integers count back from the last item in the array.

```
var elem = arr.at( 1 );
console.log(elem); // "Mike"
```

```
var elem = arr.at( );
console.log(elem); // 10
```

```
var elem = arr.at( -2 );
console.log(elem); // "John"
```

```
var elem = arr.at( 7 );
console.log(elem); // "undefined"
```

Array - Methods (includes)

```
var arr = [ 10, "Mike", true, 20, "John", false ];
```

- *includes()*
  - The includes() method determines whether an array **includes a certain value** among its entries, returning **true** or **false** as appropriate.

```
var hasElement = arr.includes( "Mike" );
console.log(hasElement); // true

var hasElement = arr.includes( "test" );
console.log(hasElement); // false


var hasElement = arr.includes( "10" );
console.log(hasElement); // false
```

Array - Methods (indexOf)

```
var arr = [ 10, "Mike", true, 20, "John", false, "Mike" ];
```

- *indexOf()*
  - The indexOf() method returns the **first index** at which a given element can be found in the array, or **-1** if it is not present.

```
var index = arr.indexOf( "Mike" );
console.log(index); // 1

var index = arr.indexOf( "Mike", 2 );
console.log(index); // 6


var index = arr.indexOf( "test" );
console.log(index); // -1
```

Array - Methods (fill)

```
var arr = [ 10, "Mike", true, 20, "John", false ];
```

- *fill()*
  - The fill() method **changes all elements** in an array to a static value. It **returns the modified array**.

```
arr.fill("test");
console.log(arr); //[ 'test', 'test', 'test', 'test', 'test', 'test' ]

arr.fill("test", 2);
console.log(arr); //[ 10, 'Mike', 'test', 'test', 'test', 'test' ]

arr.fill("test", 2, 4);
console.log(arr); //[ 10, 'Mike', 'test', 'test', 'John', false ]
```

Array - Methods (forEach, map)

```
var arr = [ 10, 20, 30, 40, 50, 60 ];
```

- *forEach()*
  - The forEach() method executes a provided function once for each array element.

```
arr.forEach(function (item) {
        console.log(item);  //10, 20, 30, 40, 50, 60
});
```

- *map()*
  - The map() method **creates a new array** populated with the results of calling a provided function on every element in the calling array.

```
var newArr = arr.map(function (item) {
   return item * 2;
});
console.log(newArr); //[20, 40, 60, 80, 100, 120]
```

Array - Methods (filter, reverse)

```
var arr = [ 10, "Mike", true, 20, "John", false ];
```

- *filter()*
  - The filter() method creates a copy of a portion of a given array, filtered down to just the elements from the given array that pass the test implemented by the provided function.

```
var newArr = arr.filter(function (item) {
   return typeof item == "number";
});
console.log(newArr); //[10, 20]
```

- *reverse() - toReversed()*
  - The reverse() method reverses an array in place and returns the reference to the same array, the first array element now becoming the last, and the last array element becoming the first.

```
arr.reverse();
console.log(arr); //[false, 'John', 20, true, 'Mike', 10]
```

Array - Methods (every, some)

```
var arr = [ 10, "Mike", true, 20, "John", false ];
```

- *every()*
  - The every() method tests whether all elements in the array pass the test implemented by the provided function. It returns a **Boolean** value.

```
var isArrayOfNumbers = arr.every(function (item) {
   return typeof item == "number";
});
console.log(isArrayOfNumbers); //false;
```

- *some()*
  - The some() method tests whether at least one element in the array passes the test implemented by the provided function. It returns **true** if, in the array, otherwise it returns **false**. It doesn't modify the array.

```
var hasStringItem = arr.some(function (item) {
   return typeof item == "string";
});
console.log(hasStringItem); //true;
```

Array - Methods (reduce)

- The **reduce()** method executes a user-supplied "reducer" **callback function** on **each** element of the array, in order, **passing in** the return value from the **calculation on the preceding element**. The final result of running the reducer across all elements of the array is a **single value**.
  - ```
    var arr = [1, 2, 3, 4];
    var initialValue = 0;

    var sumOfElements = arr.reduce(function(previousValue, currentValue, index,
        notModifiedArray) {
            return previousValue + currentValue;
        }, initialValue);

    console.log('sumOfElements: ', sumOfElements); //10
    ```
    ```

Array - Methods (sort, toSorted)

- The **sort()** method sorts the elements of an array and returns the reference to the same array, now sorted. The default sort order is **ascending**, built upon converting the elements into strings, then comparing their sequences of **UTF-16** code units values.

    - ```
      var arr = [ 1, 4, 2, 7, 6, 3, 5 ];
      arr.sort();
      console.log(arr); //[ 1, 2, 3, 4, 5, 6, 7 ]
      ```

    - ```
      var arr = [ 1, 4, 2, 7, 6, 3, 5, 10 ];
      arr.sort();
      console.log(arr); //[ 1, 10, 2, 3, 4, 5, 6, 7 ]
      ```

Array - Methods (sort, toSorted)

- var arr = [1, 4, 2, 7, 6, 3, 5, 10, 5];

```
arr.sort(function (a, b) {
    if (a > b) {
        return 1;
    } else if(a < b) {
        return -1;
    } else {
        return 0;
    }
});
console.log(arr); //[1, 2, 3, 4, 5, 5, 6, 7, 10]
```

compareFn(a, b) value return	sort order
> 0	sort **a** after **b**
< 0	sort **b** before **a**
=== 0	keep original order of **a** and **b**

Function - (properties, methods)

- *call method*
  - The **call()** method calls the function with a given **this** value and arguments provided individually.

```
var user = {
        name: "John",
        age: 20,
};

function foo( ) {
        console.log(this.age);
}
foo.call(user);
```

Function - (properties, methods)

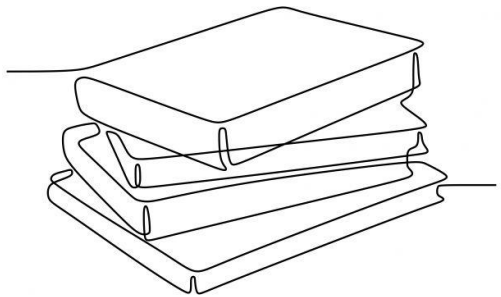
- *apply method*
  - The **apply()** method calls the function with a given **this** value and arguments provided individually.

```
var user = {
        name: "John",
        age: 20,
};
var data = [ { pass: 123, isAdmin: true } ];

function foo( data ) {
        this.data = data;
}
foo.apply(user, data);
```

Function - (properties, methods)

- *length property*
  - A Function object's **length** property indicates the number of parameters expected by the function.

```
function foo(a, b) {
      console.log(foo.length);
}

foo(); //2
```



The arguments object

Arguments is an **Array-like object** accessible inside functions that contains the values of the arguments passed to that function.

```
function foo(a, b) {
    console.log(arguments);
    console.log(arguments.length);
    console.log(arguments[2]);
}

foo(5, 6, "text", "8", { a: "a", b: "b" }, true);
```

05

Section

- Immediately Invoked Function Expression (IIFE)
- Strict Mode

Immediately Invoked Function Expression

An **IIFE** (Immediately Invoked Function Expression) is an ECMAScript function **that runs as soon as it is defined**. The name IIFE is promoted by Ben Alman.

- ```
  (function () {
      console.log('This will never run again!');
  })();
  ```

Advantages of IIFE:

- ❖ Do not create unnecessary global variables and functions
- ❖ Functions and variables defined in IIFE do not conflict with other functions & variables even if they have same name.
- ❖ Organize JavaScript code.
- ❖ Make JavaScript code maintainable.
  ```

Immediately Invoked Function Expression


```
function some() {
        var arr = [ ];
        for (var i = 0; i < 2; i++) {
                arr.push(function () {
                        console.log(i);
                });
        }
        return arr;
}
var arrayFunctions = some();
arrayFunctions[0](); //2
arrayFunctions[1]();  //2
```

Immediately Invoked Function Expression


- ```
  function some() {
      var arr = [ ];
      for (var i = 0; i < 2; i++) {
          (function(i) {
              arr.push(function () {
                  console.log(i);
              });
          })(i);
      }
      return arr;
  }
  var arrayFunctions = some();
  arrayFunctions[0](); //0
  arrayFunctions[1](); //1
  ```
  ```

Strict Mode

ECMAScript 5 was the first to introduce the concept of **strict mode**. Strict mode allows you to opt-in to stricter checking for JavaScript error conditions either globally or locally within a single function. The advantage of strict mode is that you'll be informed of errors earlier, so some of the ECMAScript quirks that cause programming errors will be caught immediately.

- *Global*
  - ```
    "use strict";
    console.log("Strict mode is ON!");
    ```
- *Local*
  - ```
    function doSomething() {
        "use strict";
        console.log("Strict mode is ON for doSomething function");
    }
    ```

Strict Mode - variables

- *Normal Mode*
  - message = "Hello, world!";
    console.log(message); //"Hello, world!"
- *Strict Mode*
  - "use strict";
    message = "Hello, world!";
    console.log(message); //ReferenceError: message is not defined

- *Normal Mode*
  - var message = "Hello, world!";
    delete message; //It will work correctly
- *Strict Mode*
  - "use strict";
    var message = "Hello, world!";
    delete message; // SyntaxError: Delete of an unqualified identifier in strict mode.

Strict Mode - functions

- *Normal Mode*
  - ```
    function some(param, param) {
        console.log(param);
    }
    some("test", false); //false
    ```

- *Strict Mode*
  - ```
    "use strict";
    function some(param, param) {
        console.log(param);
    }
    some("test", false); //SyntaxError: Duplicate parameter name not allowed in this context.
    ```

Strict Mode - functions

- *Normal Mode*
  - ```
    function some(param) {
        param =  "Mike";
        console.log(param); //"Mike"
        console.log(arguments[0]); //"Mike"
    }
    some("John");
    ```
- *Strict Mode*
  - ```
    "use strict";
    function some(param) {
        param =  "Mike";
        console.log(param); //"Mike"
        console.log(arguments[0]); //"John"
    }
    some("John");
    ```

06

Section

- Properties and methods of global object
- Object Number
- Object String
- Object Math
- Object RegExp (Regular Expression)

Properties And Methods Of Global Object

- *Infinity / -Infinity*
  - **Infinity** is a property of the **global object**. In other words, it is a **variable** in **global scope**.
  - The global property Infinity is a **numeric** value representing infinity.

```
console.log(Infinity); //Infinity
console.log(Infinity + 1); //Infinity
console.log(1 / 0); //Infinity
console.log(-1 / 0); //-Infinity
console.log(1 / Infinity); //0
```

Properties And Methods Of Global Object

- *isFinite()*
  - The **global isFinite()** function determines whether the passed value is a **finite number**. If needed, the parameter is first **converted to a number**.

```
console.log(isFinite(Infinity)); //false
console.log(isFinite(-Infinity)); //false
console.log(isFinite(NaN)); //false
console.log(isFinite(undefined)); //false
console.log(isFinite({ })); //false
console.log(isFinite(10)); //true
console.log(isFinite(0)); //true
console.log(isFinite("0")); //true
console.log(isFinite([ ])); //true
```

Properties And Methods Of Global Object

- *NaN*
  - The global **NaN** property is a value representing **Not-A-Number**. There are five different types of operations that return NaN.
    1. Failed number conversion(e.g. parseInt( ), parseFloat( ), Number( )....).
    2. Math operation where the result is not a real number(e.g. Math.sqrt(-1)...).
    3. Indeterminate form(e.g. 0 * Infinity, Infinity - Infinity....).
    4. A method or expression whose operand is or gets coerced to NaN(e.g. 7 * NaN, "blabla" * 10....).
    5. Other cases where an invalid value is to be represented as a number (e.g. an invalid Date....).
  - NaN's behaviors include:
    - ➔ If NaN is involved in a mathematical operation the result is usually also NaN
    - ➔ When NaN is one of the operands of any relational comparison (>, <, >=, <=), the result is always false.
    - ➔ NaN compares unequal (via ==, !=, ===, and !==) to any other value — including to another NaN value.

Properties And Methods Of Global Object

- *isNaN()*
  - isNaN() will return **true** if the **value is currently NaN**, or if it is **going to be NaN** after it is **coerced to a number**.

```
console.log(isNaN(NaN)); //true
console.log(isNaN(10 * "Hello")); //true
console.log(isNaN("Hello")); //true
console.log(isNaN(10)); //false
console.log(isNaN(true)); //false
```

Properties And Methods Of Global Object

- *parseInt()*
  - The **parseInt()** function parses a string argument and **returns an integer** of the specified **radix** (the base in mathematical numeral systems).

```
console.log(parseInt(10)); //10
console.log(parseInt(10.57)); //10
console.log(parseInt("10")); //10
console.log(parseInt("FF", 16)); //255
console.log(parseInt("0xF", 16)); //15

console.log(parseInt("Hello", 8)); //NaN
console.log(parseInt("234", 2)); //NaN
```

Properties And Methods Of Global Object

- *parseFloat()*
  - The **parseFloat()** function parses a string argument and **returns a floating point number**.

```
console.log(parseFloat(3.14)); //3.14
console.log(parseFloat("3.14")); //3.14
console.log(parseFloat("314e-2")); //3.14

console.log(parseFloat("FF2")); //NaN
console.log(parseFloat("NaN")); //NaN
```

Object Number

- *Static properties*
  - Number.NEGATIVE_INFINITY //-Infinity
  - Number.POSITIVE_INFINITY //Infinity
  - Number.MIN_VALUE //5e-324
  - Number.MAX_VALUE //1.7976931348623157e+308
  - Number.MIN_SAFE_INTEGER //-9007199254740991 (-((2 ** 53) - 1))
  - Number.MAX_SAFE_INTEGER //9007199254740991 ((2 ** 53) - 1)
  - Number.NaN

Object Number

- *Number methods*

  - Number.isFinite( )

```
console.log(Number.isFinite(10)); //true
console.log(Number.isFinite("0")); //false
```

  - Number.isInteger( )

```
console.log(Number.isInteger(10)); //true
console.log(Number.isInteger(10.0)); //true
console.log(Number.isInteger(10.1)); //false
```

  - Number.isNaN( )

```
console.log(Number.isNaN(NaN)); //true
console.log(Number.isNaN("Hello")); //false
```

Object Number

- *Number methods*
  - toFixed( ) - The toFixed() method formats a number using fixed-point notation.
    - ➢
```
var num = 10;
num.toFixed(); //"10"
num.toFixed(2); //"10.00"
```
    - ➢
```
var num = 123.456;
num.toFixed(1); //"123.5"
num.toFixed(5); //"123.45600"
```
  - toString( ) - The toString() method returns a string representing the specified number value.
    - ➢
```
var num = 10;
num.toString(); // "10"
num.toString(16); // "a"
```

Object String

- The String object is used to represent and manipulate a sequence of characters.
  - *String properties*
    - ➢ The **length** read-only property of a string contains the length of the string in UTF-16 code units.

```
var str = "Hello";
console.log(str.length); //5
```

  - *String methods*
    - ➢ The **concat()** method concatenates the string arguments to the calling string and returns a new string.

```
var str = "Hello";
var result = str.concat(" , ", " world!");
console.log(result); //"Hello, world!"
```

Object String

- *String methods*

  - The **toLowerCase()** method returns the calling string value converted to lowercase.
  - The **toUpperCase()** method returns the calling string value converted to uppercase.

```
var sentence = "The quick brown fox jumps over the lazy dog.";
sentence.toLowerCase(); //"the quick brown fox jumps over the lazy dog."
sentence.toUpperCase(); //"THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG."
```

Object String

- ***String methods***

```
var string = "Hello";
```

➢ **charAt()** method returns a **new string** consisting of the single UTF-16 code unit located at the specified offset into the string.

```
console.log(string.charAt(3)); //"l"
console.log(string.charAt()); //"H"
console.log(string.charAt(10)); //" "
```

➢ The **charCodeAt()** method returns an integer between 0 and 65535 representing the UTF-16 code unit at the given index.

```
console.log(string.charCodeAt(1)); //101
console.log(string.charCodeAt()); //72
console.log(string.charCodeAt(10)); //NaN
```

Object String

- *String methods*
  - The static **String.fromCharCode()** method returns a string created from the specified sequence of UTF-16 code units.

```
String.fromCharCode(101); //"e"
String.fromCharCode(72, 101, 108, 108, 111); //"Hello"
```

  - The **endsWith()** method determines whether a string ends with the characters of a specified string, returning **true** or **false** as appropriate.

```
var string = "Dogs are the best!";

console.log(string.endsWith("best!")); //true
console.log(string.endsWith("best")); //false
```

Object String

- *String methods*

```
var greeting = "   Hello world!   ";
```

➢ The **trim()** method removes whitespace from both ends of a string and returns a new string, without modifying the original string.

```
console.log(greeting.trim()); //"Hello world!"
```

➢ The **trimEnd()** method removes whitespace from the end of a string. trimRight() is an alias of this method.

```
console.log(greeting.trimEnd()); //"    Hello world!"
```

➢ The **trimStart()** method removes whitespace from the beginning of a string.

```
console.log(greeting.trimStart()); //"Hello world!  "
```

Object String

- *String methods*

```
var str = "This is my text";
```

➢ The **slice()** method extracts a section of a string and **returns** it as **a new string**, **without modifying** the original string.

```
console.log(str.slice()); // "This is my text"
console.log(str.slice(5)); // "is my text"
console.log(str.slice(5, 7)); // "is"
console.log(str.slice(7, 5)); // " "
console.log(str.slice(-7, -5)); // "my"
```

➢ The **substring()** method returns the part of the string between the start and end indexes, or to the end of the string.

```
console.log(str.substring()); // "This is my text"
console.log(str.substring(5)); // "is my text"
console.log(str.substring(5, 7)); // "is"
console.log(str.substring(7, 5)); // "is"
console.log(str.substring(-7, -5)); // " "
```

Object String

- *String methods*

```
var str = "This is my text";
```

  - The **indexOf()** method, given one argument: a substring to search for, searches the entire calling string, and returns the index of the **first** occurrence of the specified substring. Given a second argument: a number, the method returns the first occurrence of the specified substring at an index greater than or equal to the specified number.

```
console.log(str.indexOf("is")); //2
console.log(str.indexOf("is", 3)); //5
console.log(str.indexOf("some")); //-1
```

  - The **lastIndexOf()** method, given one argument: a substring to search for, searches the entire calling string, and returns the index of the **last** occurrence of the specified substring.

```
console.log(str.lastIndexOf("is")); //5
console.log(str.lastIndexOf("is", 4)); //2
```

Object String

- *String methods*
  - The **replace()** method returns a **new string** with one, some, or all matches of a **pattern** replaced by a replacement.

```
var str = "This is some text";
var newStr = str.replace("some", "long");
console.log(newStr); //"This is long text"
```

  - The **split()** method takes a pattern and divides a String into an ordered list of substrings by searching for the pattern, puts these substrings into an array, and **returns the array**.

```
var str = "This is some text";
str.split(" "); //['This', 'is', 'some', 'text']
str.split(" ", 2); //['This', 'is']
```

Object Math

- **Math** is a built-in object that has properties and methods for mathematical constants and functions. Math works with the Number type.

➢ *Properties*

- **Math.E** - The Math.E property represents Euler's number, the base of natural logarithms, e, which is approximately **2.718**.

$$Math.E = e \approx 2.718$$

- **Math.LN10** - The Math.LN10 property represents the natural logarithm of 10, approximately **2.302**.

$$Math.LN10 = \ln(10) \approx 2.302$$

- **Math.LN2** - The Math.LN2 property represents the natural logarithm of 2, approximately **0.693**.

$$Math.LN2 = \ln(2) \approx 0.693$$

Object Math

- ***Properties***
  - **Math.LOG10E** - The Math.LOG10E property represents the base 10 logarithm of e, approximately **0.434**.

$$Math.LOG10E = \log_{10}(e) \approx 0.434$$

  - **Math.LOG2E** - The Math.LOG2E property represents the base 2 logarithm of e, approximately **1.442**.

$$Math.LOG2E = \log_{2}(e) \approx 1.442$$

  - **Math.PI** - The Math.PI property represents the ratio of the circumference of a circle to its diameter, approximately **3.14159**.

$$Math.PI = \pi \approx 3.14159$$

Object Math

- *Properties*
  - **Math.SQRT1_2** - The Math.SQRT1_2 property represents the square root of 1/2 which is approximately **0.707**.

$$Math.SQRT1\_2 = \frac{\sqrt{1}}{2} \approx 0.707$$

  - **Math.SQRT2** - The Math.SQRT2 property represents the square root of 2, approximately **1.414**.

$$Math.SQRT2 = \sqrt{2} \approx 1.414$$

Object Math

- *Methods*
  - **Math.ceil()** - The Math.ceil() function always **rounds up** and returns the smaller integer **greater than** or **equal** to a given number.

```
console.log(Math.ceil(7.004)); //8
console.log(Math.ceil(-7.004)); //-7
console.log(Math.ceil(4)); //4
```

  - **Math.floor()** - The Math.floor() function always **rounds down** and returns the largest integer **less than** or **equal** to a given number.

```
console.log(Math.floor(7.004)); //7
console.log(Math.floor(-7.004)); //-8
```

  - **Math.round()** - The Math.round() function returns the value of a number **rounded to the nearest integer**.

```
console.log(Math.round(3.4)); //3
console.log(Math.round(3.5)); //4
```

Object Math

- *Methods*
  - **Math.min()** - The Math.min() function returns **the smallest** of the numbers given as input parameters, or **Infinity** if there are no parameters.

```
console.log(Math.min(2, 3, 1)); //1
console.log(Math.min(-2, -3, -1)); //-3
```

  - **Math.max()** - The Math.max() function returns **the largest** of the numbers given as input parameters, or **-Infinity** if there are no parameters.

```
console.log(Math.max(2, 3, 1)); //3
console.log(Math.max(-2, -3, -1)); //-1
```

  - **Math.random()** - The Math.random() function returns a **floating-point**, pseudo-random number that's greater than or equal to **0** and less than **1**, with approximately uniform distribution over that range — which you can then scale to your desired range.

```
console.log(Math.random()); // e.g. 0.04779097114189357.....
```

Object Math

- *Methods*
  - **Math.abs()** - The Math.abs() function returns the **absolute** value of a number.

```
console.log(Math.abs(1)); //1
console.log(Math.abs(-1)); //1
console.log(Math.abs([ -2 ])); //2
console.log(Math.abs({ })); //NaN
```

  - **Math.sign()** - The Math.sign() function returns **1** or **-1**, indicating the sign of the number passed as argument. If the input is **0** or **-0**, it will be returned **as-is**.

```
console.log(Math.sign(5)); //1
console.log(Math.sign(-5)); //-1
console.log(Math.sign(0)); //0
console.log(Math.sign(-0)); //-0
```

Object Math

- *Methods*
  - **Math.sqrt()** - The Math.sqrt() function returns the **square root of a number**.

```
console.log(Math.sqrt(0)); //0
console.log(Math.sqrt(16)); //4
console.log(Math.sqrt(-16)); //NaN
```

  - **Math.cbrt()** - The Math.cbrt() function returns the **cube root of a number**.

```
console.log(Math.cbrt(0)); //0
console.log(Math.cbrt(8)); //2
console.log(Math.cbrt(-8)); //-2
```

  - **Math.pow()** - The Math.pow() method returns the **value of a base raised to a power**.

```
console.log(Math.pow(2, 3)); //8
console.log(Math.pow(7, -2)); //0.02040816326530612, 1/49
console.log(Math.pow(2, NaN)); //NaN
console.log(Math.pow(NaN, 2)); //NaN
```

Object RegExp (Regular Expression)

- Regular expressions are **patterns** used to **match** character combinations in **strings**. In ECMAScript, regular expressions are also **objects**.

```
var string = "This is some text";
var regExp = /pattern/;
```

  - ***Object String methods***
    - string.search(regExp);
    - string.replace(regExp, " ");
    - string.split(regExp);
    - string.match(regExp);
  - ***Object RegExp methods***
    - regExp.test(string);
    - regExp.exec(string);

Object RegExp (Regular Expression)

```
var string = "myEmail@volo.global";
var regExp = /@/;
```

- **search()** - The search() method executes a search for a match between a regular expression and string and **return the index of the first match or -1 if no match was found**.
    - `var index = string.search(regExp);`
    - `console.log(index); //7`
- **test()** - The test() method executes a search for a match between a regular expression and a specified string. **Returns true or false**.
    - `var isContain = regExp.test(string);`
    - `console.log(isContain); //true`

Object RegExp (Regular Expression)

```
var string = "myEmail@volo.global";
var regExp = /gmail|volo|yahoo/; //or
```

- ```
  var index = string.search(regExp);
  console.log(index); //8
  ```

- ```
  var isContain = regExp.test(string);
  console.log(isContain); //true
  ```

```
var string = "/";
var regExp = /\/;
```

- ```
  var index = string.search(regExp);
  console.log(index); //0
  ```

- ```
  var isContain = regExp.test(string);
  console.log(isContain); //true
  ```

Object RegExp (Regular Expression)

```
var string = "awesome text";
var regExp = /[abc]/;

var index = string.search(regExp);
var isContain = regExp.test(string);
console.log(index); //0
console.log(isContain); //true
```

```
var string = "abc";
var regExp = /[^abc]/; //except

var index = string.search(regExp);
var isContain = regExp.test(string);
console.log(index); //-1
console.log(isContain); //false
```

Object RegExp (Regular Expression)

```
var string = "aw3s0me Text";
var regExp = /[a-zA-Z0-9]/;

var index = string.search(regExp);
var isContain = regExp.test(string);
console.log(index); //0
console.log(isContain); //true
```

- . - matches any character except newlines
- \w - [a-zA-Z0-9_]
- \W - [^a-zA-Z0-9_]
- \d - [0-9]
- \D - [^0-9]
- \s - Unicode whitespace characters
- \S - Except Unicode whitespace characters

Object RegExp (Regular Expression)

```
var string = "awesome text";
```

- { n } - repeated n times.
  - `var regExp = /w{1}/;`
  - `regExp.test(string); //true`
- { n, } - repeated no less than n times.
  - `var regExp = /w{1,}/;`
  - `regExp.test(string); //true`
- { n, m } - repeated not less than n times but not more than m times.
  - `var regExp = /w{1,3}/;`
  - `regExp.test(string); //true`

- ? - Compatible with {0,1} //0 or 1
- + - Compatible with {1,} //1 or more
- * - Compatible with {0,} //0 or more

Object RegExp (Regular Expression)

- ^ - Search from beginning of line
- $ - Search to end of line

```
var string = "awesome text";
```

Flags

- *i - ignoreCase - Searches for both uppercase and lowercase letters*
  - `var regExp = /w/i;`
  - `regExp.test(string); //true`
- *g - global - Searches for all characters*
  - `var regExp = /e/g`
  - `regExp.test(string); //true`
- *m - multiline*
  - Searches in all lines(\n)

Object RegExp (Regular Expression)

- var string = "this is some text";
  var regExp = /^([a-z]+) ([a-z]+) ([a-z]+) ([a-z]+)$/;
- **match()** - The match() method retrieves the result of matching a string against a regular expression.
  - var result = string.match(regExp);
    console.log(result); //['this is some text', 'this', 'is', 'some', 'text'...]
- **exec()** - The exec() method executes a search for a match in a specified string and returns a result array, or null.
  - var result = regExp.exec(string);
    console.log(result); //['this is some text', 'this', 'is', 'some', 'text'...]
- **replace()**
  - var result = string.replace(regExp, '$2 $1 $3 $4');
  - console.log(result); //is this some text

07

Section

- OOP in JavaScript
- Constructor Function
- Prototypes
- Object.create( )
- Object Date
- Object Error

OOP (Constructor Function)

- ```
  function Person(firstName, birthDate) {
      this.firstName = firstName;
      this.birthDate = birthDate;
  }

  var person = new Person("John", 1998);
  console.log(person); //{ firstName: 'John', birthAge: 1998 }
  ```

- The **instanceof** operator tests to see if the **prototype** property of a constructor appears anywhere in the **prototype chain** of an object. The return value is a boolean value.

  ```
  person instanceof(Person) //true
  ```
  ```

OOP | Primitive Values (Number, String, Boolean)


- ```
  var num = 5;
  console.log(typeof num); //number
  var num = new Number(5);
  console.log(typeof num); //object
  ```
- ```
  var firstName = "John";
  console.log(typeof firstName); //string
  var firstName = new String("John");
  console.log(typeof firstName); //object
  ```
- ```
  var bool = true;
  console.log(typeof bool); //boolean
  var bool = new Boolean(true);
  console.log(typeof bool); //object
  ```
  ```

OOP (Object, Array)

JS

- ```
  var obj = { };
  var obj = new Object( );
  console.log(typeof obj); //object
  ```

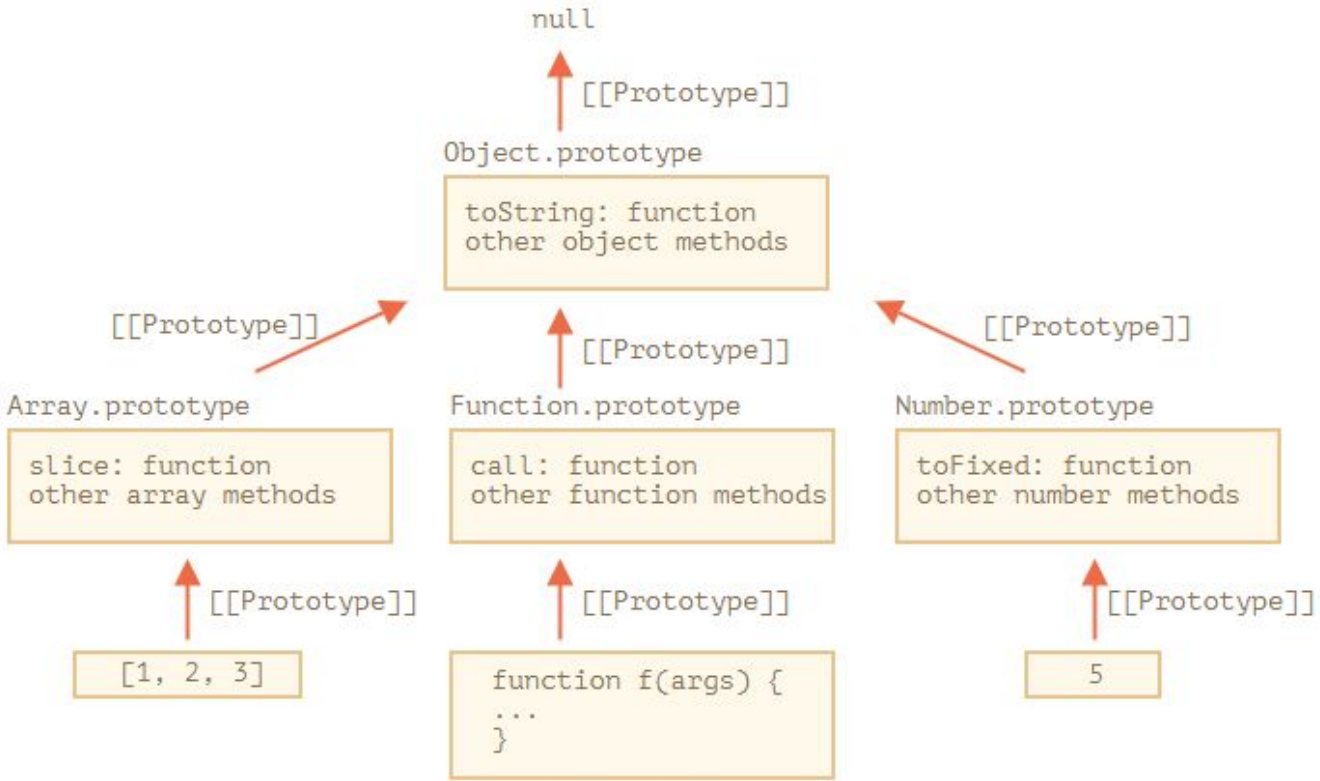
- ```
  var array = [ ];
  var array = new Array( );
  console.log(typeof array); //object
  console.log(Array.isArray(array)); //true

  var array = new Array(5);
  console.log(array); //[empty × 5]
  var array = new Array(5, "John", true);
  console.log(typeof array); //[ 5, "John", true ]
  ```

OOP (Prototypes)

- ```
  function Person(firstName, birthDate) {
      this.firstName = firstName;
      this.birthDate = birthDate;
  }
  Person.prototype.sayHello = function() {
      console.log("Hello from " + this.firstName + "!");
  };
  var person = new Person("John", 1894);
  person.sayHello(); //"Hello from John!"
  ```
- The **hasOwnProperty()** method returns a boolean indicating whether the object has the specified property as its own property.

  ```
  person.hasOwnProperty("sayHello"); //false
  person.hasOwnProperty("firstName"); //true
  ```
  ```

OOP (Object.create( ))

- The **Object.create()** method **creates** a **new object**, using an **existing** object as the **prototype** of the **newly created** object.
  -
```
const PersonProto = {
    greeting: function() {
        return `Hello from ${this.firstName}!`;
    },
    init: function(firstName, birthYear) {
        this.firstName = firstName;
        this.birthYear = birthYear;
    }
};
const personOne = Object.create(PersonProto);
personOne.init('John', 2000);
console.log(personOne);
```

OOP (Prototypal Inheritance)


- ```
  function Person(firstName, birthYear) {
      this.firstName = firstName;
      this.birthYear = birthYear;
  }
  function Student(firstName, birthYear, course) {
      Person.call(this, firstName, birthYear);
      this.course = course;
  }
  Student.prototype = Object.create(Person.prototype);
  Student.prototype.learning = function() {
      console.log(`${this.firstName} is starting learning...`);
  };
  const student = new Student('Mike', 2002, 'JavaScript');
  console.log(student);
  ```
  ```

OOP (Prototypal Inheritance)



Object Date

- **Date objects** represent a single moment in time in a platform-independent format. Date objects contain a Number that represents milliseconds **since 1 January 1970 UTC**.
  - ```
    var date = new Date();
    console.log(date); //Thu Nov 03 2022 00:36:56 GMT+0400
    ```

- **Date object** can accept the time given by us, in the following sequence: **year, month, day, hour, minute, millisecond**.
  - ```
    var date = new Date(1992, 05, 10, 15, 37, 03);
    console.log(date); //Wed Jun 10 1992 15:37:03 GMT+0400
    ```

    Note: Counting **months** starts from **0**.

Object Date | get

```
var date = new Date(1992, 05, 10);
```

- The **getDate()** method returns the **day of the month** for the specified date according to **local time**.
  - `console.log(date.getDate()); //10`
- The **getDay()** method returns the **day of the week** for the specified date according to **local time**, where **0** represents **Sunday**.
  - `console.log(date.getDay()); //3`
- The **getFullYear()** method returns the **year** of the specified date according to **local time**.
  - `console.log(date.getFullYear()); //1992`
- The **getMonth()** method returns the **month** in the specified date according to **local time**, where **0** indicates the **first month of the year**.
  - `console.log(date.getMonth()); //5`

Object Date | get

```
var date = new Date(1992, 05, 10, 15, 43, 28);
```

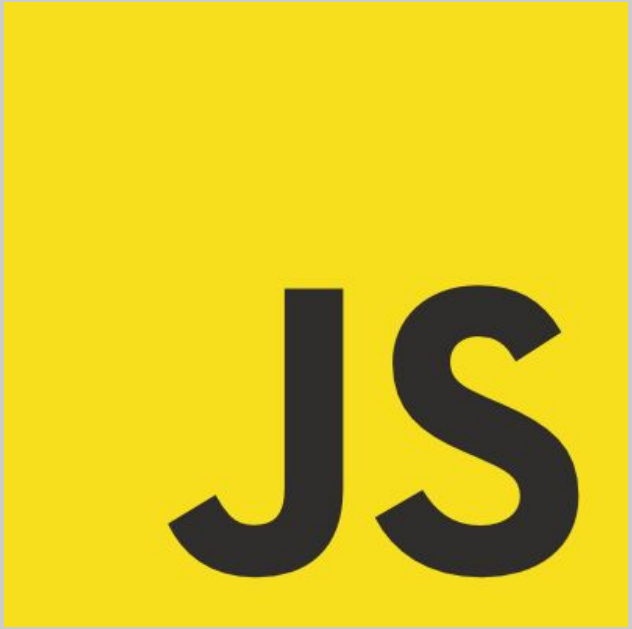
- The **getHours()** method returns the **hour** for the specified date, according to **local time**.
  - `console.log(date.getHours()); //15`
- The **getMinutes()** method returns the **minutes** in the specified date according to **local time**.
  - `console.log(date.getMinutes()); //43`
- The **getSeconds()** method returns the **seconds** in the specified date according to **local time**.
  - `console.log(date.getSeconds()); //28`
- The **getMilliseconds()** method returns the **milliseconds** in the specified date according to **local time**.
  - `console.log(date.getMilliseconds()); //0`

Object Date | set

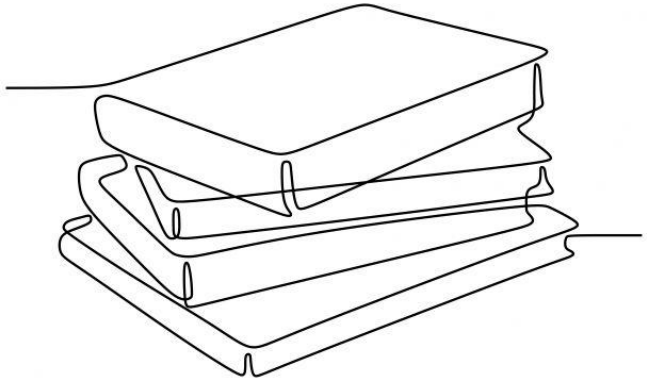
```
var date = new Date(1992, 05, 10);
```

- The **setDate()** method **changes** the day of the month of a given Date instance, based on **local time.**
  - ```
    date.setDate(14); //708522208000
    console.log(date); //Sun Jun 14 1992 15:43:28 GMT+0400
    ```

- date.setDay( )
- date.setFullYear( )
- date.setMonth( )
- date.setHours( )
- date.setMinutes( )
- date.setSeconds( )
- date.setMilliseconds( )
    ```

Object Date

```
var date = new Date(1992, 05, 10, 15, 43, 28);
```

- The static **Date.now()** method returns the number of **milliseconds** elapsed since **January 1, 1970 00:00:00 UTC**.
  - `var dateInMilliseconds = Date.now();`
  - `console.log(dateInMilliseconds); //1667423748056`
- The **Date.parse()** method parses a **string** representation of a date, and returns the **number of milliseconds** since **January 1, 1970, 00:00:00 UTC**.
  - `var dateInMilliseconds = Date.parse(date);`
  - `console.log(dateInMilliseconds); //708176608000`
- The **toDateString()** method returns the date portion of a Date object interpreted in the l**ocal timezone in English**.
  - `console.log(date.toDateString()); //Wed Jun 10 1992`

Object Error

- The **Error()** constructor creates an error object.
  - ```
    var error = new Error("Error text here!");
    console.log(error); //Error: Error text here!
    ```
- The **throw** statement throws a user-defined exception. Execution of the current function will stop, and control will be passed to the first **catch** block in the call stack. If no catch block exists among caller functions, the program will terminate.
  - ```
    var numOne = 10;
    var numTwo = 0;
    if(numTwo == 0) {
       throw new Error("Number cannot be divided by 0");
                                        //Uncaught Error: Number cannot be divided by 0
    } else {
       numOne / numTwo;
    }
    ```

Object Error | Handling

- The **try...catch** statement is comprised of a **try block** and either a **catch block**, a **finally block**, or both. The code in the try block **is executed first**, and if it **throws** an exception, the code in the **catch block will be executed**. The code in the **finally block will always be executed** before control flow exits the entire construct.
  - ```
    try {
        nonExistingFunction();
    } catch(error) {
        console.log(error); //ReferenceError: nonExistingFunction is not defined
        console.log(error.name); //ReferenceError
        console.log(error.message); //nonExistingFunction is not defined
    } finally {
        console.log('Error handled!');
    }
    ```
    ```



Browser Object Model(BOM)



01

Section

- A brief introduction to BOM
- Object History
- Object Screen
- Object Location
- Popup Boxes(alert, prompt, confirm)
- Window properties
- setTimeout(), setInterval()

Browser Object Model (BOM) | History

- The **Window object** provides access to the browser's session history through the **history object**. It exposes useful methods and properties that let you **navigate back and forth through the user's history**, and manipulate the contents of the history stack.
    - ***window.history.back()*** - To move backward through history. This acts exactly as if the user clicked on the **Back** button in their **browser toolbar**.
    - ***window.history.forward()*** - To move forward through history. Similarly, you can move forward (as if the user clicked the Forward button).

    - ***window.history.length*** - returns an integer representing the number of elements in the session history, including the currently loaded page **(for a page loaded in a new tab this property returns 1)**.

Browser Object Model (BOM) | History

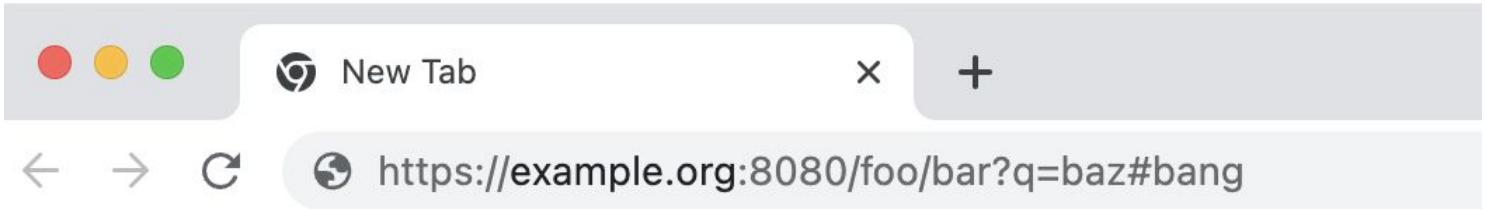
- ***window.history.go()*** - loads a specific page from the session history. You can use it to move **forwards** and **backwards** through the history **depending on the value of a parameter**.

  - To **move back** one page (the equivalent of calling back()):
    - window.history.go(-1);
  - To **move forward** a page, just like calling forward():
    - window.history.go(1);
  - To move forward two pages:
    - window.history.go(2);
  - To **reload** the current page:
    - window.history.go( );
    - window.history.go(0);

Browser Object Model (BOM) | Screen

- The Window property **screen** returns a reference to the screen object associated with the window. The screen object, implementing the Screen interface, is a special object for inspecting properties of the screen on which **the current window is being rendered**.
  - window.screen.width;
    - The **screen.width** read-only property returns the width of the screen in CSS pixels.
  - window.screen.height;
    - The **screen.height** read-only property returns the height of the screen in pixels.
  - window.screen.availWidth;
    - The **screen.availWidth** property returns the amount of horizontal space (in pixels) available to the window.
  - window.screen.availHeight;
    - The **screen.availHeight** property returns the height, in CSS pixels, of the space available for Web content on the screen.

Browser Object Model (BOM) | Location



- The **window.location** read-only property returns a Location object with information about the current location of the document.
    - window.location.href - "https://example.org:8080/foo/bar?q=baz#bang"
    - window.location.origin - "https://example.org:8080"
    - window.location.pathname - "/foo/bar"
    - window.location.search - "?q=baz"
    - window.location.hash - "#bang"
    - window.location.protocol - "https:"
    - window.location.host - "example.org:8080"
    - window.location.hostname - "example.org"
    - window.location.port - "8080"

Browser Object Model (BOM) | Location

- The **assign()** method causes the window to load and display the document at the URL specified. After the navigation occurs, the user can **navigate back** to the page.
    - `window.location.assign("https://www.google.com");`
- The **replace()** method of the Location replaces the current resource with the one at the provided URL. After the navigation occurs, the user can't **navigate back** to the page.
    - `window.location.replace("https://www.google.com");`
- The **reload()** method reloads the current URL, like the Refresh button.
    - `window.location.reload();`

Browser Object Model (BOM) | Popup Boxes

- **window.alert()** instructs the browser to display a dialog with an optional message, and to wait until the user dismisses the dialog.
  - window.alert("Hello, world!");
- **window.prompt()** instructs the browser to display a dialog with an optional message prompting the user to input some text, and to wait until the user either submits the text or cancels the dialog.
  - window.prompt("Type text here....", "default text");
  - OK - value
  - Cancel - null
- **window.confirm()** instructs the browser to display a dialog with an optional message, and to wait until the user either confirms or cancels the dialog.
  - window.confirm("Are you sure?");
  - OK - true,
  - Cancel - false

Browser Object Model (BOM) | Window properties

- The **window.screenX** read-only property returns the **horizontal distance**, in CSS pixels, of the **left border** of the user's browser viewport to the **left side of the screen**.
  - window.screenX;
- The **window.screenY** read-only property returns the **vertical distance**, in CSS pixels, of the **top border** of the user's browser viewport to the **top edge of the screen**.
  - window.screenY;
- The read-only Window property **innerWidth** returns the interior width of the window in pixels. This includes the width of the vertical scroll bar, if one is present.
  - window.innerWidth;
- The read-only **innerHeight** property of the Window interface returns the interior height of the window in pixels, including the height of the horizontal scroll bar, if present.
  - window.innerHeight;

Browser Object Model (BOM) | Window methods

- The **open()** method of the Window interface loads a specified resource into a new or existing browsing context under a specified name.
  - window.open();
  - window.open("https://www.google.com");
  - window.open("https://www.google.com","tabName","width=200 height=200 top=100 left=10");
- The **window.close()** method closes the current window, or the window on which it was called.
  - window.close();

Browser Object Model (BOM) | Window methods

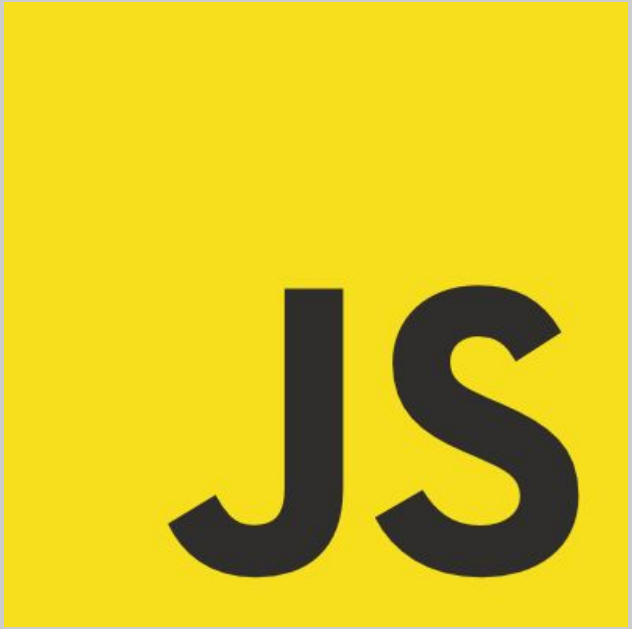
- The **moveTo()** method of the Window interface moves the current window to the specified **coordinates**.
  - window.moveTo(X, Y);
- The **moveBy()** method of the Window interface moves the current window by a specified **amount**.
  - window.moveBy(deltaX, deltaY);
- The **window.resizeTo()** method dynamically resizes the window.
  - window.resizeTo(width, height);
- The **window.resizeBy()** method resizes the current window by a **specified amount.**
  - window.resizeBy(xDelta, yDelta);

Browser Object Model (BOM) | Window properties

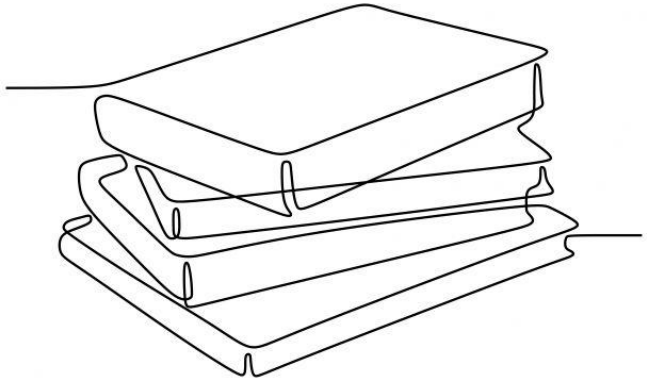
- The **scroll()** method scrolls the element to a particular set of coordinates inside a given element.
  - window.scroll(X, Y);
- The **scrollTo()** method scrolls to a particular set of coordinates inside a given element.
  - window.scrollTo(X, Y);
  - window.scrollTo({
    top: 200,
    left: 0,
    behavior: "instant"
    });
- The **scrollBy()** method of the Element interface scrolls an element by the given amount.
  - window.scrollBy(X, Y);
  - window.scrollBy({
    top: 200,
    left: 0,
    behavior: "smooth"
    });

Browser Object Model (BOM) | Timers

- The global **setTimeout()** method sets a timer which executes a function or specified piece of code once the timer expires.
  - ```
    var timer = setTimeout(function(param) {
        console.log(param); //"test"
        console.log("Hello, world!");
    }, 5000, "test");
    clearTimeout(timer);
    ```
- The **setInterval()** method **repeatedly** calls a function or executes a code snippet, with a **fixed time delay between each call**.
  - ```
    var timer = setInterval(function(param) {
        console.log("Hello, world!");
    }, 5000, "test");
    clearInterval(timer);
    ```



Document Object Model(DOM)

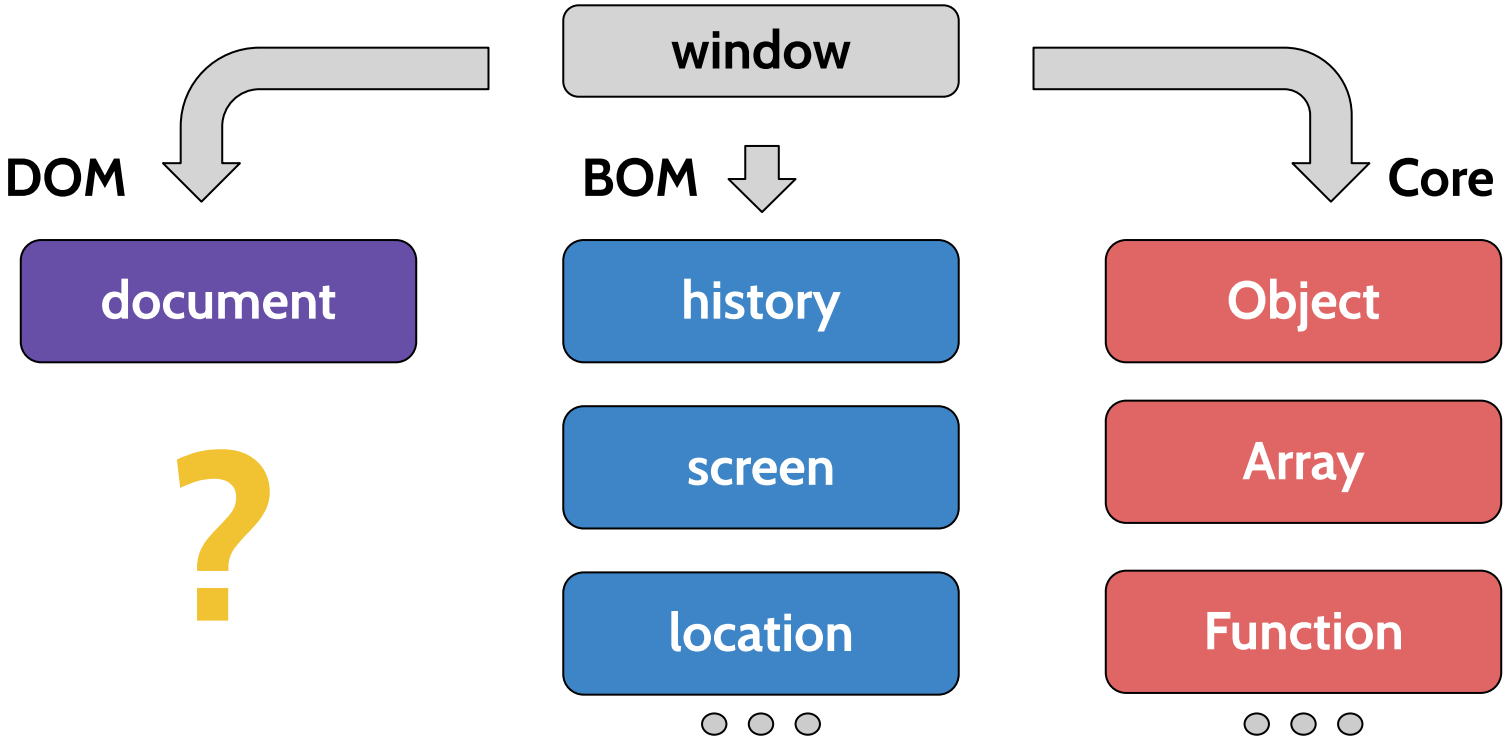


01

Section

- A brief introduction to DOM
- DOM load, DOMContentLoaded
- Document.readyState
- DOM Tree
- DOM Nodes

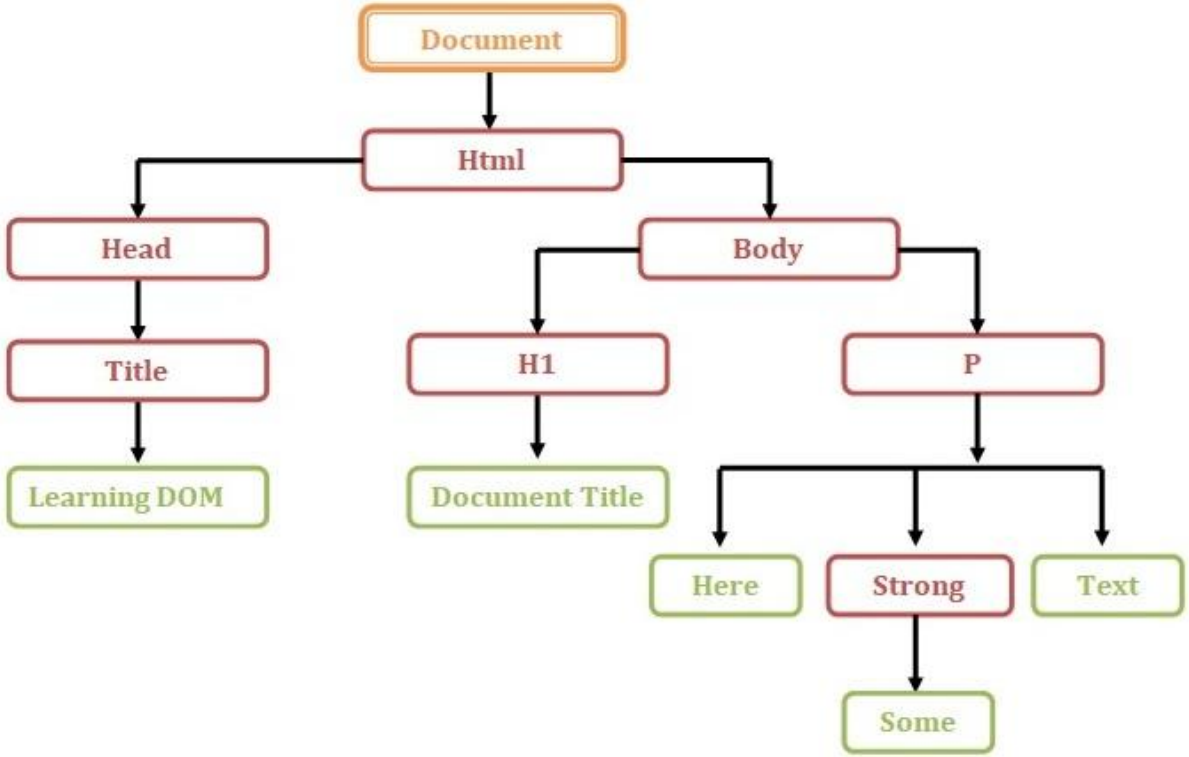
Document Object Model (DOM)



Document Object Model | DOM loading

- **Window: DOMContentLoaded event**
  - The **DOMContentLoaded** event **fires** when the HTML document has been completely **parsed**, and all scripts have downloaded and executed. It doesn't wait for other things like **images, subframes, and async scripts** to finish loading.
  - ```
    window.addEventListener('DOMContentLoaded', function() {
        console.log('DOMContentLoaded');
    });
    ```
- **Window: load event**
  - The **load** event is **fired** when the **whole page has loaded**, including all dependent resources such as **stylesheets and images**.
  - ```
    window.addEventListener('load', function() {
        console.log('Page is fully loaded');
    });
    ```

DOM | Document.readyState

- **Document.readyState**
  - The **Document.readyState** property describes the **loading state** of the document. When the value of this property changes, a **readystatechange** event fires on the **document object**.
  - ```
    window.addEventListener('readystatechange', function() {
        console.log(document.readyState);
    });
    ```
  - The readyState of a document can be one of following:
    - ❖ Loading
      - ➢ The document is **still loading**.
    - ❖ Interactive
      - ➢ The document **has finished loading** and the **document has been parsed** but sub-resources such as **scripts, images, stylesheets** and **frames** are **still loading**.
    - ❖ Complete
      - ➢ The document and all sub-resources have **finished loading**. The state indicates that the load event is about to fire.
    ```

Document Object Model | DOM Tree


```
<!DOCTYPE html>
<html>
<head>
    <title>Learning DOM</title>
</head>
<body>
    <h1>Document Title</h1>
    <p>
        Here <strong>some</strong> text
    </p>
</body>
</html>
```

Document Object Model | DOM Tree

```
<!DOCTYPE html>
<html>
<head>
    <title>Learning DOM</title>
</head>
<body>
    <h1>Document Title</h1>
    <p>
        Here <strong>some</strong> text
    </p>
</body>
</html>
```



- *Parent*
- *Child*
- *Sibling*
- *Ancestry*
- *Descendant*

Document Object Model | DOM Tree

Node Types	
1	ELEMENT_NODE (1)
2	ATTRIBUTE_NODE (2)
3	TEXT_N0DE (3)
4	CDATA_SECTION_NODE (4)
5	ENTITY_REFERENCE_NODE (5)
6	ENTITY_NODE (6)
7	PROCESSING_INSTRUCTION_NODE (7)
8	COMMENT_NODE (8)
9	DOCUMENT_NODE (9)
10	DOCUMENT_TYPE_NODE (10)
11	DOCUMENT_FRAGMENT_NODE (11)
12	NOTATI0N_N0DE (12)

02

Section

- Preselected items
- Documents as Node Trees
- Structure and traversal of a document
- Selecting Document Elements
- Attributes
- ClassList Property

DOM (Preselected Items)

- **Window.document**
  - **window.document** returns a reference to the document contained in the window.

```
console.log(window.document);
```

- **Document.head**
  - The **head** read-only property of the Document **returns the <head> element of the current document**.

```
console.log(document.head);
```

- **Document.body**
  - The **document.body** property represents the **<body>** node of the current document, or **null** if no such element exists.

```
console.log(document.body);
```

DOM (Preselected Items)

- **Document.title**
  - The **document.title** property **gets** or **sets** the current title of the document. When present, it defaults to the value of the **<title>**.

```
console.log(document.title); //Learning JavaScript
document.title = "Learning DOM";
console.log(document.title); //Learning DOM
```

- **Document.URL**
  - The **URL** read-only property of the **Document** returns the document location as a **string**.

```
console.log(document.URL);
```

DOM (Preselected Items)

- **Document.links**
  - The **links** read-only property of the **Document** returns a **collection** of all **<area>** elements and **<a>** elements in a document **with a value for the href attribute**.

```
console.log(document.links); //HTMLCollection [a]
```

- **Document.images**
  - The **images** read-only property of the **Document** returns a **collection** of the images in the **current HTML document**.

```
console.log(document.images); //HTMLCollection [img]
```

DOM (Preselected Items)

- **Document.forms**
  - The **forms** read-only property of the **Document** returns an **HTMLCollection** listing all the **<form>** elements contained in the document.

```
console.log(document.forms); //HTMLCollection [form]
```

```
<form name="login">
    <input name="email" type="email" placeholder="Email"/>
    <input name="password" type="password" placeholder="Password"/>
</form>
```

  - console.log(document.forms.login); //form
  - console.log(document.forms["login"]); //form
  - console.log(document.forms.login.elements); //HTMLFormControlsCollection(2) [input, input, email: input, password: input]

DOM (Documents as Node Trees)

- **Node.parentNode**
  - The read-only **parentNode** property of the **Node** returns the parent of the specified node in the DOM tree.
  - `var body = document.body;`
    `console.log(body.parentNode); //<html>....</html>`

- **Node.childNodes**
  - The read-only **childNodes** property of the **Node** returns a **live NodeList** of child nodes of the given element where the first child node is assigned index **0**. Child nodes include **elements, text and comments**.
  - `var body = document.body;`
    `console.log(body.childNodes); //NodeList(7) [ text, h1, text, p, text, script, text ]`

DOM (Documents as Node Trees)

- **Node.firstChild**
  - The read-only **firstChild** property of the **Node** returns the node's **first child** in the tree, or **null** if the node has no children.
  - ```
    var body = document.body;
    console.log(body.firstChild); //#text
    ```

- **Node.lastChild**
  - The read-only **lastChild** property of the **Node** returns the **last child** of the node. If its parent is an element, then the child is generally an element node, a text node, or a comment node. It returns **null** if there are **no child nodes**.
  - ```
    var body = document.body;
    console.log(body.lastChild); //#text
    console.log(document.lastChild); //<html>....</html>
    ```

DOM (Documents as Node Trees)

- **Node.previousSibling**
  - The read-only **previousSibling** property of the **Node** returns the node immediately preceding the specified one in its parent's childNodes list, or **null** if the specified node is the first in that list.

```
console.log(element.previousSibling);
```

- **Node.nextSibling**
  - The read-only **nextSibling** property of the **Node** returns the node immediately following the specified one in their parent's childNodes, or returns **null** if the specified node is the last child in the parent element.

```
console.log(element.nextSibling);
```

DOM (Documents as Node Trees)

- **Node.nodeName**
  - The read-only **nodeName** property of Node **returns** the name of the current node as a string.
  - `console.log(body.nodeName); //'BODY'`
- **Node.nodeType**
  - The read-only **nodeType** property of a Node is an **integer** that identifies what the node is. It distinguishes different kind of nodes from each other, such as **elements, text and comments**.
  - `console.log(body.nodeType); //1`
- **Node.nodeValue**
  - The **nodeValue** property of the Node interface returns or sets the value of the current node.
  - `console.log(body.value); //text`

DOM (Structure and traversal of a document)

- **Element.children**
  - The read-only **children** property returns a **live HTMLCollection** which contains all of the child elements of the element upon which it was called. ***Element.children includes only element nodes***.
  - ```
    var body = document.body;
    console.log(body.children); //HTMLCollection(3) [ h1, p, script ]
    ```

- **Element.childElementCount**
  - The **Element.childElementCount** read-only property returns the number of child elements of this element.
  - ```
    var body = document.body;
    console.log(body.childElementCount); //3
    ```

DOM (Structure and traversal of a document)

- **Element.firstElementChild**
  - The **Element.firstElementChild** read-only property returns an **element's first child Element**, or **null** if there are no child elements. **Element.firstElementChild** includes only element nodes.
  - ```
    var body = document.body;
    console.log(body.firstElementChild); //h1
    ```

- **Element.lastElementChild**
  - The **Element.lastElementChild** read-only property returns an **element's last child Element**, or **null** if there are no child elements. **Element.lastElementChild** includes only element nodes.
  - ```
    var body = document.body;
    console.log(body.lastElementChild); //script
    ```

DOM (Structure and traversal of a document)

- **Element.nextElementSibling**
  - The **Element.nextElementSibling** read-only property returns **the element immediately following the specified one in its parent's children list**, or **null** if the specified element is the last one in the list.
  - `console.log(element.nextElementSibling);`

- **Element.previousElementSibling**
  - The **Element.previousElementSibling** read-only property returns t**he element immediately prior to the specified one in its parent's children list**, or **null** if the specified element is the first one in the list.
  - ```
    var body = document.body;
    console.log(body.previousElementSibling);
    ```
    ```

DOM (Selecting Document Elements)

- **Element.querySelector()**
  - The **querySelector()** method of the Element returns the **first element** that is a descendant of the element on which it is invoked that matches the specified group of selectors.
  - element.querySelector(selectors);
  - console.log(document.body.querySelector("h1")); //h1

- **Element.querySelectorAll()**
  - The Element method **querySelectorAll()** returns a **static (not live) NodeList** representing a list of elements matching the specified group of selectors which are descendants of the element on which the method was called.
  - element.querySelectorAll(selectors);
  - console.log(document.body.querySelectorAll("p.text")); //[p.text, p.text, p.text]

DOM (Selecting Document Elements)

- **Element.closest()**
  - The **closest()** method of the Element traverses the element and its parents **(heading toward the document root)** until it finds a node that matches the specified CSS selector.
  - `var boldText = document.querySelector("p.text strong");`
    `console.log(boldText.closest("p")); //<p class="text">...</p>`
- **Document.getElementById()**
  - The Document method **getElementById()** returns an Element object representing the element whose **id** property matches the specified string. Since element IDs are required to be unique if specified, they're a useful way to get access to a specific element quickly.
  - `var elem = document.getElementById("btn");`
    `console.log(elem); //<button id="btn">Click</button>`

DOM (Selecting Document Elements)

- **Document.getElementsByTagName()**
  - The **getElementsByTagName** method of Document returns an HTMLCollection of elements with the given **tag name**.
  - `var paragraph = document.getElementsByTagName("p");`
    `console.log(paragraph); //[ p.text, p.text, p.text ]`
- **Document.getElementsByClassName()**
  - The **getElementsByClassName** method of Document returns an array-like object of all child elements which have **all of the given class name(s)**.
  - `var paragraph = document.getElementsByClassName("text");`
    `console.log(paragraph); //[ p.text, p.text, p.text ]`
- **Document.getElementsByName()**
  - The **getElementsByName()** method of the Document object returns a **NodeList Collection** of elements with a given **name attribute** in the document.
  - `console.log(document.getElementsByName("userName")); // input`

DOM (Attributes)

- var image = document.querySelector("img");
    - image.id;
    - image.className;
    - image.src;
- var form = document.querySelector("form");
    - form.action;
    - form.method;
- var label = document.querySelector("label");
    - label.htmlFor
- **Element.attributes**
    - The **Element.attributes** property returns **a live collection of all attribute nodes** registered to the specified node.
    - var element = document.querySelector("p");
      console.log(element.attributes); //{ 0: class, class: class, length: 1 }

DOM (Attributes of Form)

```
var form = document.forms.formName;
```

- **form.name**
  - A string reflecting the value of the form's name HTML attribute, containing the name of the form.
- **form.action**
  - A string reflecting the value of the form's action HTML attribute, containing the URI of a program that processes the information submitted by the form.
- **form method**
  - A string reflecting the value of the form's method HTML attribute, indicating the HTTP method used to submit the form.
- **form length**
  - A long reflecting the number of controls in the form.

DOM (Attributes of Form)

```
var form = document.forms.formName;
```

- **form.elements**
  - A **HTMLFormControlsCollection** holding all form controls belonging to this form element.

```
var input = form.elements["user.name"];
```

- input.form
- input.name
- input.type
- input.value
- input.placeholder
- input.disabled
- input.readOnly
- input.required

DOM (Attributes of Form)

- **Radio Buttons**
  - ```
    var form = document.forms.formName;
    var radio = form.elements["user.gender"];
    console.log(radio.value);
    console.log(radio.item(0).checked);
    console.log(radio.item(1).defaultChecked);
    ```

- **CheckBoxes**
  - ```
    var form = document.forms.formName;
    var checkbox = form.elements["user.interests.coding"];
        console.log(checkbox.value);
        console.log(radio.item(0).checked);
        console.log(radio.item(1).defaultChecked);
    ```

DOM (Attributes of Form)

- **Select and Options**

  - ```
    var form = document.forms.formName;
    var select = form.elements["user.birthYear"];

    console.log(select.length); //count of options
    console.log(select.options); //options list
    console.log(select.options.selectedIndex); //index of selected option
    ```
    ```

DOM (Attributes)

- **Element.getAttribute()**
  - The **getAttribute()** method of the Element returns **the value of a specified attribute on the element**.
  - ```
    var element = document.querySelector("h1");
    console.log(element.getAttribute("id")); //"title"
    ```
- **Element.getAttributeNames()**
  - The **getAttributeNames()** method of the Element returns **the attribute names of the element as an Array of strings**. If the element has no attributes it returns an **empty array**.
  - ```
    var element = document.querySelector("h1");
    console.log(element.getAttributeNames()); //[ "id" ]
    ```
- **Element.setAttribute()**
  - **Sets** the value of an attribute on the specified element. If the attribute **already exists**, the value is **updated**:
  - ```
    var element = document.querySelector("h1");
    element.setAttribute("class", "className");
    ```
    ```

DOM (Attributes)

- **Element.removeAttribute()**
  - The Element method **removeAttribute()** removes the attribute with the specified name from the element.
  - ```
    var element = document.querySelector("h1");
    element.removeAttribute("id");
    ```
- **Element.hasAttribute()**
  - The **Element.hasAttribute()** method returns a **Boolean** value indicating whether the specified element **has the specified attribute or not**.
  - ```
    var element = document.querySelector("h1");
    console.log(element.hasAttribute("id")); //true
    ```
- **Element.hasAttributes()**
  - The **hasAttributes()** method of the Element returns a **boolean** value indicating whether **the current element has any attributes or not**.

DOM (Attribute data)

- **HTMLElement.dataset**
  - The **dataset** read-only property provides **read/write access** to custom data attributes **(data-*)** on elements.

  - ```

        Here some text

    ```

  - ```
    var element = document.querySelector(".title");
    console.log(element.dataset); //DOMStringMap {id: '123456'}
    console.log(element.dataset.id); //123456
    console.log(element.dataset.textInfo); //information
    ```

03

Section

- textContent, innerHTML
- Creating, adding and removing elements
- Working with CSS
- Events

DOM (textContent, innerHTML)

- **Node.textContent**
  - The **textContent** property represents the **text content** of the node and its descendants.
  - Here some text
    var element = document.querySelector(".title");
    console.log(element.textContent); //'Here some text'
- **Element.innerHTML**
  - The Element property **innerHTML gets** or **sets** the HTML markup contained within the element.
  - Here some text
    var element = document.querySelector(".title");
    console.log(element.innerHTML); //'Here some text'

DOM (Creating, adding and removing elements)

- **document.createElement**
  - In an HTML document, the **document.createElement()** method creates the HTML element specified by tagName, or an HTMLUnknownElement if tagName isn't recognized.
  - `var p = document.createElement("p");`
    `console.log(p); //<p></p>`
- **Element.append()**
  - The **Element.append()** method inserts a set of Node objects or string objects after the **last child** of the Element.
  - `var element = document.querySelector("body");`
    `element.append(p);`
- **Element.prepend()**
  - The **Element.prepend()** method inserts a set of Node objects or string objects before the **first child** of the Element.
  - `var element = document.querySelector("body");`
    `element.prepend(p);`

DOM (Creating, adding and removing elements)

- **Element.after()**
  - The **Element.after()** method inserts a set of Node or string objects in the **children list of the Element's parent**, just **after** the Element.
  - after(node1, node2, /* ... ,*/ nodeN)
- **Element.before()**
  - The **Element.before()** method inserts a set of Node or string objects in the **children list of this Element's parent**, just **before** this Element.
  - before(param1, param2, /* ... ,*/ paramN)
- **Node.cloneNode()**
  - The **cloneNode()** method returns a **duplicate of the node** on which this method was called. Its parameter **controls** if the subtree contained in a node **is also cloned or not**.
  - cloneNode(deep)

DOM (Creating, adding and removing elements)

- **Element.remove()**
  - The **Element.remove()** method removes the element from the DOM.
  - ```
    var mainTitle = document.querySelector("h1");
    mainTitle.remove();
    ```
- **Element.replaceWith()**
  - The **Element.replaceWith()** method replaces this Element in the children list of its parent with a set of Node or string objects. String objects are inserted as equivalent **Text nodes**.
  - ```
    var mainTitle = document.querySelector("h1");
    var pElement = document.createElement("p");
    pElement.textContent = "Here some text";

    mainTitle.replaceWith(pElement);
    ```

DOM (Working with CSS)

- **HTMLElement.style**
  - The style read-only property returns the inline style of an element in the form of a **CSSStyleDeclaration** object that contains a **list of all styles properties** for that element with values assigned for the attributes that are defined in the element's inline style attribute.
  - var element = document.querySelector(".text");
    - element.style.display;
    - element.style.margin
    - element.style.position;
    - element.style.left;
    - element.style.color;
    - element.style.fontFamily;
    - element.style.cssText;

DOM (Working with CSS)

- **Window.getComputedStyle()**
  - The **Window.getComputedStyle()** method returns an object containing the values of all CSS properties of an element, after applying active stylesheets and resolving any basic computation those values may contain.
  - ```
    var element = document.querySelector(".text");
    console.log(window.getComputedStyle(element));
    ```
    ```

DOM (Working with CSS | classList)

- **Element.classList**
  - The **Element.classList** is a read-only property that returns a **live collection** of the class attributes of the element. This can then be used **to manipulate the class list**.
  - ```
    var element = document.querySelector("p");
    console.log(element.classList); //[ 'text', 'red', 'strong', value: 'text red strong' ]
    console.log(element.classList.length); //3
    ```

- **add()**
  - The **add()** method adds the given tokens to the list, omitting any that are already present.
  - ```
    var element = document.querySelector("p");
    element.classList.add("className_1"..... "className_N");
    ```

DOM (Working with CSS | classList)

- **remove()**
  - The **remove()** method **removes** the specified tokens from the list.
  - var element = document.querySelector("p");
    element.classList.remove("className_1"..... "className_N");
- **contains()**
  - The **contains()** method returns a boolean value — **true** if the underlying list contains the given token, otherwise **false**.
  - var element = document.querySelector("p");
    element.classList.contains("className_1");
- **replace()**
  - The replace() method **replaces an existing token with a new token**. If the first token doesn't exist, replace() returns false immediately, without adding the new token to the token list.
  - var element = document.querySelector("p");
    element.classList.replace("className_1", "toClassName_1");

DOM (Working with CSS | classList)

- **toggle()**
  - The **toggle()** method **removes** an existing token from the list and returns **false**. If the token doesn't exist it's **added** and the function returns **true**.

  - ```
    var element = document.querySelector("p");
    element.classList.toggle("className_1"); //add or remove
    element.classList.toggle("className_1",  true); //only add
    element.classList.toggle("className_1", false); //only remove
    ```
    ```

DOM (Events)

The Event interface represents an event which takes place in the DOM.

An event can be triggered by the user action e.g. clicking the mouse button or tapping keyboard, or generated by APIs to represent the progress of an asynchronous task. It can also be triggered programmatically, such as by calling the HTMLElement.click() method of an element.

There are many types of events, some of which use other interfaces based on the main Event interface. Event itself contains the properties and methods which are common to all events.

- `<button onclick="alert('Hello, world!')">Click Me</button>`
- `<button onclick="clickFunction()">Click Me</button>`

```
function clickFunction() {
        console.log('Hello, world!');
}
```

DOM (Events)

- ```
  <button id="btn">Click Me</button>
  ```

  ```
  var btn = document.getElementById("btn");
  btn.onclick = function() {
     console.log("Hello, world!");
  };
  btn.onclick = null;
  ```

- ```
  <button id="btn">Click Me</button>
  ```

  ```
  var btn = document.getElementById("btn");
  btn.addEventListener("click", function( event ) {
        console.log("Hello, world!");
  });
  btn.removeEventListener("click", function);
  ```

DOM (Event types)

- **Element: click event**
  - An element receives a click event when a pointing device button (such as a mouse's primary mouse button) is both pressed and released while the pointer is located inside the element.
  - addEventListener('click', function() { /* code */ });
- **Element: mousedown event**
  - The mousedown event is fired at an Element when a pointing device button is pressed while the pointer is inside the element.
  - addEventListener('mousedown', function() { /* code */ });
- **Element: mouseup event**
  - The mouseup event is fired at an Element when a button on a pointing device (such as a mouse or trackpad) is released while the pointer is located inside it.
  - addEventListener('mouseup', function() { /* code */ });

DOM (Event types)

- **Element: mousemove event**
  - The mousemove event is fired at an element when a pointing device (usually a mouse) is moved while the cursor's hotspot is inside it.
  - addEventListener('mousemove', function() { /* code */ });
- **Element: mouseover event**
  - The mouseover event is fired at an Element when a pointing device (such as a mouse or trackpad) is used to move the cursor onto the element or one of its child elements.
  - addEventListener('mouseover', function() { /* code */ });
- **Element: mouseout event**
  - The mouseout event is fired at an Element when a pointing device (usually a mouse) is used to move the cursor so that it is no longer contained within the element or one of its children.
  - addEventListener('mouseout', function() { /* code */ });

DOM (Event types)

- **Element: mouseenter event**
  - The mouseenter event is fired at an Element when a pointing device (usually a mouse) is initially moved so that its hotspot is within the element at which the event was fired.
  - addEventListener('mouseenter', function() { /* code */ });
- **Element: keydown event**
  - The keydown event is fired when a key is pressed.
  - addEventListener('keydown', function() { /* code */ });
- **Element: keyup event**
  - The keyup event is fired when a key is released.
  - addEventListener('keydup', function() { /* code */ });

DOM (Event types)

- **Element: focus event**
  - The focus event fires when an element has received focus. The event does not bubble, but the related focusin event that follows does bubble.
  - addEventListener('focus', function() { /* code */ });
- **Element: focusin event**
  - The focusin event fires when an element has received focus, after the focus event. The two events differ in that focusin bubbles, while focus does not.
  - addEventListener('focusin', function() { /* code */ });
- **Element: blur event**
  - The blur event fires when an element has lost focus. The event does not bubble, but the related focusout event that follows does bubble.
  - addEventListener('blur', function() { /* code */ });
- **Element: focusout event**
  - The focusout event fires when an element has lost focus, after the blur event. The two events differ in that focusout bubbles, while blur does not.
  - addEventListener('focusout', function() { /* code */ });

DOM (Event types)

- **HTMLElement: change event**
  - The change event is fired for **<input>, <select>**, and **<textarea>** elements when the **user modifies the element's value**. Unlike the input event, the change event is not necessarily fired for each alteration to an element's value.
  - addEventListener('change', function() { /* code */ });
- **HTMLElement: input event**
  - The input event fires when the value of an **<input>, <select>,** or **<textarea>** element has been changed.
  - addEventListener('input', function() { /* code */ });
- **HTMLInputElement: select event**
  - The select event fires when **some text has been selected**.
  - addEventListener('select', function() { /* code */ });

DOM (Event types)

- **HTMLFormElement: submit event**
  - The submit event fires when a **<form> is submitted**.
  - addEventListener('submit', function() {

```
/* code */
});
```

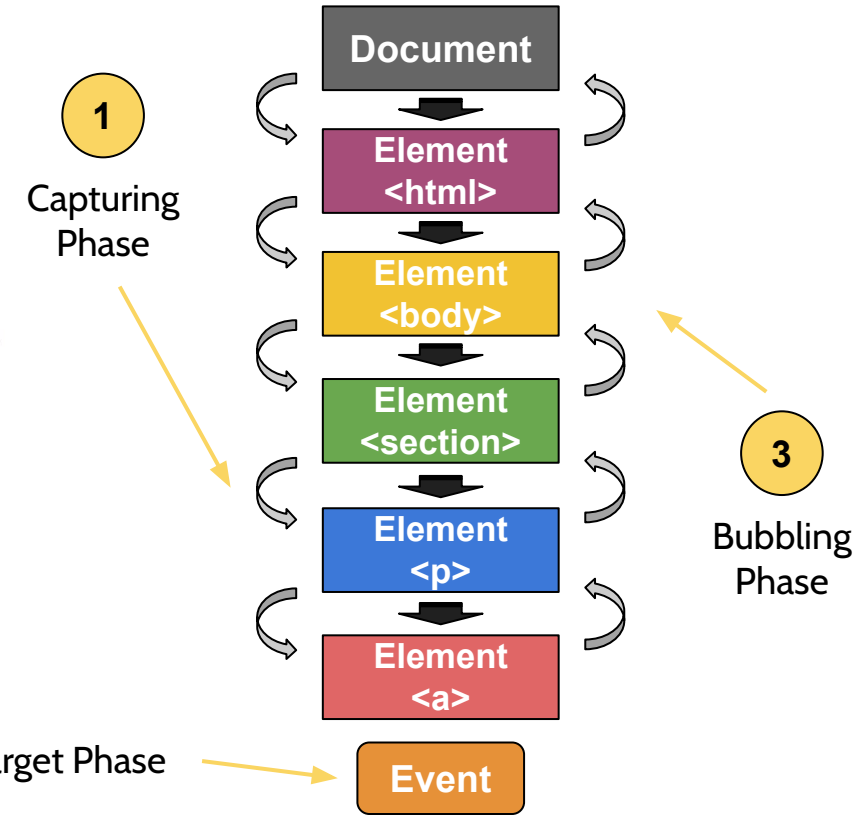
- **HTMLFormElement: reset event**
  - The reset event fires when a **<form> is reset**.
  - addEventListener('reset', function() {

```
/* code */
});
```

DOM (Events | Capturing and Bubbling)

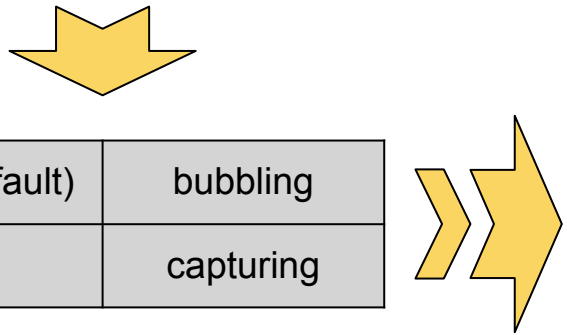
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Learning JS</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <section class="section">
        <p class="paragraph">
            A paragraph with a <a href="#">Link</a>
        </p>
    </section>
    <script src="script.js"></script>
</body>
</html>
```

```js
var link = document.querySelector('a');
link.addEventListener('click', function() {
  console.log('Clicked!');
});
```



DOM (Events | Capturing and Bubbling)

```
var link = document.querySelector('a');
link.addEventListener('click', function() {
   console.log('Clicked!');
}, false);
```



false (default)	bubbling
true	capturing

```
var link = document.querySelector('a');
link.addEventListener('click', function() {
   console.log('Clicked!');
}, { capture: false, //capturing or bubbling
     once: true, //works only the first time
     passive: true}); //preventing will
                      never work
```

DOM (Event Object)

```
var link = document.querySelector('a');
link.addEventListener('click', function(event) {
    console.log(event);
}
```

- **Properties**
  - *Event.type*
    The case-insensitive name identifying the type of the event.
  - *Event.target*
    A reference to the object to which the event was originally dispatched.
  - *Event.currentTarget*
    A reference to the currently registered target for the event. This is the object to which the event is currently slated to be sent. It's possible this has been changed along the way through retargeting.

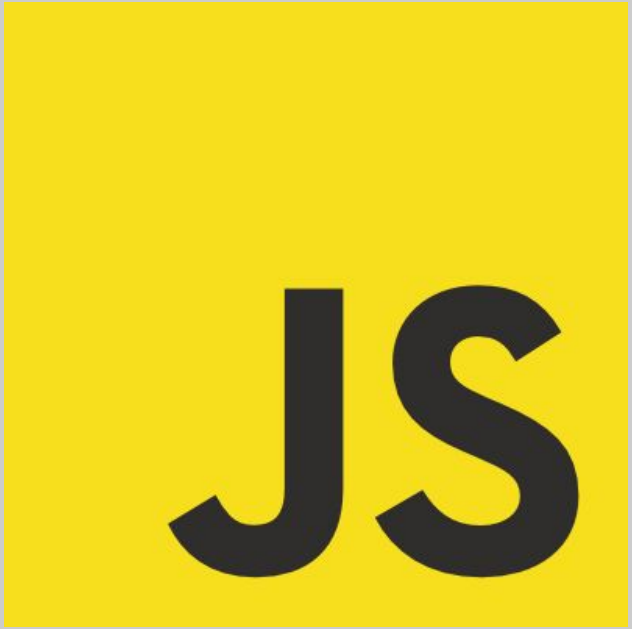
DOM (Event Object)

- ```
  var link = document.querySelector('a');
  link.addEventListener('click', function(event) {
     console.log(event);
  }
  ```
- **Methods**
  - *Event.preventDefault()*

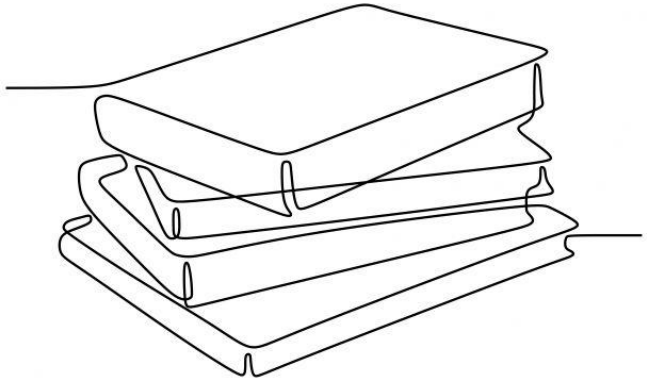
    The preventDefault() method of the Event interface tells the user agent that if the event does not get explicitly handled, its default action should not be taken as it normally would be.
  - *Event.stopPropagation()*

    The stopPropagation() method of the Event interface prevents further propagation of the current event in the capturing and bubbling phases.
  - *Event.stopImmediatePropagation()*

    The stopImmediatePropagation() method of the Event interface prevents other listeners of the same event from being called.
  ```



ECMAScript(ES 6+)
Modern JavaScript



01

Section

- Declaring variables let, const
- Template literals
- Destructuring (Objects, Arrays)
- Sets and Maps

ES 6+ (Let)

- **Let** - The **let** declaration declares a **block-scoped local** variable, optionally initializing it to a value.
  - ```
    let num = 10;
    console.log(num); //10
    ```
  - ```
    let firstName = "John";
    let firstName = "Mike";
    console.log(firstName); //Identifier 'firstName' has already been declared
    ```
  - ```
    {
        let num = 10;
        console.log(num); //10
    }
    console.log(num); //num is not defined
    ```
    ```

ES 6+ (Let)


- ```
  let firstName = "John";
  if(true){
      let firstName = "Mike";
          console.log(firstName); // Mike
  }
  console.log(firstName); // John
  ```

- ```
  if(true){
      // Temporal Dead Zone
      console.log(typeof value); //Cannot access 'value' before initialization
      let value = 10;
  }
  ```

ES 6+ (Let)


- ```
  for (var i = 0; i < 10; i++) {
      //code
  }
  console.log(i); //10
  ```

- ```
  for (let i = 0; i < 10; i++) {
      //code
  }
  console.log(i); //i is not defined
  ```

ES 6+ (Const)

- **Const** - The **const** declaration creates **block-scoped constants**, much like variables declared using the let keyword. The value of a constant **can't be changed** through reassignment, and it **can't be redeclared**. However, if a constant is an **object** or **array** its properties or items **can be updated or removed**.

    - ```
      const num; //Missing initializer in const declaration
      ```

    - ```
      const num = 10;
      console.log(num); //10
      ```

    - ```
      const num = 10;
      num = 20; //Assignment to constant variable
      ```
      ```

ES 6+ (Const)

- ```
  const user = {
      age: 20
  };

  user.age = 50;
  console.log(user); //{ age: 50 }
  ```

- ```
  const arr = [ 10, "string", true ];
  arr[2] = false;

  console.log(arr); //[ 10, 'string', false ]
  ```

ES 6+ (Template literals)

- Template literals are literals delimited with backtick **(`)** characters, allowing for **multi-line strings**, string interpolation with **embedded expressions**, and special constructs called **tagged templates**.
  - ```
    let message = `Hello, world!`;
    console.log(message); \\"Hello, world!"
    console.log(typeof message); \\"string"
    ```
  - ```
    let message = `Hello,
    world!`;
    console.log(message); \\"Hello,
                                  world!"
    ```
  - ```
    let userName = "John" ;
    let message = `Hello, ${userName}!`;
    console.log(message); \\"Hello, John!"
    ```
    ```

ES 6+ (Template literals | Tagged templates)

- Tagged templates
  - A more advanced form of template literals are **tagged templates**. Tags allow you to **parse** template literals with a **function**. The **first argument** of a tag function contains an **array** of string values. The remaining arguments are related to the expressions.
  - ```
    function tagFunction(strings, value) {
      return strings[0] + value.toUpperCase() + strings[1];
    }

    let firstName = 'John';
    let message = tagFunction`Hello, ${firstName}!`;
    console.log(message);
    ```
    ```

ES 6+ (Destructuring)

- **Destructuring assignment**
  - The destructuring **assignment syntax** is a JavaScript expression that makes it possible to **unpack values** from **arrays**, or properties from **objects**, into **distinct variables**.
- **Destructuring Objects**
  - ```
    const user = {
      firstName: "John",
      age: 20,
      isAdmin: false
    };
    const { firstName, age, isAdmin } = user;
    console.log(firstName, age, isAdmin); //"John" 20 false
    ```
    ```

ES 6+ (Destructuring)

- **Destructuring Objects**
  - ```
    const user = {
      firstName: "John",
      age: 20,
      isAdmin: false
    };

    const { firstName: myName,
            age: myAge,
            isAdmin: myIsAdmin } = user;

    console.log(myName, myAge, myIsAdmin); //"John" 20 false
    ```
    ```

ES 6+ (Destructuring)

- **Destructuring Objects (*default value*)**
  - ```
    const user = {
      firstName: "John",
      age: 20,
      isAdmin: false
    };

    const { firstName: myName = " ",
            age: myAge = 0,
            isAdmin: myIsAdmin = false
    } = user;
    console.log(myName, myAge, myIsAdmin); //"John" 20 false
    ```
    ```

ES 6+ (Destructuring)

- **Destructuring Objects**
  - ```
    const user = {
      firstName: "John",
      info: {
        address: "some address",
        phone: "000-00-000"
      }
    };
    const {
      info: { address, phone }
    } = user;
    console.log(address, phone); //"some address" "000-00-000"
    ```
    ```

ES 6+ (Destructuring)

- **Destructuring Objects**
  - ```
    const user = {
      firstName: "John",
      age: 20
    };

    function getUserInfo({ firstName = "Guest", age = 0 }) {
      return `The user name is ${ firstName } and age is ${ age }.`;
    }

    console.log(user);
    console.log({ });
    ```
    ```

ES 6+ (Destructuring)

- **Destructuring Arrays**
  - ```
    const colors = [ "Red", "Green", "Blue" ];
    let [ colorOne, colorTwo, colorThree ] = colors;
    console.log(colorOne, colorTwo, colorThree); //Red Green Blue
    ```
  - ```
    const colors = [ "Red", "Green", "Blue" ];
    let [ colorOne, , colorThree="Aqua" ] = colors;
    console.log(colorOne, colorThree); //Red Blue
    ```
  - ```
    const colors = [ "Red", "Green", "Blue" ];
    let [ colorOne, colorTwo ] = colors;
    [ colorTwo, colorOne ] = [ colorOne, colorTwo ];
    console.log(colorOne, colorTwo); //Green Red
    ```
    ```

ES 6+ (Destructuring)

- **Destructuring Arrays**
  - ```
    function getColors(indexOne, indexTwo) {
      const colors = [ "Red", "Green", "Blue" ];
      return [ colors[indexOne], colors[indexTwo] ];
    }

    let [ colorOne, colorTwo ] = getColors(0, 1);
    console.log(colorOne, colorTwo);
    ```
  - ```
    const colors = [ "Red", "Green", "Blue", [ true, 10] ];

    let [ red, , , [ bool, num ] ] = colors;
    console.log(red, bool, num);
    ```

ES 6+ (Sets)

- The **Set object** lets you store **unique values** of any type, whether primitive values or object references.
  - ```
    const names = new Set([ "John", "Mike", "Kevin", "John"]);
    console.log(names); // { 'John', 'Mike', 'Kevin' }
    ```
  - ```
    const set = new Set( );
    console.log(set); //{ }
    ```
  - ```
    const set = new Set("John");
    console.log(set); //{ 'J', 'o', 'h', 'n' }
    ```
- **Set.size**
  - The size accessor property returns the number of (unique) elements in a Set object.
  - ```
    const names = new Set([ "John", "Mike", "Kevin", "John"]);
    console.log(names.size); //3
    ```

ES 6+ (Sets)

- **Set.has( )**
  - The **has()** method returns a **boolean** indicating whether an element with the specified value **exists** in a Set object **or not**.
  - ```
    const names = new Set([ "John", "Mike", "Kevin" ]);
    console.log(names.has("Mike")); //true
    ```

- **Set.add( )**
  - The **add()** method inserts a new element with a specified value in to a Set object, if there isn't an element with the same value already in the Set.
  - ```
    const names = new Set([ "John", "Mike", "Kevin" ]);
    names.add("Tom");
    console.log(names); //{'John', 'Mike', 'Kevin', 'Tom'}
    ```

ES 6+ (Sets)

- **Set.delete( )**
  - The **delete()** method removes a specified value from a Set object, if it is in the set.
  - ```
    const names = new Set([ "John", "Mike", "Kevin" ]);
    names.delete("Mike");
    console.log(names); //{'John', 'Kevin'}
    ```

- **Set.clear( )**
  - The **clear()** method removes all elements from a Set object.
  - ```
    const names = new Set([ "John", "Mike", "Kevin" ]);
    names.clear( );
    console.log(names); //{ }
    ```

ES 6+ (Maps)

- The **Map object** holds key-value pairs and remembers the original insertion order of the keys. Any value (both objects and primitive values) may be used as either a key or a value.
  - ```
    const user = new Map();
    console.log(user); //{ }
    ```

- **Map.set()**
  - The **set()** method adds or updates an entry in a Map object with a specified key and a value.
  - ```
    const user = new Map();
    user.set("name", "John Doe");
    console.log(user); //{ 'name' => 'John Doe' }
    ```

ES 6+ (Maps)

- **Map.get()**
  - The get() method returns a specified element from a Map object. If the value that is associated to the provided key is an object, then you will get a reference to that object and any change made to that object will effectively modify it inside the Map object.

  - ```
    const user = new Map();
    user.set("name", "John Doe");

    const n = user.get("name");
    console.log(n);  //John Doe
    ```
    ```

ES 6+ (Maps)

- **Map.size**
  - The size accessor property returns the number of elements in a Map object.
  - ```
    const user = new Map();
    user.set("name", "John Doe");
    console.log(user.size); // 1
    ```

- **Map.has()**
  - The has() method returns a boolean indicating whether an element with the specified key exists or not.
  - ```
    const user = new Map();
    user.set("name", "John Doe");
    console.log(user.has("name")); //true
    ```

ES 6+ (Maps)

- **Map.delete()**
  - The delete() method removes the specified element from a Map object by key.
  - ```
    const user = new Map();
    user.set("name", "John Doe");
    user.delete("name");
    console.log(user); // { }
    ```

- **Map.clear()**
  - The clear() method removes all elements from a Map object.
  - ```
    const user = new Map();
    user.set("name", "John Doe");
    user.clear();
    console.log(user); // { }
    ```

02

Section

- The Spread Operator (...)
- Rest Pattern and Parameters
- Loop for of
- Nullish coalescing operator
- Logical Assignment Operators
- Optional chaining (?.)

ES 6+ (The spread Operator)

- Spread syntax can be used when all elements from an object or array need to be included in a new array or object, or should be applied one-by-one in a function call's arguments list.
  - ```
    const arr = [ 10, "John", true ];
    const newArr = [ 1, 2, ...arr ];
    const copyArray = [ ...arr ];
    console.log(newArr);
    console.log(copyArray);
    console.log(...arr);
    ```
  - ```
    const obj = { prop: 10 };
    const objCopy = { ...obj };
    console.log(obj);
    console.log(objCopy);
    ```

ES 6+ (Rest Pattern and Parameters)


- ```
  const arr = [ 10, "John", true, 20 ];
  let [ num, text, ...others ] = arr;

  console.log(num, text, others); //10 'John'  [true, 20]
  ```

- ```
  const obj = {
     id: 0,
     text: "Hello",
     bool: false
  };
  let { id, ...others } = obj;
  console.log(id, others); //0 {text: 'Hello', bool: false}
  ```

ES 6+ (Rest Pattern and Parameters)


- ```
  function sum(...numbers) {
      let result = 0;
      for (let i in numbers) {
          result += numbers[i];
      }
      return result;
  }

  console.log(sum(1 ,2)); //3
  console.log(sum(4 ,2, 9)); //15
  console.log(sum(7 ,1, 3, 5, 8, 0)); //24
  ```
  ```

ES 6+ (Rest Pattern and Parameters)


- ```
  function some(param, ...args) {
      console.log(args); // [text, true]
      console.log(arguments); // [3, text, true ...]
  }

  some( 3, "text", true );
  ```
  ```

ES 6+ (loop for of)

- const browsers = ["Chrome", "Mozilla", "Edge", "Opera", "Safari"];

  - *EcmaScript 5*

```
for (var i in browsers) {
        console.log(browsers[ i ]);
}
```

  - *EcmaScript 6+*

```
for (let i of browsers) {
        console.log( i );
}
```

ES 6+ (Nullish coalescing operator ??)

- **The nullish coalescing (??)** operator is a logical operator that returns its right-hand side operand when its left-hand side operand is **null** or **undefined**, and otherwise returns its left-hand side operand.

  - ```
    const foo = null ?? 'default string';
    console.log(foo); //default string
    ```
  - ```
    const foo = undefined ?? 'default string';
    console.log(foo); //default string
    ```
  - ```
    const num = 0 ?? 10;
    console.log(num); //0
    ```
  - ```
    const str =  " " ?? "default string";
    console.log(num); //" "
    ```

ES 6+ (Logical Assignment Operators)

- Logical OR assignment (||=)
  - The logical OR assignment **(x ||= y)** operator only assigns if x is **falsy**.
    - *EcmaScript 5*
      - ```
        var num = 10;
        num = num || 0;
        ```
    - *EcmaScript 6*
      - ```
        let num = 10;
        num ||= 0;
        ```

- Logical AND assignment (&&=)
  - The logical AND assignment **(x &&= y)** operator only assigns if x is **truthy**.
    - *EcmaScript 5*
      - ```
        var firstName = 'John';
        firstName = firstName && 'Guest';
        ```
    - *EcmaScript 6*
      - ```
        let firstName = 'John';
        firstName &&= 'Guest';
        ```

ES 6+ (Logical Assignment Operators)

- Nullish coalescing assignment (??=)
  - The nullish coalescing assignment **(x ??= y)** operator only assigns if **x is nullish (null or undefined)**.
  - ```
    const user = {
       id: null (or undefined)
    };

    user.id = user.id ?? 0;
            OR
    user.id ??= 0;
    console.log(user); //{ id: 0 }
    ```
    ```

ES 6+ (Optional chaining (?.))

- Optional chaining (?.)
  - The **optional chaining (?.)** operator accesses an object's **property** or calls a **function**. If the object is undefined or null, it returns **undefined** instead of **throwing an error**.
  - ```
    const user = {
      firstName: 'John',
      dog: {
        name: 'Baron',
        breed: 'pug'
      }
    }
    const dogName = user.cat?.name; //undefined
    const method = user.nonExistingMethod?.(); //undefined
    ```
    ```

03

Section

- Functions (Default parameters)
- Arrow function expressions
- Enhanced object literals
- Object.keys()
- Object.values()
- Object.entries()
- Object.fromEntries()
- Object.freeze()

ES 6+ (Functions)

- *EcmaScript 5*
  - ```
    function some(name) {
        name = name || "Guest";

        console.log(name); //Guest
    }
    some();
    ```

- *EcmaScript 6*
  - ```
    function some(name = "Guest") {
        console.log(name); //Guest
    }
    some();
    ```

ES 6+ (Functions)

- ```
  function getValue(){
      return 0;
  }
  ```

- ```
  function sum(numOne, numTwo = getValue() ){
      console.log(numOne + numTwo);
  }
  sum( 3 ); //3
  sum( 3, 10 ); //13
  ```

ES 6+ (Functions)

- ```
  function sum(numOne, numTwo = numOne){
      console.log(numOne + numTwo);
  }
  sum( 3 ); //6
  sum( 3, 10 ); //13
  ```

- ```
  function getValue(value){
      return value + 1;
  }
  function sum(numOne, numTwo = getValue(numOne) ){
      console.log(numOne + numTwo);
  }
  sum( 3 ); //7    sum( 3, 10 ); //13
  ```

ES 6+ (Arrow function expressions)

- An **arrow function expression** is a compact alternative to a traditional function expression, with some semantic differences and deliberate limitations in usage.
  - ```
    const sayHello = ( ) => {
        console.log("Hello, World!");
    };
    sayHello( );
    ```
  - ```
    const sayHello = ( ) => {
        return `Hello, ${value}!`;
    };
    sayHello("John");
    ```
  - ```
    const sayHello = name => Hello, ${value}!`;
    ```
    ```

ES 6+ (Enhanced object literals)

- ***EcmaScript 5***

  - ```
    var firstName = 'John';
    var user = {
        firstName: firstName,
        greeting: function() {
            return 'Hello ' + this.firstName;
        }
    };
    console.log(user);
    ```
    ```

ES 6+ (Enhanced object literals)

- ***EcmaScript 6***

  - ```
    const firstName = 'John';
    const user = {
        firstName,
        greeting() {
            return `Hello ' + ${ this.firstName }`;
        }
    };
    console.log(user);
    ```
    ```

ES 6+ (Object.keys())

- **Object.keys()**
  - The **Object.keys()** method **returns an array** of a given object's own enumerable string-keyed property **names**.
  - ```
    const user = {
      firstName: 'John',
      age: 20,
      isAdmin: false
    };

    const keys = Object.keys(user); //[ 'firstName', 'age', 'isAdmin' ]
    console.log(keys);
    ```
    ```

ES 6+ (Object.values())

- **Object.values()**
  - The **Object.values()** method **returns an array** of a given object's own enumerable string-keyed property **values**.
  - ```
    const user = {
      firstName: 'John',
      age: 20,
      isAdmin: false
    };

    const values = Object.values(user);
    console.log(values); //[ 'John', 20, false ]
    ```
    ```

ES 6+ (Object.entries())

- **Object.entries()**
  - The **Object.entries()** method **returns an array** of a given object's own enumerable **string-keyed property key-value pairs**.
  - ```
    const user = {
      firstName: 'John',
      age: 20,
      isAdmin: false
    };

    const entries = Object.entries(user);
    console.log(entries); //[Array(2), Array(2), Array(2)]
    ```
    ```

ES 6+ (Object.fromEntries())

- **Object.fromEntries()**
  - The **Object.fromEntries()** method transforms a **list** of key-value pairs into an **object**.
  - ```
    const arr = [
        [0, "a"],
        [1, "b"],
        [2, "c"],
    ];

    const result = Object.fromEntries(arr);
    console.log(result);
    ```
    ```

ES 6+ (Object.freeze())

- **Object.freeze()**
  - The **Object.freeze()** method **freezes** an object. Freezing an object **prevents** extensions and makes existing properties **non-writable** and **non-configurable**. A frozen object can no longer be changed: new properties **cannot be added**, **existing properties cannot be removed, their enumerability, configurability, writability, or value cannot be changed, and the object's prototype cannot be re-assigned**. **freeze()** returns the same object that was passed in.
  - ```
    const obj = {
            prop: 42
    };
    Object.freeze(obj);
    obj.prop = 33; //Throws an error in strict mode
    console.log(obj.prop); //{prop: 42}
    ```
    ```

JS

04

Section

- Classes
- Setters and Getters
- Static Properties and Methods
- Inheritance
- Private Class Fields and Methods

ES 6+ (Classes)

- Classes are a **template** for **creating objects**. They **encapsulate data** with code to work on that data. Classes in JS are built on prototypes but also have some syntax and semantics that are not shared with ES5 class-like semantics.
  - ```
    class Person {
      constructor(firstName, year, isAdmin) {
        this.firstName = firstName;
        this.birthYear = year;
        this.isAdmin = isAdmin
      }
      greeting() {
        return `Hello from ${this.firstName} !`;
      }
    }
    const user = new Person('John', 2000, false);
    console.log(user); //{ firstName: 'John', birthYear: 2000, isAdmin: false }
    ```
    ```

ES 6+ (Setters and Getters)

- **getter**
  - Sometimes it is desirable to allow access to a property that **returns a dynamically computed value**, or you may want to reflect the status of an internal variable without requiring the use of explicit method calls.
  - ```
    class Person {
      constructor(firstName, year) {
        this.firstName = firstName;
        this.birthYear = year;
      }
      get age() {
        return 2022 - this.birthYear;
      }
    }
    const user = new Person('John', 2000);
    console.log(user.age); //22
    ```
    ```

ES 6+ (Setters and Getters)

- **setter**
  - A setter can be used to execute a function whenever a specified property is **attempted to be changed**. Setters are most often used in **conjunction** with **getters** to **create** a type of pseudo-property.
  - ```
    class Person {
      constructor(fullName) {
        this.fullName = fullName;
      }
      get fullName() { return this._fullName; }
      set fullName(name) {
            name.includes(' ') ? this._fullName = name : ' ';
      }
    }
    const user = new Person('John Doe');
    console.log(user.fullName); //John Doe
    ```
    ```

ES 6+ (Static Methods)

- **static**
  - Neither static methods nor static properties can be called on instances of the class. Instead, they're called on the class itself.
  - ```
    class Person {
      constructor(firstName, year) {
        this.firstName = firstName;
        this.birthYear = year;
      }
      static isAdmin = false;
      static calculateAge() {
        return 2022 - this.birthYear;
      }
    }
    const user = new Person('John', 2000);
    console.log(user); //{firstName: 'John', birthYear: 2000}
    ```
    ```

ES 6+ (Inheritance)


```
class Person {
    constructor(firstName, year) {
        this.firstName = firstName;
        this.birthYear = year;
    }
}
class Student extends Person {
    constructor(firstName, year, course){
        super(firstName, year);
        this.course = course;
    }
}
const student = new Student('John', 2000, 'JavaScript');
console.log(student); //{firstName: 'John', birthYear: 2000, course: 'JavaScript'}
```

ES 6+ (Private Class Fields and Methods)


- ```
  class Person {
      #isAdmin = true;
      constructor(firstName, lastName, birthYear) {
          this.firstName = firstName;
          this.lastName = lastName;
          this.birthYear = birthYear;
      }
      #greet() {
          return `Hello from ${ this.fullName }.`;
      }
  }
  const person = new Person("John", "Doe", 2000);
  console.log(person.#isAdmin); //Error          console.log(person.#greet()); //Error
  ```
  ```

05

Section

- Asynchronous JavaScript:
  - Promise
  - async/await
  - Working With HTTP Requests (XMLHttpRequest/Fetch API)

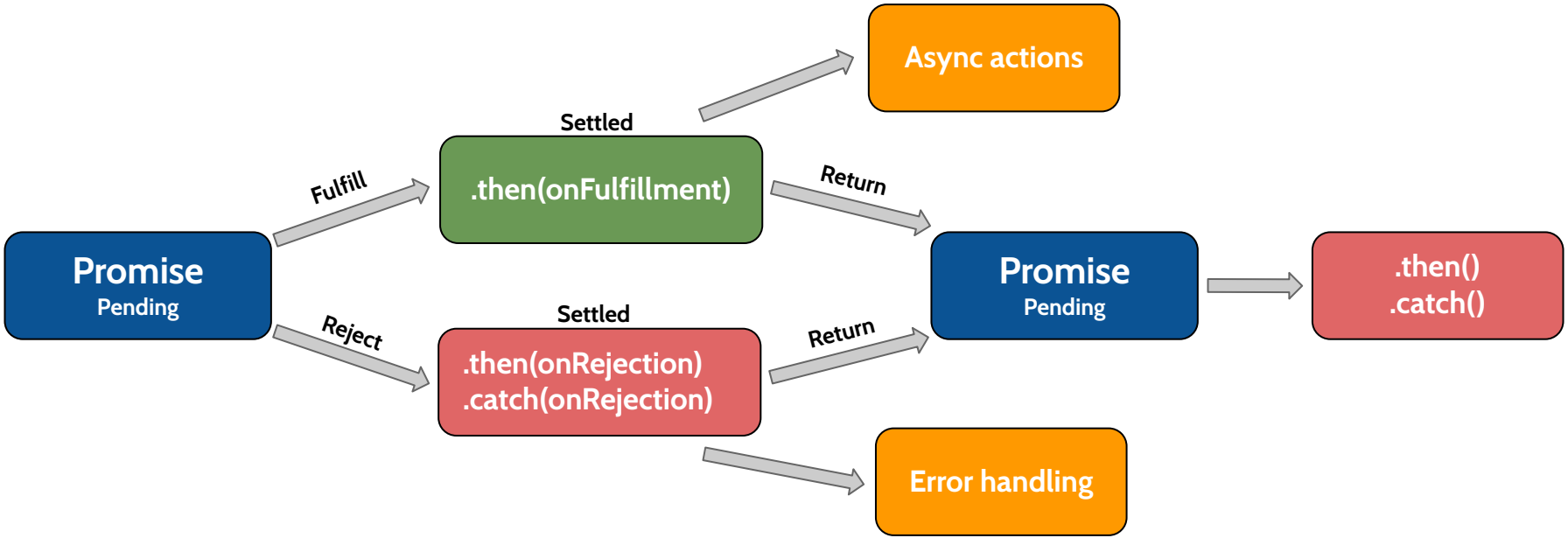
Asynchronous JavaScript | Promise

- The **Promise object** represents the eventual **completion (or failure)** of an **asynchronous** operation and its **resulting value**. It allows you to associate handlers with an asynchronous action's eventual **success** value or **failure** reason. This lets asynchronous methods return values **like synchronous methods**: instead of immediately returning the final value, the asynchronous method returns a **promise** to supply the value at some point **in the future**.

- A Promise is in one of these **states**:
  - ***pending***: initial state, neither fulfilled nor rejected.
  - ***fulfilled***: meaning that the **operation was completed successfully**.
  - ***rejected***: meaning that the **operation failed**.

Asynchronous JavaScript | Promise

- The **eventual state** of a pending promise can either be **fulfilled** with a value or **rejected** with a reason (error). When either of these options occur, the associated handlers queued up by a promise's **then** method are called. A promise is said to be **settled** if it is either fulfilled or rejected, but not pending.



Asynchronous JavaScript | Promise

```
function getAccess(hasAccess) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if(hasAccess) {
        return resolve({ id: 1, firstName: 'John' });
      }
      return reject({ status: false, message: 'You have not permissions' });
    }, 3000);
  });
}
getAccess(false) //or true
  .then((data) => { console.log(data); //{ id: 1, firstName: 'John' } })
  .catch((err) => { console.log(err); //{ status: false, message: 'You have not permissions' } });
  .finally(() => { console.log('Done!'); //Done });
```

Asynchronous JavaScript | Chained Promises

The methods **then()**, **catch()**, and **finally()** are used to associate **further action** with a promise that becomes settled. As these methods **return promises**, they can be **chained**.

- The **then()** method takes up to **two** arguments; the first argument is a **callback function** for the **fulfilled** case of the promise, and the second argument is a **callback function** for the **rejected** case.
- The **catch()** method of Promise instances schedules a function to be called when the promise is r**ejected**.
- The **finally()** method of Promise instances schedules a function to be called promise is **settled (either fulfilled or rejected)**.

Asynchronous JavaScript | Promise static methods

- The **Promise class** offers **four** static methods to facilitate async task concurrency:
  - ***Promise.all( )***

    Fulfills when all of the promises fulfill, rejects when any of the promises rejects.
  - ***Promise.allSettled()***

    Fulfills when all promises settle.
  - ***Promise.any()***

    Fulfills when any of the promises fulfills, rejects when all of the promises reject.
  - ***Promise.race()***

    Settles when any of the promises settles. In other words, fulfills when any of the promises fulfills, rejects when any of the promises rejects.

Asynchronous JavaScript | async/await

The **async** and **await** keywords enable **asynchronous, promise-based behavior** to be written in a **cleaner style**, avoiding the need to explicitly configure **promise chains**.

- The **async function** declaration declares an async function where the await keyword is permitted within the function body. Async functions may also be defined as **expressions.**

```
function resolver() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { resolve('resolved!'); }, 5000);
    });
}
async function asyncCall() {
    const result = await resolver(); //'resolved!'
}
asyncCall();
```

Asynchronous JavaScript | async/await

Async functions always **return a promise**. If the return value of an async function is not explicitly a promise, it will be implicitly **wrapped** in a promise.

- ```
  async function foo() {
    return 1;
  }

  console.log(foo());
  //Promise {<fulfilled>: 1}
  ```

=

- ```
  function foo() {
    return Promise.resolve(1);
  }

  console.log(foo());
  //Promise {<fulfilled>: 1}
  ```

Note: Even though the return value of an async function behaves as if it's wrapped in a Promise.resolve, they are not equivalent.

Working with HTTP Requests | XMLHttpRequest

- **XMLHttpRequest**
  - XMLHttpRequest (XHR) objects are used to **interact with servers**. You can retrieve data from a URL without having to do a **full page refresh**. This enables a Web page to update just **part of a page** without disrupting what the user is doing.
  - **XMLHttpRequest()** constructor initializes an XMLHttpRequest. It must be called before any other method calls.

```
const xhr = new XMLHttpRequest();
console.log(xhr); //XMLHttpRequest { ... }
```

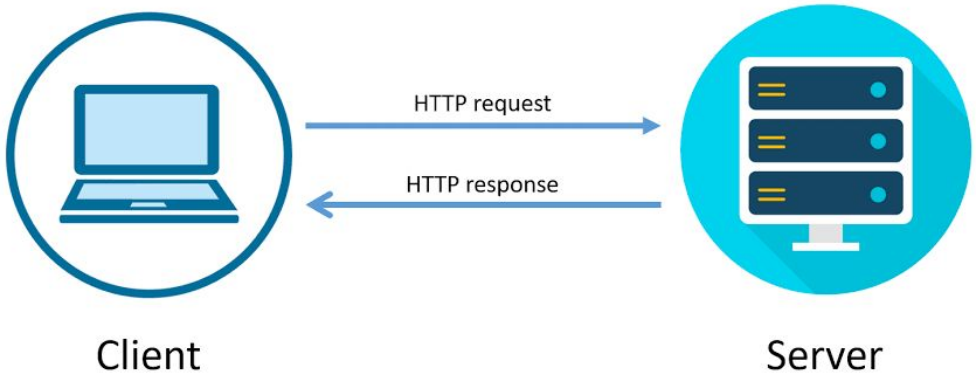
- **Instance methods**
  - **xhr.open()** - Initializes a request.
  - **xhr.send()** - Sends the request. If the request is asynchronous (which is the default), this method returns as soon as the request is sent.
  - **xhr.setRequestHeader()** - Sets the value of an **HTTP request header**. You must call setRequestHeader() **after** open(), but **before** send().
  - **xhr.abort()** - Aborts the request if it has already been sent.

Working with HTTP Requests | XMLHttpRequest

- Here is an example of a simple XMLHttpRequest
  - ```
    const xhr = new XMLHttpRequest();

    xhr.responseType = 'json';
    xhr.open('GET', 'url');
    xhr.send();

    xhr.onload = () => {
       console.log(xhr.status);
       console.log(xhr.readyState);
       console.log(xhr.response);
    };
    ```


    ```

Working with HTTP Requests | XMLHttpRequest

- **Instance properties**

  ```
  const xhr = new XMLHttpRequest();
  ```

  - **xhr.readyState** - Returns a number representing the state of the request.
  - **xhr.status** - Returns the HTTP response status code of the request.
  - **xhr.response** - Returns an **ArrayBuffer**, a **Blob**, a **Document**, a **JavaScript object**, or a **string**, depending on the value of XMLHttpRequest.responseType, that contains the response entity body.
  - **xhr.responseURL** - Returns the **serialized URL** of the response or the empty string if the URL is null.
  - **xhr.responseType** - Specifies the type of the response.

Working with HTTP Requests | XMLHttpRequest

- **Instance events**

  ```
  const xhr = new XMLHttpRequest();
  ```

  - **load** - Fired when an XMLHttpRequest transaction completes successfully. Also available via the onload event handler property.
  - **error** - Fired when the request encountered an error. Also available via the onerror event handler property.
  - **readystatechange** - Fired whenever the readyState property changes. Also available via the onreadystatechange event handler property.
  - **timeout** - Fired when progress is terminated due to preset time expiring. Also available via the ontimeout event handler property.
  - **abort** - Fired when a request has been aborted, for example because the program called XMLHttpRequest.abort(). Also available via the onabort event handler property.

Working with HTTP Requests | readyState

- **XMLHttpRequest: readyState property**
  - The XMLHttpRequest.readyState property returns the state an XMLHttpRequest client is in. An XHR client exists in one of the following states:

Value	State	Description
`0`	`UNSENT`	Client has been created. `open()` not called yet.
`1`	`OPENED`	`open()` has been called.
`2`	`HEADERS_RECEIVED`	`send()` has been called, and headers and status are available.
`3`	`LOADING`	Downloading; `responseText` holds partial data.
`4`	`DONE`	The operation is complete.

Working with HTTP Requests | JSON

- The **JSON** namespace object contains static methods for parsing values from and converting values to **JavaScript Object Notation** (JSON).
- Unlike most global objects, JSON is **not** a constructor. You **cannot** use it with the **new** operator or **invoke** the JSON object as a **function**. All properties and methods of JSON are **static** (just like the Math object).
  - **Example JSON**

```
{
   "browsers": {
      "firefox": {
         "name": "Firefox",
         "pref_url": "about:config",
      }
   }
}
```

Working with HTTP Requests | JSON

JSON is a syntax for serializing objects, arrays, numbers, strings, booleans, and null. It is based upon JavaScript syntax, but is distinct from JavaScript: most of JavaScript is not JSON.
For example:

- **Objects and Arrays** - Property names must be double-quoted strings; trailing commas are forbidden.
- **Numbers** - Leading zeros are prohibited. A decimal point must be followed by at least one digit. NaN and Infinity are unsupported.

Other differences include allowing only double-quoted strings and no support for undefined or comments.

- **Static methods**
  - **JSON.parse()** - Parse a piece of string text as JSON, optionally transforming the produced value and its properties, and return the value.
  - **JSON.stringify()** - Return a JSON string corresponding to the specified value, optionally including only certain properties or replacing property values in a user-defined manner.

Working with HTTP Requests | Fetch API

The **Fetch API** provides a **JavaScript interface** for accessing and manipulating parts of the **protocol**, such as **requests** and **responses**. It also provides a **global fetch() method** that provides an **easy**, **logical** way to fetch resources **asynchronously** across the network. Unlike XMLHttpRequest that is a callback-based API, Fetch is **promise-based**.

- **fetch()** - The **global fetch()** method starts the process of fetching a resource from the network, returning a promise which is fulfilled once the response is available.
    - ```
      async function fetchData() {
        const response = await fetch('url', {
          method: 'POST',
          body: JSON.stringify({ })
        });
        return response.json();
      }
      ```
      ```

Working with HTTP Requests | Fetch API

- **Parameters**

  fetch(resource, options)

  - **resource** - This defines the resource that you wish to fetch.
  - **options** - An object containing any custom settings that you want to apply to the request. The possible options are:
    - **method** - The request method, e.g., GET, POST....
    - **headers** - Any headers you want to add to your request.
    - **body** - Any body that you want to add to your request.

06

Section

- Modular JavaScript
  - What are JS modules
  - Importing And Exporting
  - Renaming Exports
  - Default export
  - Dynamic Imports
  - Module Scope And globalThis

Modular JavaScript | What are JS modules?

JS modules (also known as **“ES modules”** or **“ECMAScript modules”**) are a major new feature, or rather a collection of new features. JavaScript modules allow you to **break up your code** into separate files. This makes it easier **to maintain** a code-base.

What is a module?

- A module is just a file. **One script is one module**. As simple as that.
- Modules can **load each other** and use special directives **export** and **import** to interchange functionality, call functions of one module from another one.

You can tell browsers to treat a **<script>** element as a module by setting the **type attribute** to **module**.

- `<script type="module" src="app.js"></script>`

NOTE: **Modules work only via HTTP(s), not locally.**

- If you try to open a web-page locally, via **file:// protocol**, you’ll find that import/export directives **don’t work**. Use a **local web-server**, such as static-server or use the **“live server”** capability of your editor, such as **VS Code Live Server Extension** to test modules.

Modular JavaScript | Importing and Exporting

- **export** keyword labels variables and functions that should be accessible **from outside the current module**.
  - ```
    //user.js
    const  age = 25;
    export const firstName = 'John';
    export function getAge() {
      return age;
    }
    ```
- **import** allows the import of functionality **from other modules**.
  - ```
    //main.js
    import { firstName, getAge } from './modules/user.js';
    console.log('FirstName:', firstName); //'John'
    getAge(); //25
    ```

Modular JavaScript | Renaming Exports

- ES modules also allow you to **rename** a function or variable while exporting it, again use the as syntax.
  - ```
    //user.js
    export const  age = 25;
    export const firstName = 'John';
    export function getUserAge() {
      return age;
    }
    ```
  - ```
    //main.js
    import { firstName as  name,  getUserAge as getAge } from './modules/user.js';
    console.log('FirstName:', name); //'John'
    getAge(); //25
    ```

Modular JavaScript | Default export

The functionality we've exported so far has been comprised of **named exports** — each item (be it a function, const, etc.) has been referred to by its name upon export, and **that name** has been used to **refer** to it on import as well.

- There is also a type of export called the **default export** — this is designed to make it easy to have a default function provided by a module.
    - ```
      //user.js
      const  age = 25;
      const firstName = 'John';
      function getAge() {
         return age;
      }
      export default { firstName, getAge };
      ```

- ***NOTE:*** There is only **one default export** allowed **per module**.
      ```

Modular JavaScript | Dynamic Imports

- A recent addition to JavaScript modules functionality is **dynamic module loading**. This allows you to dynamically load modules **only when they are needed**, rather than having to load everything up front.
  - ```
    const btn = document.getElementById('btn');
    btn.addEventListener('click', (ev) => {
      ev.preventDefault();

      import('./modules/user.js').then(module => {
        console.log(module);
      });
    });
    ```
- Call **import()** as a **function**, passing it the **path** to the module as a parameter. It returns a **Promise**, which fulfills with a module object giving you access to that object's exports.
    ```

Modular JavaScript | Module Scope And globalThis

- Each module has its own **top-level scope**. In other words, top-level **variables** and **functions** from a module are **not seen in other scripts**.
  - //module.js
    const firstName = 'John';
  - //app.js
    console.log(firstName); //ReferenceError: firstName is not defined
- The **globalThis** variable is a global object that is available in **every environment** and is useful if you want to read or create global variables within modules(In a module, top-level **this** is **undefined**).
  - //module.js
    console.log(this); //undefined
    console.log(globalThis); //window
- Modules always **use strict**.
  - //app.js
    firstName = 'John'; //ReferenceError: firstName is not defined