JavaScript

WEB DEVELOPMENT- 102044505

PROFESSOR: Dr. KRUNAL PATEL

ADIT ENGINEERING COLLEGE

NEW VALLABH VIDYANAGAR

Introduction

JavaScript or **JS** in short is prototype (*Object*) based, interpreted (*JIT or runtime Complied in modern web browsers*), **scripting language** of web developed by **Netscape** in 1995.

What is JavaScript?

JavaScript is a Scripting language designed for Web pages.

Why Use JavaScript?

JavaScript enhances Web pages with dynamic and interactive features.	JavaScript runs in client software.	Scripts are programs just like any other programs.

Introduction

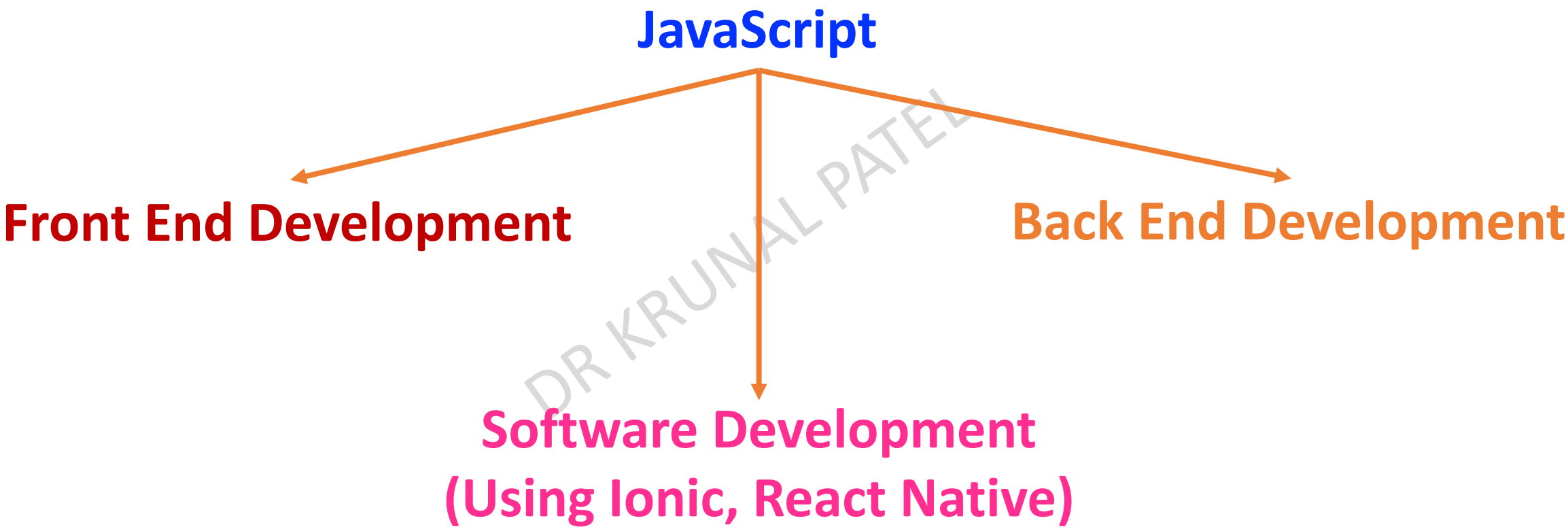
JavaScript is used to build **interactive websites** with **dynamic** features and to **validate form data**.

JavaScript is **high-level**, **dynamic** and **browser interpreted** programming language, supported by all modern web browsers.

Apart from web browser, JavaScript is also used to build scalable web applications using **Node JS** or **Deno.**

JavaScript is also known as the **Programming Language of Web** as it is the only programming language for Web browsers.

Introduction



Introduction

- **What Can JavaScript Do?**
  - Common JavaScript tasks can replace server-side scripting.
  - JavaScript enables form validation, calculations, special graphic and text effects, image swapping, image mapping, clocks, and more.
- **Some other tasks like,**
  - Handling User Interaction
  - Doing small calculations
  - Checking for accuracy and appropriateness of data entry from forms
  - Doing small calculations/manipulations of forms input data
  - Search a small database embedded in the downloaded page
  - Save data as cookie so it is there upon visiting the page

JavaScript Applications

- Frontend Development
- Backend Development
- Software Development
- Mobile Apps Development
- Web Animations
- Games
- AR and VR
- 2D and 3D

Advantages of JavaScript

- Less server interaction
- Immediate feedback to the visitors
- Increased interactivity
- Richer interfaces
- Web surfers don’t need a special plug-in to use your scripts.
- Java Script is relatively secure.

Limitations with JavaScript

We can not treat JavaScript as a full-fledged programming language.

It lacks the important features like:

Not allow the reading or writing of files.

It can not be used for Networking

It doesn't have any multithreading or multiprocessor capabilities.

What is a script?

Script is a small, embedded program.

The most popular scripting languages on the web are, JavaScript or VBScript.

HTML does not have scripting capability. You need to use **<script>** tag.

The **<script>** tag is used to generate a script.

The **</script>** tag indicates the end of the script or program.

JavaScript Example

- **<script type="text/javascript">**

  and end with

  </script>

This type attribute is no more required in new version

- **<script type="text/javascript">**

  <!–-

  and end with

  // -->

  </script>

This method is not used in modern web browsers

JavaScript Structure

```
<html>
  <head>
      <script type=“text/javascript”>
          document.write("Simple javascript");
      </script>
  </head>
</html>
```

Types of JavaScript

- **Internal JavaScript**

```
<script>
        document.write("Hello Javascript");
</script>
```

- **External JavaScript**

```
<script src="custom.js">
</script>
```

```
<script src="custom.js">
   alert(1) ; // the content is
       ignored, because src is set
</script>
```

Types of JavaScript

- **Inline JavaScript**

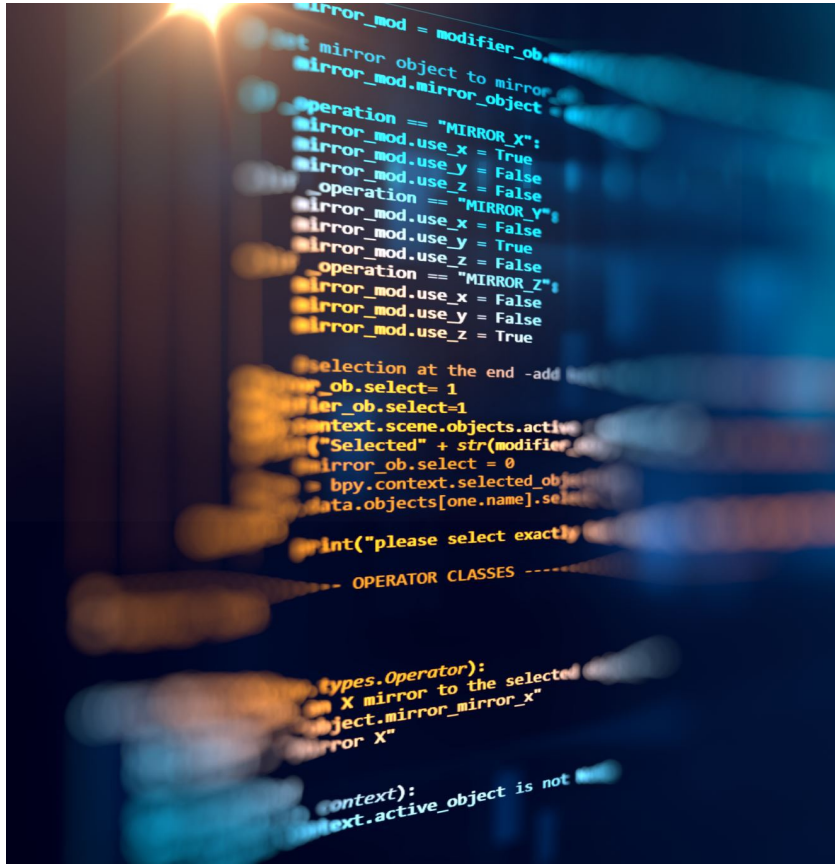
```
<button onclick="alert('Hello JS')">Check</button>
```

```
<marquee onmouseover="stop()" onmouseout="start()">Hello Javascript</marquee>
```

Types of Scripts

Client-Side Scripting

- Client-Side Scripting is an important part of the Dynamic HTML (DHTML).
- JavaScript is the main client-side scripting language for the web.
- The scripts are interpreted by the browser.
- Client-Side scripting is used to make changes in the web page after they arrive at the browser.
- It is useful for making the pages a bit more interesting and user-friendly.
- It provides useful gadgets such as calculators, clocks etc.
- It enables interaction within a web page.
- It is affected by the processing speed of the user's computer.



Types of Scripts

Server-Side Scripting

- Server-Side Scripting is used in web development.
- The server-side environment runs a scripting language which is called a web server.
- Server-Side Scripting is used to provide interactive web sites.
- It is different from Client-Side Scripting where the scripts are run by viewing the web browser, usually in JavaScript.
- It is used for allowing the users to have individual accounts and providing data from databases.

Types of Scripts

Server-Side Scripting

- It allows a level of privacy, personalization and provision of information that is very useful.
- It includes ASP.NET and PHP.
- It does not rely on the user having specific browser or plug-in.
- It is affected by the processing speed of the host server.

Methods of Using JavaScript

There are three ways of executing JavaScript on a web browser.

1. Inside <HEAD> tag
2. Within <BODY> tag
3. In an External File

Methods of Using JavaScript

- **Inside <HEAD> tag**

```
<script language="javascript”  type="text/javascript">
    <!-- Begin hiding
        <script type="text/javascript">
            function f() {
                document.write("Welcome to JavaScript....")
            }
        </script>
        // End hiding script-->
</script>
```

Methods of Using JavaScript

- **Inside <BODY> tag**

```
<body>
        <script>
                document.write("This Script is executed by Body tag....")
        </script>
</body>
```

Methods of Using JavaScript

- **External Script**
  - **HTML File**

```
<head>
   <meta charset="UTF-8">
   <title>External Javascript Example</title>
   <script type="text/javascript" src="js/ExternalJs.js" ></script>
</head>
```

  - **JS File**

```
document.write("This is executed by External Javascript Method...")
```

Using Comment Tags

```
/* An example with two messages.
This is a multiline comment. */

alert('Hello’);
alert('World');
```

```
// This comment occupies a line of its own

alert('Hello’);

alert('World'); // This comment follows the statement
```

Use hotkeys!

In most editors, a line of code can be commented out by pressing the `Ctrl+/` hotkey for a single-line comment and something like `Ctrl+Shift+/` – for multiline comments (select a piece of code and press the hotkey). For Mac, try `Cmd` instead of `Ctrl` and `Option` instead of `Shift`.

Weakly Typed Language

let name = “abhishek”

let object= {name: “abhishek”} ;

Strongly Typed Language

Integer num= 1 ;

Cat cat = Cat() ;

Strongly Typed Language : C
C++ Java

Variables and Literals

- Most of the time, a JavaScript application needs to work with information. Here are two examples:
  1. **An online shop** – the information might include goods being sold and a shopping cart.
  2. **A chat application** – the information might include users, messages, and much more.
- **Variables are used to store this information.**
- **A variable is a "named storage" for data.**
- We can use variables to store goodies, visitors, and other data.

Variables and Literals

In Older version of JavaScript variable definition should always start with **'var'.**

```
var message = "hello";
```

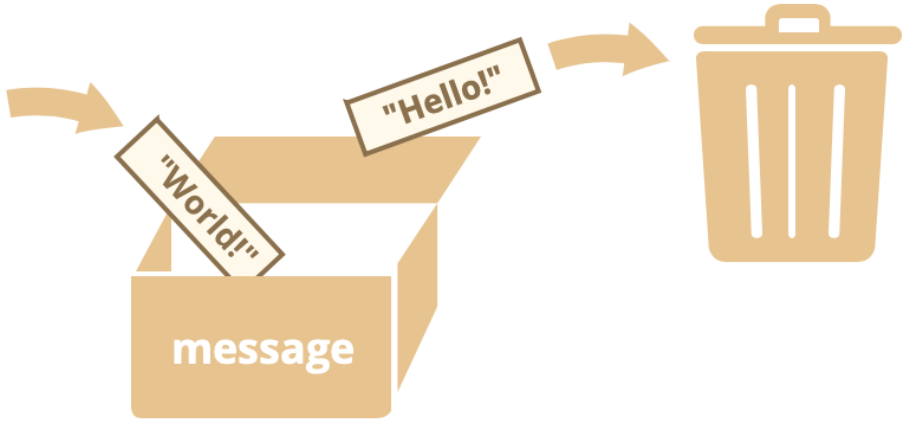
No types are declared

JS is dynamically typed language	Same variable can hold different types during the life of the execution

Variables and Literals

- Variables contain values and use the equal sign to specify their value.
- Variables are created by declaration using the **var** command with or without an initial value state.
  - e.g. **var message**
  - e.g. **var message = “Hello”;**

We can put any value in the box.



When the value is changed, the old data is removed from the variable:

Variables and Literals

JavaScript **let** keyword

In JavaScript, **let is a keyword** that is used to declare a **block scoped** variable.

e.g. **let variable_name;**

e.g. **let message = “Hello”;**

let keyword was introduced later by the ES2015/ES6, which create **block scope** variable.

Variables and Literals

Declaring twice triggers an error

```
<script>
"use strict";
let message = "This";
// repeated 'let' leads to an error
let message = "That"; // Syntax Error: 'message'
has already been declared
</script>
```

Variables and Literals

JavaScript **const** keyword

ES6 introduced the **const** keyword, which is used to define a new variable in JavaScript.

- e.g. **const variable_name;**
- e.g. **const message = “Hello”;**

const is another keyword to declare a variable when you do not want to change the value of that variable for the whole program.

Variables and Literals

Declaration

- – **Explicit**: var i = 12;
- – **Implicit**: i = 12;

Variable Scope

- **Global**
  - Declared outside functions
  - Any variable ***implicitly*** *defined*
- **Local**
  - Explicit declarations inside functions

Variable Scope

Scope determines the visibility and accessibility of a variable.

JavaScript has three scopes:

Global Scope

Local Scope

Block Scope

Global scope

- When you execute a script, the JavaScript engine creates a global execution context.
- It also assigns variables that you declare outside of functions to the **global execution context.**
- These variables are in the **global scope**.
- They are also known as **global variables**.

```
var message = 'Hi';
```

Local scope

- Variables that you declare inside a function are local to the function. They are called **local variables.**

```
var message = 'Hi';

function say() {
    var message = 'Hello';
    console.log('message');
}

say();
console.log(message);
```

Output:

```
Hello
Hi
```

Global variable leaks: the weird part of JavaScript

In this example, we **assigned 10** to the counter variable without the **var, let, or const** keyword and then returned it.

Outside the function, we called the **getCounter()** function and showed the result in the console.

This issue is known as the **leaks of the global variables**.

```
function getCounter() {
    counter = 10;
    return counter;
}

console.log(getCounter());
```

Output:

```
10
```

Global variable leaks: the weird part of JavaScript

- The JavaScript engine first looks up the counter variable in the local scope of the **getCounter()** function.
- Because there is no **var, let, or const** keyword, the counter variable is not available in the local scope. It hasn't been created.
- Then, the JavaScript engine follows the scope chain and looks up the counter variable in the global scope.
- The global scope also doesn't have the counter variable, so the JavaScript engine creates the counter variable in the global scope.

```
function getCounter() {
    counter = 10;
    return counter;
}

console.log(getCounter());
```

Output:

```
10
```

Use of Strict Keyword

- To fix this “weird” behavior, you use the **'use strict'** at the top of the script or at the top of the function:

```
'use strict'

function getCounter() {
    counter = 10;
    return counter;
}

console.log(getCounter());
```

Now, the code throws an error:

```
ReferenceError: counter is not defined
```

```
function getCounter() {
    'use strict'
    counter = 10;
    return counter;
}

console.log(getCounter());
```

Block scope

- **ES6** provides the let and const keywords that allow you to declare variables in block scope.
- Generally, whenever you see curly **brackets {},** it is a block.
- It can be the area within the if, else, switch conditions or for, do while, and while loops.

```
function say(message) {
    if(!message) {
        let greeting = 'Hello'; // block scope
        console.log(greeting);
    }
    // say it again ?
    console.log(greeting); // ReferenceError
}

say();
```

Data Types

- As JavaScript and all scripting languages are **loosely typed**, there is no **typecast** in JavaScript.
- JS supports We can create a **dynamic typing .**
- Any type of data using a single **variable**.
  - **Number**
  - **String**
  - **Boolean**
  - **Function**
  - **Object**
  - **Array**
  - **Map**
  - **Set**

typeof Operator

- **typeof** operator in JavaScript is used to check **data type** of a variable.
- It can return string, number, Boolean and undefined.
- For reference type and null, **typeof** operator will return object.

```
var x; // undefined
var y=9; // number
var z="Tech firm"; // string

typeof(x) // typeof x will return undefined,
typeof(y) // typeof y will return number,
typeof(z) // typeof z will return string
```

Template Literals

- ES6 introduced **Template Literals** or **Template Strings** in JavaScript by using **back-tick** or **grave characters.**

```
var str = `Java Script`; // undefined
```

- To string interpolation or to insert a placeholder variable in **template literal**, use **${expression}** inside.

```
var x=3;
var y=5;
console.log(`sum of ${x} and ${y} is ${x+y}`); //ES6
console.log("sum of " + x + " and " + y + " is " + (x+y)); //ES5
// returns "sum of 3 and 5 is 8"
```

Multi line string

- **Template literals** can also add multi-line strings. Till ES5, we use + operator to do the same.

```
var template=`<!doctype html>
              <html>
              <head>
              <meta charset="UTF-8">
              <title></title>
              </head>
              <body>
              </body>
              </html>`;
```

JavaScript Operators

- **Arithmetic Operators**

Operator	Description	Example	Result
+	Addition	x + y	Sum of x and y
-	Subtraction	x - y	Difference of x and y.
*	Multiplication	x * y	Product of x and y.
/	Division	x / y	Quotient of x and y
%	Modulus	x % y	Remainder of x divided by y

JavaScript Operators

- **Comparison Operators**

Operator	Name	Example	Result
==	Equal	x == y	True if x is equal to y
===	Identical	x === y	True if x is equal to y, and they are of the same type
!=	Not equal	x != y	True if x is not equal to y
!==	Not identical	x !== y	True if x is not equal to y, or they are not of the same type
<	Less than	x < y	True if x is less than y
>	Greater than	x > y	True if x is greater than y
>=	Greater than or equal to	x >= y	True if x is greater than or equal to y
<=	Less than or equal to	x <= y	True if x is less than or equal to y

JavaScript Operators

- **Assignment Operators**

Operator	Description	Example	Is The Same As
=	Assign	x = y	x = y
+=	Add and assign	x += y	x = x + y
-=	Subtract and assign	x -= y	x = x - y
*=	Multiply and assign	x *= y	x = x * y
/=	Divide and assign quotient	x /= y	x = x / y
%=	Divide and assign modulus	x %= y	x = x % y

JavaScript Operators

- **Logical Operators**

Operator	Name	Example	Result
&&	And	x && y	True if both x and y are true
||	Or	x || y	True if either x or y is true
!	Not	!x	True if x is not true

JavaScript Operators

- **Bitwise Operators**

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Left Shift
>>	Right Shift
>>>	Right Shift with Zero

JavaScript Operators

- **Special Operators**

Operator	Description
NEW	Creates an instance of an object type.
DELETE	Deletes property of an object.
DOT(.)	Specifies the property or method.
VOID	Does not return any value. Used to return a URL with no value.

Type Conversion

Numeric Conversion

```
alert( "6" / "2" ); // 3, strings are converted to numbers
```

```
let str = "123";
alert(typeof str); // string

let num = Number(str); // becomes a number 123

alert(typeof num); // number
```

We can use the Number(value) function to explicitly convert a value to a number:

String Conversion

```
let value = true;
alert(typeof value); // boolean

value = String(value); // now value is a string "true"
alert(typeof value); // string
```

Type Conversion

- 3 + 3 // results in 6
- 3 + “3” // results in “33”
- 3 + 3 + “3” /results in “63”

Converting String to Number

- parseInt(“33”); // results in 33
- parseInt(33.33); // results in 33
- parseFloat(“33”); // results in 33
- parseFloat(“33.33”); // results in 33.33
- 3 + 3 + parseInt(“3”); // results in 9

Converting Number to String

- “” + 2500 // results in “2500”
- (“” + 2500).length // results in 4

Type Coercion

let a = 3 let b = “4” ;

concat	a+ b		“34”
Multiply	a*b		12
Subtract	a-b		-1

Type Coercion

let a = 3 let b = “hi” ;

Concat	a + b	→	“3hi”
Multiply	a*b	→	NaN
Subtract	a-b	→	NaN

Type Conversion

```
let a = "3"  let  b = 4  ;
```

String to Number

Number(a) → 3

Number to String

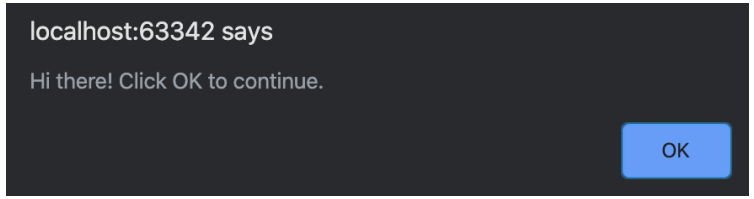
String(b) → "4"

JavaScript Pop-up Boxes

1. Alert Box
2. Confirm Box
3. Prompt Box

Alert Dialog Boxes

- An alert dialog box is the simplest dialog box.
- It enables you to display a short message to the user.
- It also includes **OK** button, and the user must click this OK button to continue.




```
var message =
"Hi there! Click OK to continue.";

alert(message);
/* The following line won't
execute until you dismiss
previous alert */

alert("This is another alert box.");
```

Confirm Dialog Boxes

A **confirm dialog box** allows user to **confirm or cancel** an action.

A confirm dialog looks similar to an alert dialog but it includes a Cancel button along with the OK button.

You can create confirm dialog boxes with the **confirm()** method.

This method simply returns a Boolean value (true or false) depending on whether the user clicks **OK** or **Cancel** button.

That's why its result is often assigned to a variable when it is used.

Confirm Dialog Boxes

```
var result = confirm("Are you sure?");
if (result)
{
     document.write("You clicked OK button!");
}
else
{
     document.write("You clicked Cancel button!");
}
```

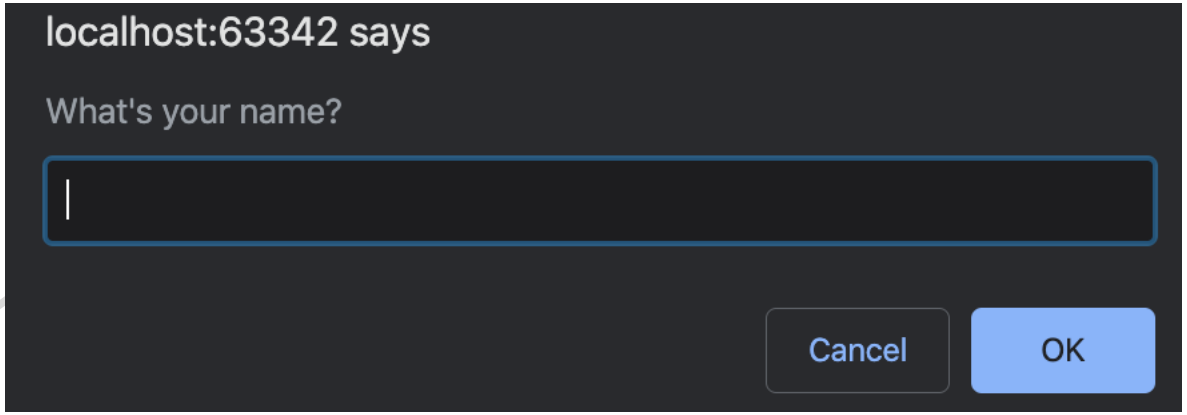
You clicked OK button!

Prompt Dialog Box

- The **prompt dialog box** is used to prompt the user to enter information.
- A prompt dialog box includes a text input field, an **OK** and a **Cancel** button.
- You can create prompt dialog boxes with the **prompt()** method.
- This method returns the text entered in the input field when the user clicks the **OK button**, and null if user clicks the **Cancel button**.
- If the user clicks OK button without entering any text, an empty string is returned.
- For this reason, its result is usually assigned to a variable when it is used.

Prompt Dialog Box

```
var name = prompt("What's your name?");
if (name.length > 0 && name != "null")
{
     document.write("Hi, " + name);
}
else
{
     document.write("Anonymous!");
}
```



Hi, Krunal

JavaScript – Control and Looping

- There are several conditional statements in JavaScript that you can use to make decisions:
  - The **if** statement
  - The **if...else** statement
  - The **if...else if....else** statement
  - The **switch...case** statement

JavaScript – Control and Looping

```
if (condition)
{
// Code to be executed
}
```

```
var result = (condition) ? value1 : value2
```

```
if (condition)
{
// Code to be executed if condition is
true

}
else
{
// Code to be executed if condition is
false

}
```

JavaScript – Control and Looping

```
switch (x)
{
case value1:
// Code to be executed if x === value1

break;
case value2:
// Code to be executed if x === value2

break;
...
default:
// Code to be executed if x is
different from all values
}
```

```
if (condition1)
{
/* Code to be executed if condition1 is true*/
}
else if (condition2)
{
/* Code to be executed if the condition1 is false
and
condition2 is true
*/
}
else
{
/* Code to be executed if both condition1 and
condition2 are false
*/
}
```

Different Types of Loops in JavaScript

- **while** — loops through a block of code as long as the condition specified evaluates to true.
- **do…while** — loops through a block of code once; then the condition is evaluated. If the condition is true, the statement is repeated as long as the specified condition is true.
- **for** — loops through a block of code until the counter reaches a specified number.
- **for…in** — loops through the properties of an object.
- **for…of** — loops over iterable objects such as arrays, strings, etc.

JavaScript – Control and Looping

```
while (condition)
{
// Code to be executed
}
```

```
do
{
// Code to be executed
}
while (condition);
```

```
for (initialization; condition; increment)
{
// Code to be executed
}
```

```
for (variable in object)
{
// Code to be executed
}
```

```
for (let letter of letters)
{
        console.log(letter); // a,b,c,d,e,f
}
```

Arrays

- **Arrays** are complex variables that allow us to store more than one value or a group of values under a single variable name.
- **JavaScript arrays** can store any valid value, including strings, numbers, objects, functions, and even other arrays, thus making it possible to create more complex data structures such as an array of objects or an array of arrays.

Defines different variable for same group

```
var color1 = "Red";
var color2 = "Green";
var color3 = "Blue";
```

Arrays

1

Creating Arrays

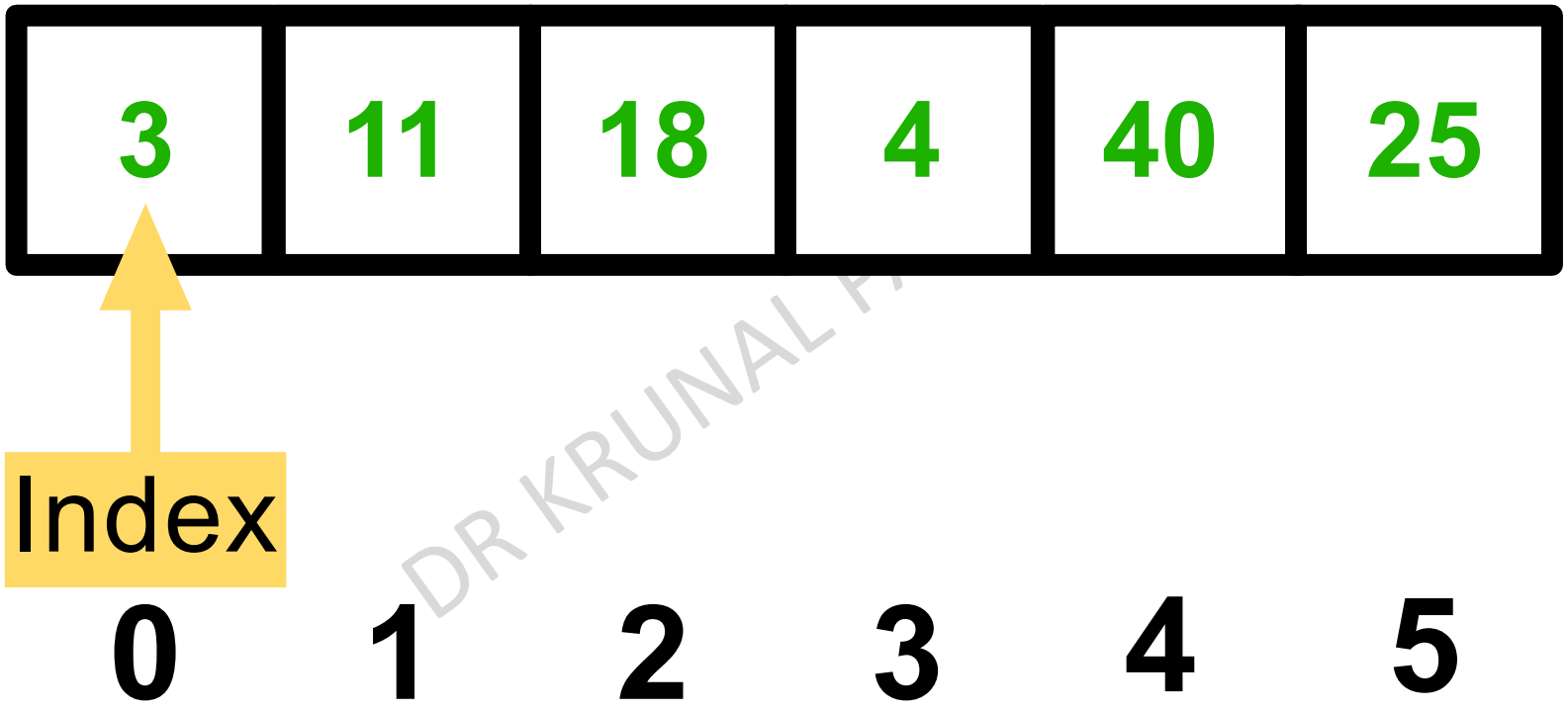
- var a = new Array(); *// empty array*
- var b = new Array('dog', 3, 8);
- var c = new Array(10); *// array of size 10*
- var d = [2, 5, 'a', 'b'];

2

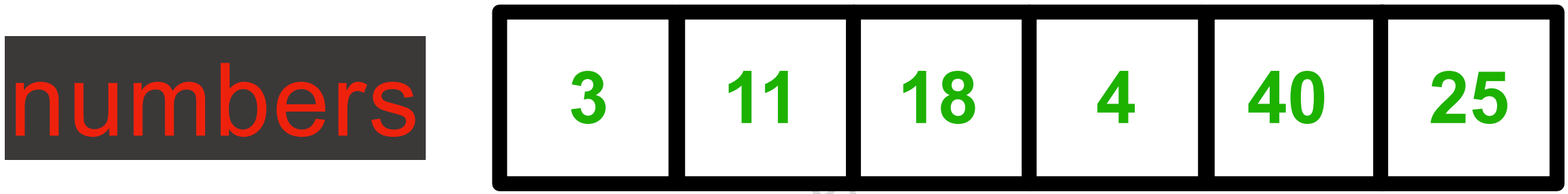
Assigning values to Arrays

- c[15] = "hello";
- c.push("hello");

Array



Initialising Array in JS



Empty Array

```
let numbers = [ ];
```

```
let numbers = [3,11,18,4,40,25];
```

Creating an Array

```
var myArray = [element0, element1, ..., elementN];
```

```
var myArray
= new Array(element0, element1, ..., elementN);
```

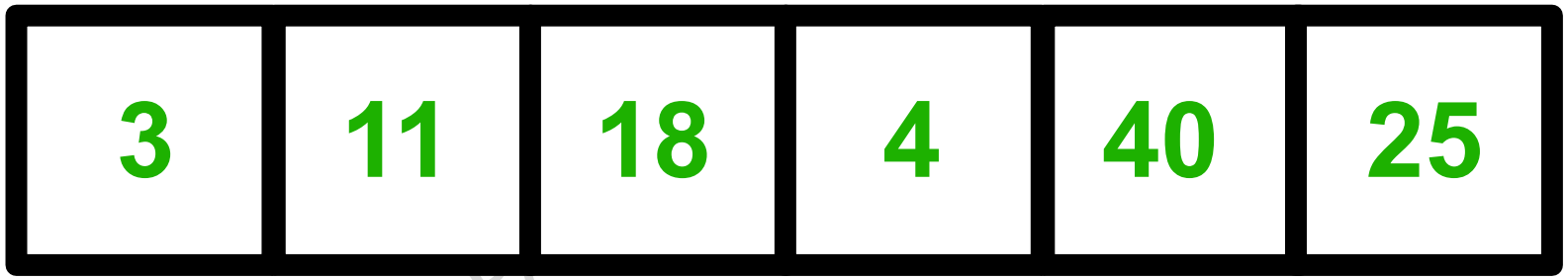
```
var colors = ["Red", "Green", "Blue"];
var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"];
var cities = ["London", "Paris", "New York"];
var person = ["John", "Wick", 32];
```

Accessing the Elements of an Array

```
var fruits = ["Apple", "Banana", "Mango",
"Orange", "Papaya"];
document.write(fruits[0]);// Prints: Apple
document.write(fruits[1]);// Prints: Banana
document.write(fruits[2]);// Prints: Mango
document.write(fruits[fruits.length - 1]);//
Prints: Papaya
```

Reading Array

numbers

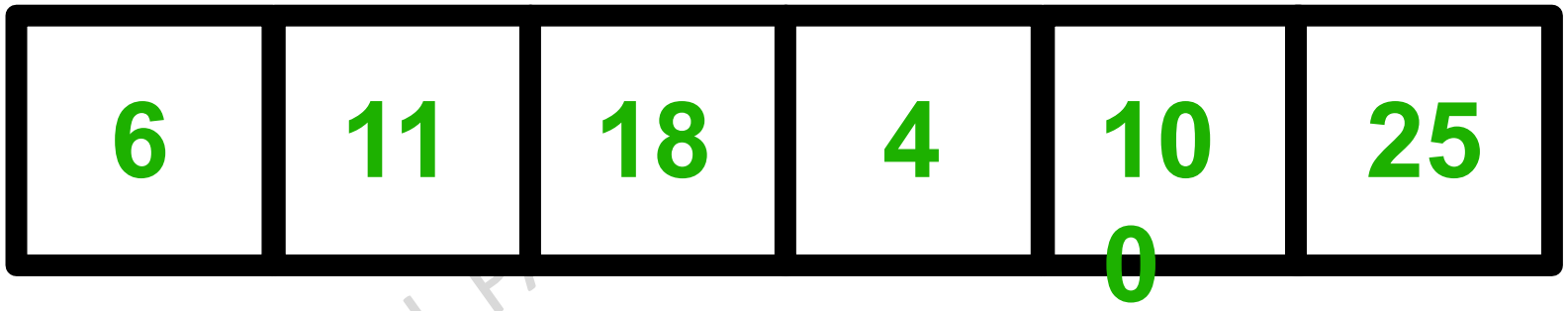


numbers[0] → 3

numbers[4] → 40

Writing Array

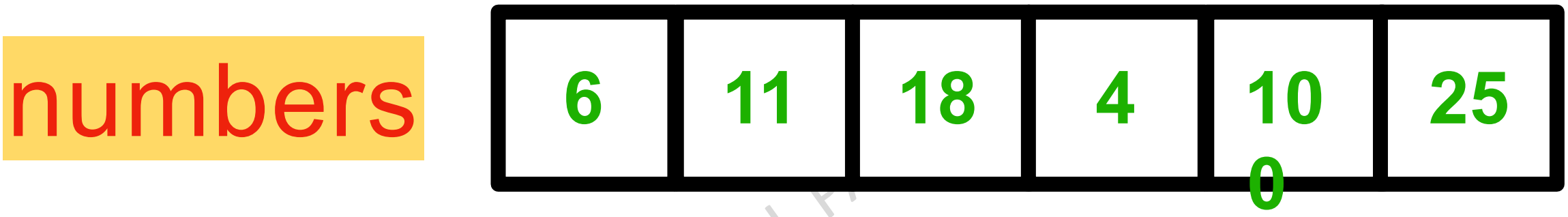
numbers



numbers[0]= 6

numbers[4] = 10

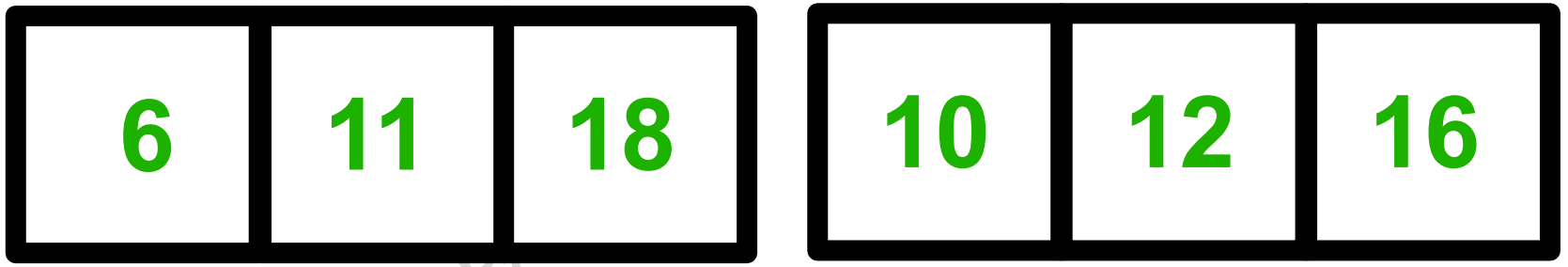
Array : length property



numbers.length → 6

PUSH function

numbers




```
numbers.push(10 )
numbers.push(12 )
numbers.push(16 )
```

4

5

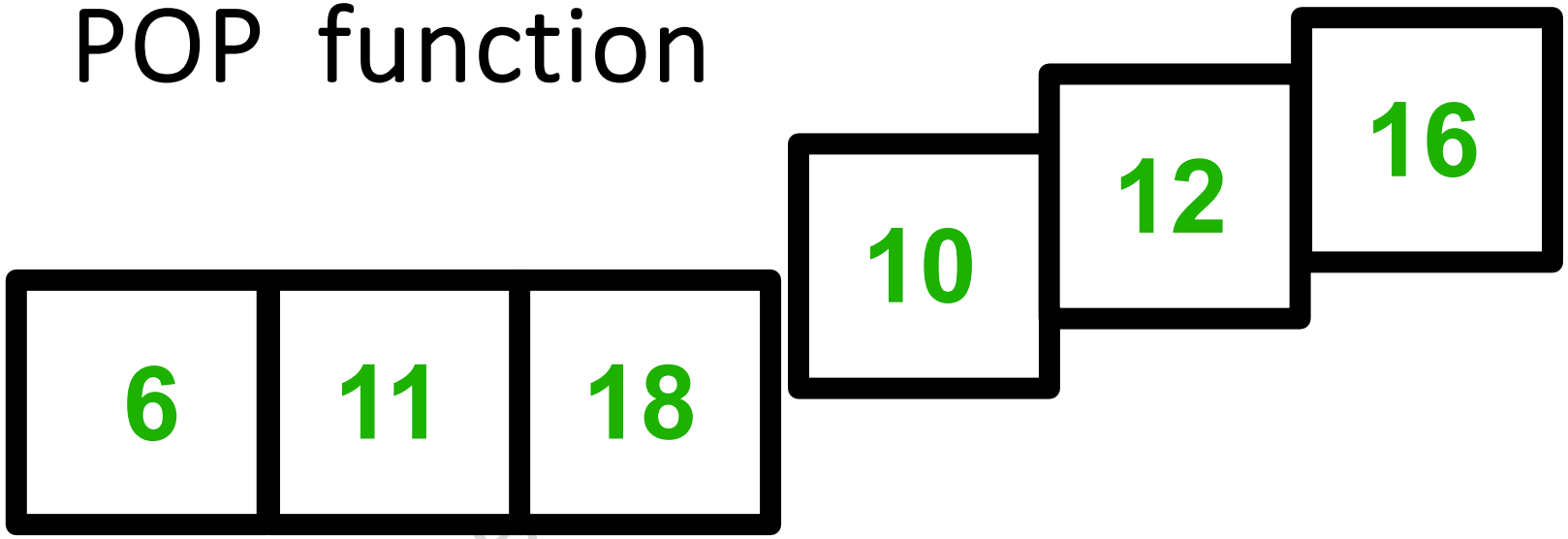
6

Mutating Method

array length after push

POP function

numbers




```
numbers.pop( ) ➡ 16
numbers.pop( ) ➡ 12
numbers.pop( ) ➡ 10
```

Mutating Method

indexOf function

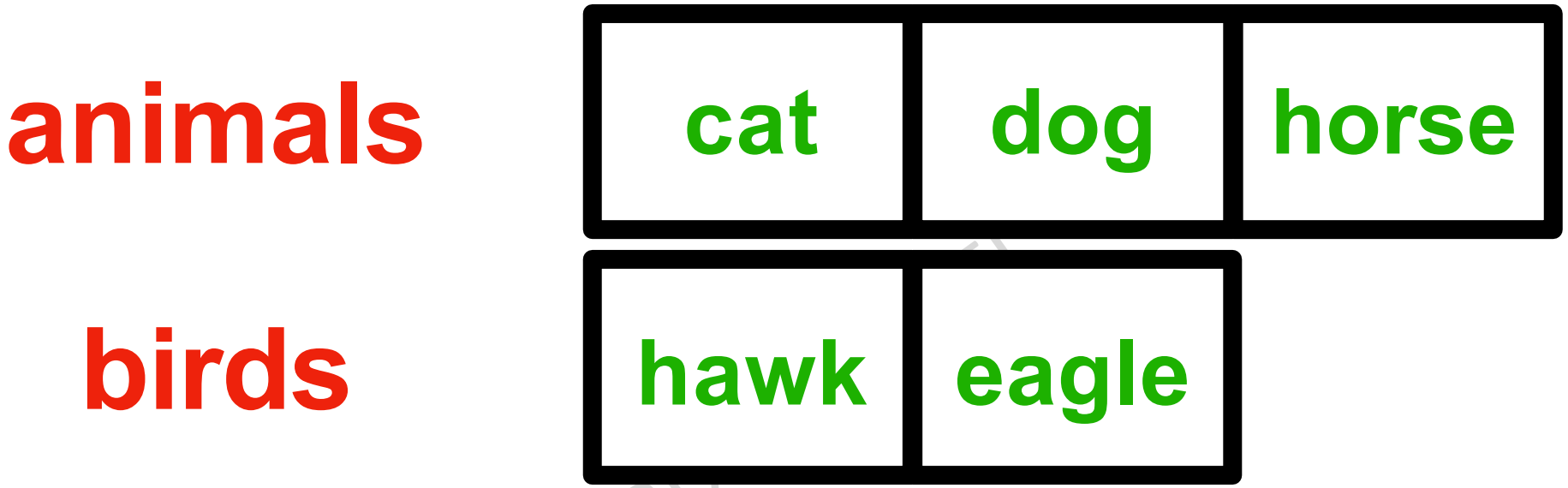
words

words.indexOf(“cat”) ➡ **0**

words.indexOf(“fox”) ➡ **-1**

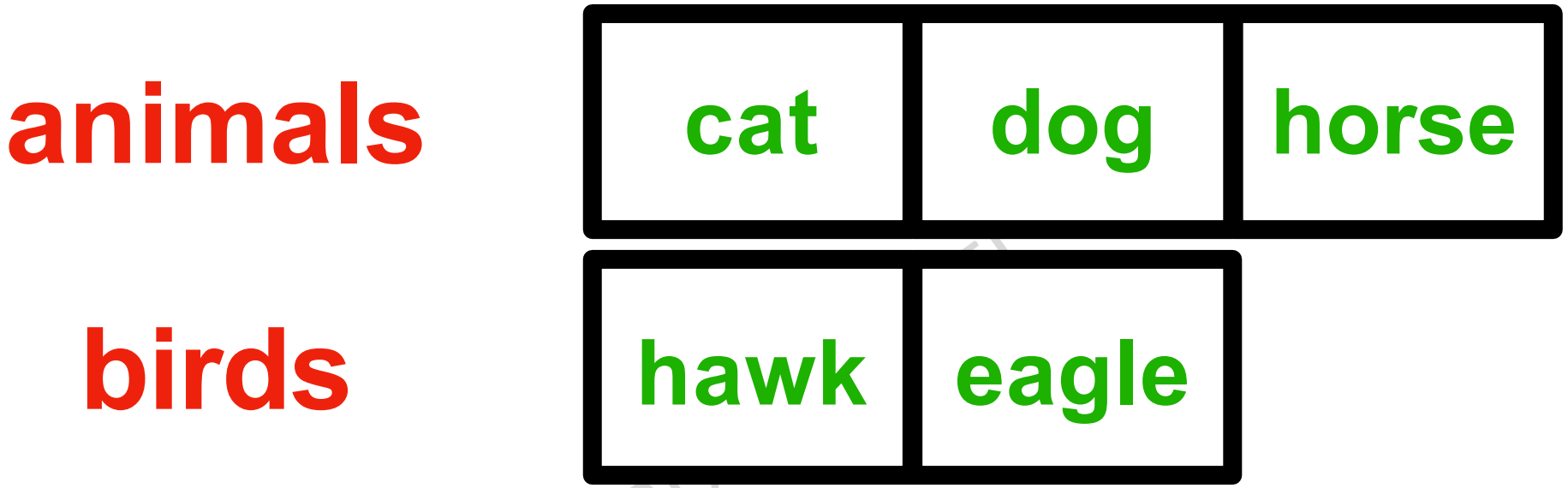
Non-Mutating Method

CONCAT function

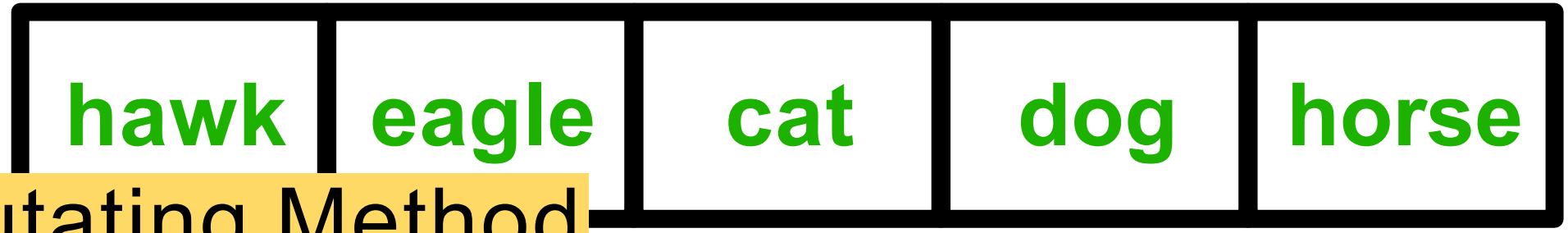


animals.concat(birds)

CONCAT function



birds.concat(animals)



Non-Mutating Method

Function

Function is a set of code, that is defined once and can be called n number of times.

A **function** can be reused; thus, they are also used to avoid repetition of code.

Function are ***reference datatype*** in JavaScript.

Function includes multiple statements to perform task.

A function can take input and then **return** output.

Functions are one of the major building blocks in JavaScript.

Function

- JavaScript Function Declaration

```
<script>
	function add(){
		// code inside
	}
</script>
```

```
<script>
	var myfunction=function(){
		// code inside
	};
</script>
```

Functions

Functions

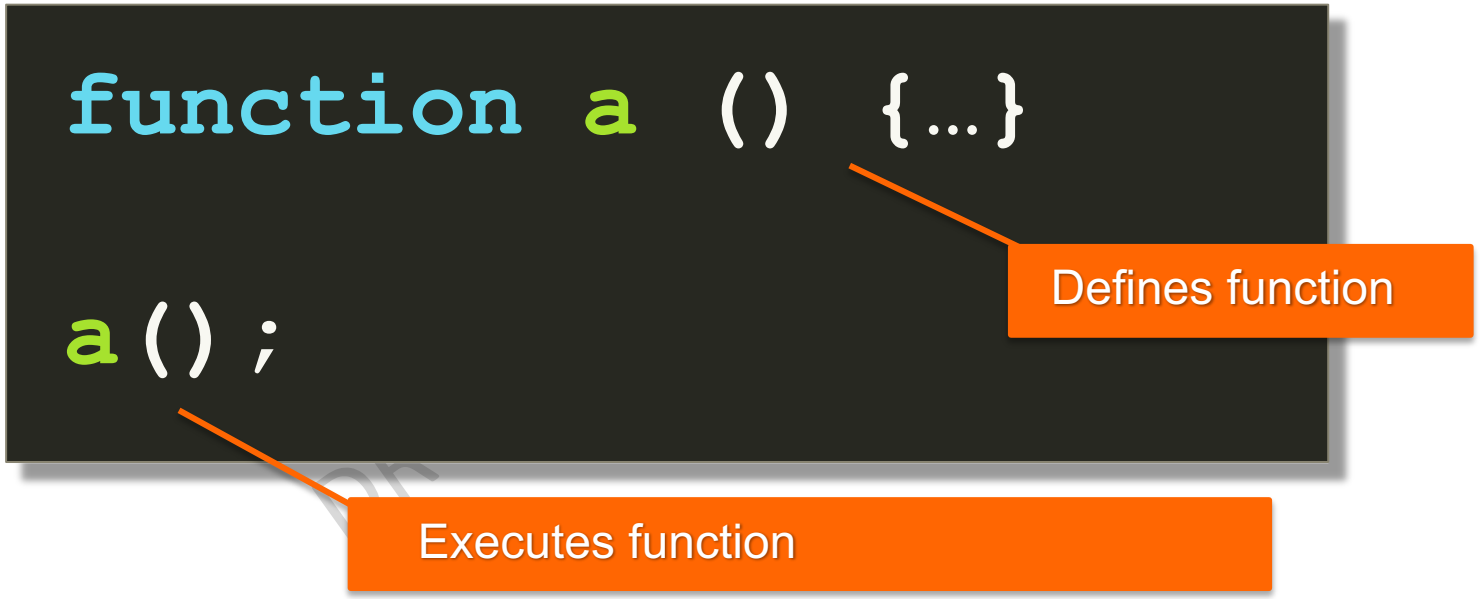
```
function a () {…}
```

```
var a = function () {…}
```

Value of function assigned, NOT the returned result!

No name defined

Functions



Functions

Arguments defined without ‘var’

```
function compare (x, y) {
   return x > y;
}
```

Functions

```
function compare (x, y) {…}
var a = compare(4, 5);
compare(4, "a");
compare();
```

ALL LEGAL

Function Expression

- Another way to create function is **function expression**.
- In **function expression**, a **variable is declared and then assigned a anonymous function**, as it does not have a name.
- They are not named functions, as they are stored inside a variable.
- **Function Expression** are only invoked after function.
- If we call a **function expression** before, an error will occur (function_name is not defined).

Function expression

```
var sum = function(a, b, c){
    return a + b + c;
}
```

Declared like variable

No name (anonymous function

sum(2,3,4) ➡ **9**

Function Expression

```
<script>
        varFunc() // will not work
        var varFunc = function(){
                                alert(”Hello Variable")
                        };
                varFunc() // will work
</script>

<button onclick=”varFunc()">Click</button>
```

Both Type of definition work Same

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

```
var sum = function(a, b, c){
    return a + b + c;
}
```

sum(2,3,4) ➡ 9

Normal function definition can be called before initialisation

sum(2,3,4)  **9**

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

Reason : Hoisting

function expression Can't be called before initialisation

sum(2,3,4) → ERROR


```
var sum = function(a, b, c){
    return a + b + c;
}
```

Hoisting

sum(2,3,4)  **9**

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

JS Interpreter reads function definition before executing code

Default Parameters

```
let weight = function(m, g=9.8) {
    return m * g;
}
```

```
weight(10,9)   ➡   90
weight(10)     ➡   98
```

10 *9.8

Arrow Functions

- **Arrow Function:** Arrow functions are handy for one-liners. : **ES6**
- They come in two flavors:
  - **Without curly braces**: **(...args) => expression** – the right side is an expression: the function evaluates it and returns the result.
  - **With curly braces**: **(...args) => { body }** – brackets allow us to write multiple statements inside the function, but we need an explicit return to return something.


```
const x = (x,y) => { return x * y};
let x1 = x(100,200);
document.write(x1);
```

Arrow Functions

```
let sum = function(a, b, c){
      return a + b + c;
       }
```

```
let sum = (a, b, c) => {
return a + b + c;
 }
```


parameters

Arrow

Arrow Functions

```
let sum = function(a, b, c){
    return a + b + c;
    }
```

```
let sum =(a, b, c) => { return a + b + c;}
```

```
let sum =(a, b, c) => a + b + c;
```

No Braces implicitly mean return

Functions v/s Arrow Functions

Functions

Good for multi-line logic

Creates a new “this” context

Arrow functions

Good for single line returns

Doesn’t create a “this” context

Higher order functions

Functions which contain other function to do some task

other function can be argument (Callback function)

other function can be inner return value (closure)

Callback Functions

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

FUNCTIONS ARE OBJECTS

can be passed to as arguments

Callback Functions

```
var sum = function(a, b){
            return a + b;
          }
```

Higher Order function

fx(sum)

Callback function

Callback Functions

‘sayHi’ called by ‘talk’ at later stage

```
var talk = function(fx) {
        fx();
}

var sayHi = function(){
    console.log(“hi”);
}

talk(sayHi)
```

Callback Functions

```
var calc = function(fx,a,b){
        return fx(a,b);
}

var sum = function(x,y){
    return a+b;
}
```

calc(sum,4,5)  **9**

Callback Functions

```
var calc = function(fx,a,b){
     return fx(a,b);
}

var diff = function(x,y){
    return a-b;
}
```


calc(diff,4,5) → **-1**

Function returning function

```
function makeFunc

   function displayName

   return displayName
```

same functionality as displayName, but can access “name” variable

```
      myFunc    makeFunc

myFunc
```

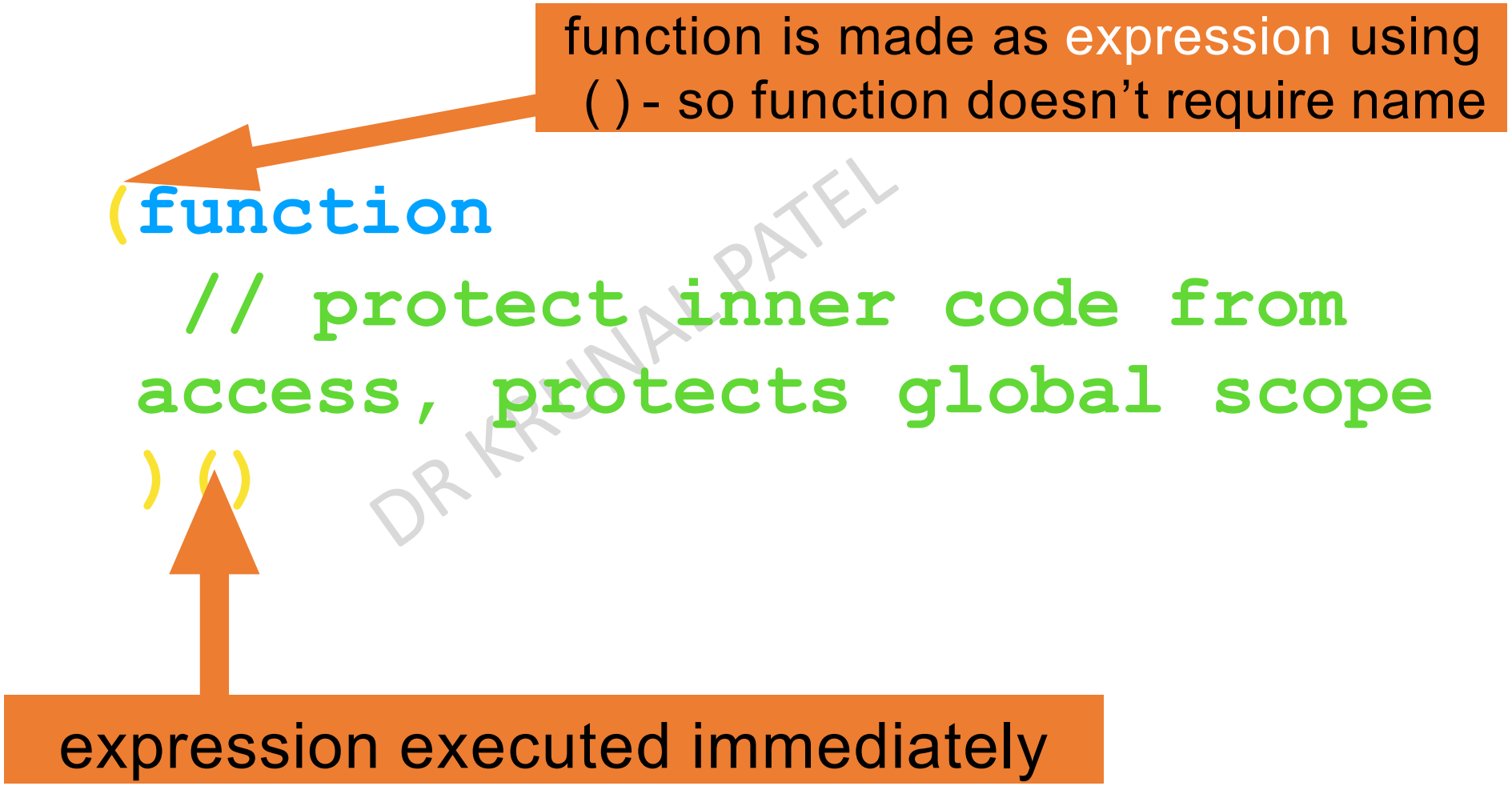
this is also example of a “Closure” which we will cover at last

Immediate Invoke Function

- **Immediate Invoke Function** or **self-invoking function** are anonymous function who call themself.
- They are not assigned and named.
- That's why they are anonymous.
- But they call themself on page load.
- They are mainly used to keep variables local.

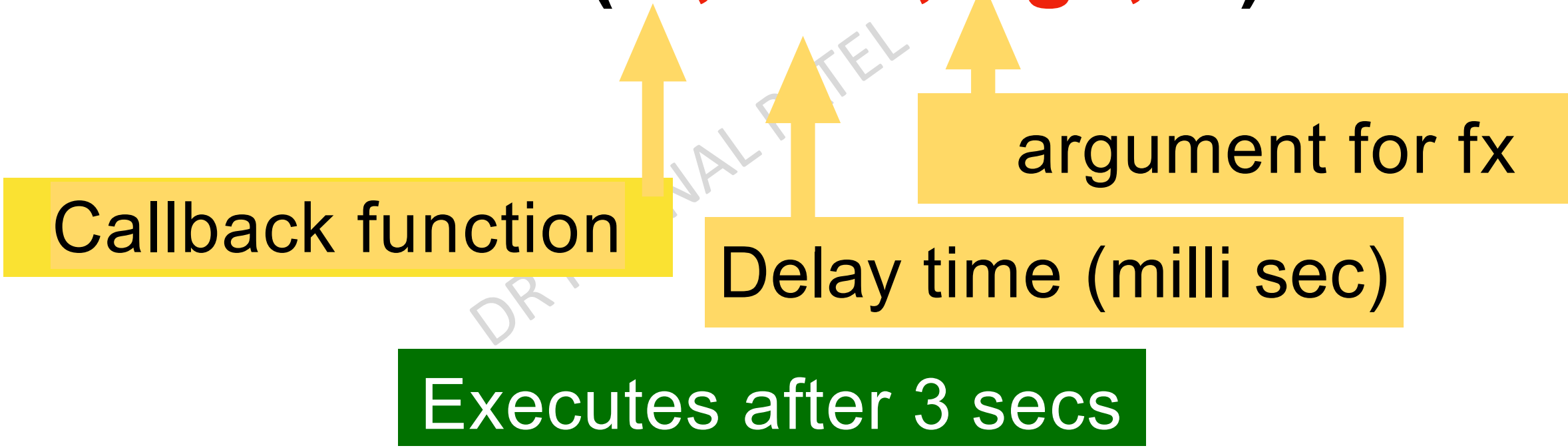

```
(function(x){
      console.log("Hello", x );
}("user"));
```

IIFE- Immediately Invoked Function Expression



Timer Functions

setTimeout(fx,3000,arg1,…)



Timer Functions

```
setTimeout ( function( ) {
                console.log("hello")
            },
        3000 )
```

Timer Functions

setInterval(fx,3000,arg1,..)

Callback function

Delay time (milli sec)

arguments for fx

Executes every 3 secs

Timer Functions

```
setInterval( function( ) {
        console.log("hello")
    },
3000 )
```

JavaScript Objects

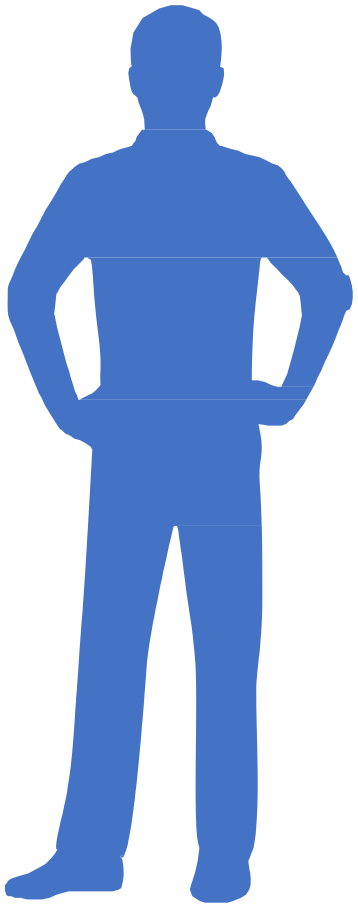
- **JavaScript is an object-based language** and in JavaScript almost everything is an object
- A JavaScript object is just a collection of named values. These named values are usually referred to as properties of the object.

Example: String JavaScript object has **length** property and toUpperCase() method

```
var txt="Hello World!”
document.write(txt.length)
document.write(txt.toUpperCase())
```

String , Date , Array , Boolean and Math are Built in javascript objects

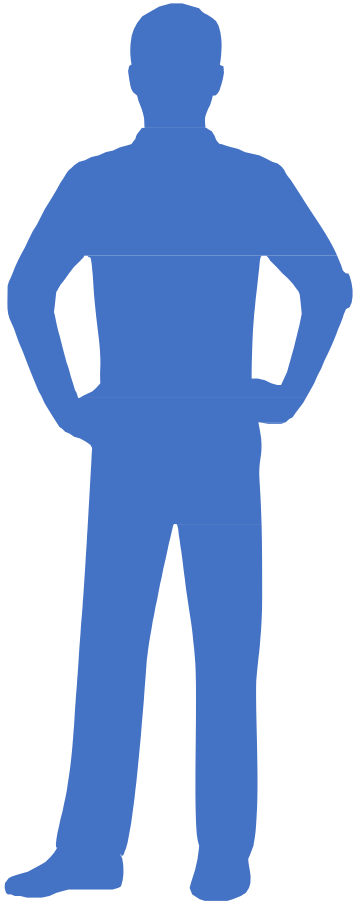
Objects



person

- Name → abhishek
- Age → 30
- Address → Street 10, mumbai, india
- Mobile → 888888888

Objects



person

```
var person = {
        name : "abhishek",
        age :30 ,
        address : "street 10, Mumbai,
        phone:8888888888
    }
```

key

value

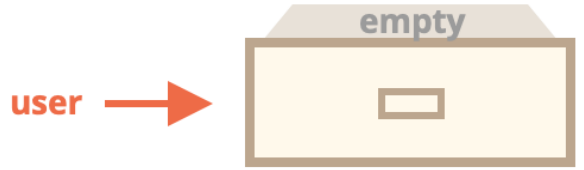
JavaScript Objects

- **Creating Objects**
  - We can imagine an object as a cabinet with signed files.
  - Every piece of data is stored in its file by the **key**.
  - It's easy to find a file by its name or add/remove a file.

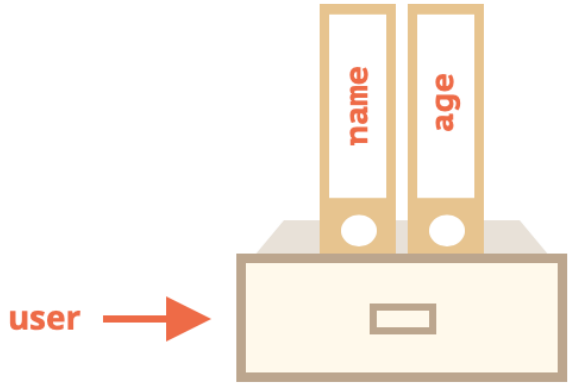
JavaScript Objects

- **An empty object (“empty cabinet”) can be created using one of two syntaxes:**

```
let user = new Object(); // "object constructor"
let user = {}; // "object literal"
```




```
let user = {// an object
        name: "John", // by key "name" store value "John"
        age: 30 // by key "age" store value 30
};
```



Accessing Object's Properties

```
var book = {
      "name": "Harry Potter and the Goblet of Fire",
      "author": "J. K. Rowling",
      "year": 2000
};

// Dot notation
document.write(book.author); // Prints: J. K. Rowling

// Bracket notation
document.write(book["year"]); // Prints: 2000
```

Calling Object's Methods

```
var person = {
    name: "Peter",
    age: 28,
    gender: "Male",
    displayName: function() {
        alert(this.name);
    }
};

person.displayName(); // Outputs: Peter
person["displayName"](); // Outputs: Peter
```

Accessing Objects (dot)

```
var person = {
        name : "abhishek",
        age :30 ,
        address : "street 10, Mumbai, India",
        phone:8888888888
    }
```

person.name → “abhishek

person.age → ” 30

Accessing Objects (bracket style)

```
var person = {
        name : "abhishek",
        age :30 ,
        address : "street 10, Mumbai, India",
        phone:8888888888
    }
```

person.[“name”] → “abhishek

person.[“age”] → ” 30

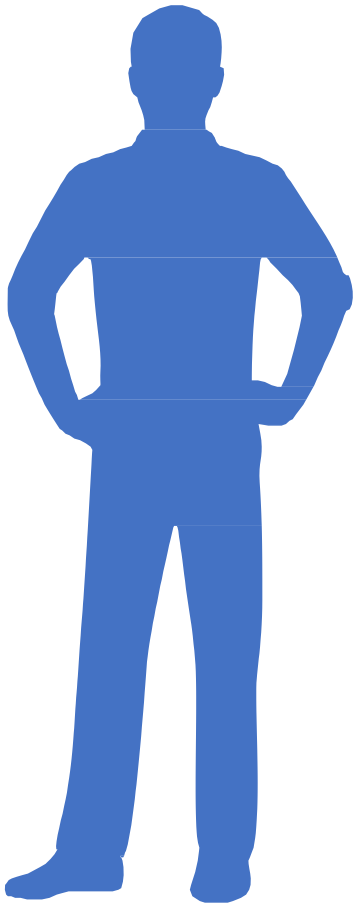
Writing Objects (dot)

```
var person = {
        name : "ajay",
        age :40 ,
        address : "street 10, Mumbai, India",
        phone:8888888888
    }
```

person.name = “ajay

person.age = ” 40

Nested Objects



person

- Name → abhishek
- Age → 30
- Address → Street 10, mumbai, india
- Mobile → 888888888

Nested Objects

```
var person = {
        name : "abhishek",
        age :30 ,
        address : {
                  street:“street 10”,
                  city:“mumbai”,
                  country:“india”
                },
        phone:8888888888
    }
```

person.address.city ➡ **“mumbai”**

person.address.country ➡ **“india”**

Deleting Properties

```
var person = {
        name : "abhishek",
        age :30 ,
        address : "street 10, Mumbai, India",
        phone:8888888888
    }
```

delete person.name

delete person.age

Function vs Methods

```
var person = {
        name : "abhishek",
        age :30 ,
        address : "street 10, Mumbai, India",
        phone:function(){ return this.age}
    }
```

methods = function of an object

this

```
const person = {
                name : "p1",
                getName: function(){
                        return this.name
                      }
}
```

```
person.getName()
```

"p1"

'this' here will refer to calling object (person)

forEach()

```
const cities = ["NY","LA","TX"];

const lowerCased = [];

cities.forEach((city) => lowerCased.push(city))
```

lowerCased → ["ny","la","tx"]

for-in loop

```
const object = { a: 1, b: 2, c: 3 };

for  const property in
  console.log(`${property}: ${object[property]}`)
```


these properties are called enumerable

JavaScript's built-in objects

- **Math Object**

Math Property	Description
SQRT2	Returns square root of 2.
PI	Returns Π value.
E \	Returns Euler's Constant.
LN2	Returns natural logarithm of 2.
LN10	Returns natural logarithm of 10.
LOG2E	Returns base 2 logarithm of E.
LOG10E	Returns 10 logarithm of E.

JavaScript's built-in objects

- **Math Object**

Methods	Description
abs()	Returns the absolute value of a number.
acos()	Returns the arccosine (in radians) of a number.
ceil()	Returns the smallest integer greater than or equal to a number.
cos()	Returns cosine of a number.
floor()	Returns the largest integer less than or equal to a number.
log()	Returns the natural logarithm (base E) of a number.
max()	Returns the largest of zero or more numbers.
min()	Returns the smallest of zero or more numbers.
pow()	Returns base to the exponent power, that is base exponent.

JavaScript's built-in objects

- **Date Object**
  - Date is a data type.
  - Date object manipulates date and time.
  - Date() constructor takes no arguments.
  - Date object allows you to get and set the year, month, day, hour, minute, second and millisecond fields.

Syntax:

```
var variable_name = new Date();
```

Example:

```
var current_date = new Date();
```

JavaScript's built-in objects

- **Date Object**

Methods	Description
Date()	Returns current date and time.
getDate()	Returns the day of the month.
getDay()	Returns the day of the week.
getFullYear()	Returns the year.
getHours()	Returns the hour.
getMinutes()	Returns the minutes.
getSeconds()	Returns the seconds.
getMilliseconds()	Returns the milliseconds.
getTime()	Returns the number of millisecond

JavaScript's built-in objects

- **Date Object**

Methods	Description
getTimezoneOffset()	Returns the timezone offset in minutes for the current locale.
getMonth()	Returns the month.
setDate()	Sets the day of the month.
setFullYear()	Sets the full year.
setHours()	Sets the hours.
setMinutes()	Sets the minutes.
setSeconds()	Sets the seconds.
setMilliseconds()	Sets the milliseconds.
getTimezoneOffset()	Returns the timezone offset in minutes for the current locale.

JavaScript's built-in objects

- **Date Object**

Methods	Description
setTime()	Sets the number of milliseconds since January 1, 1970 at 12:00 AM.
setMonth()	Sets the month.
toDateString()	Returns the date portion of the Date as a human-readable string.
toLocaleString()	Returns the Date object as a string.
toGMTString()	Returns the Date object as a string in GMT timezone.
valueOf()	Returns the primitive value of a Date object.
setTime()	Sets the number of milliseconds since January 1, 1970 at 12:00 AM.
setMonth()	Sets the month.
toDateString()	Returns the date portion of the Date as a human-readable string.

JavaScript's built-in objects

- **String Object**
  - String objects are used to work with text.
  - It works with a series of characters.

Syntax:

```
var variable_name = new String(string);
```

Example:

```
var s = new String(string);
```

JavaScript's built-in objects

- **String Object**

Properties	Description
length	It returns the length of the string.
prototype	It allows you to add properties and methods to an object.
constructor	It returns the reference to the String function that created the object.

JavaScript's built-in objects

- **String Object**

Methods	Description
charAt()	It returns the character at the specified index.
charCodeAt()	It returns the ASCII code of the character at the specified position.
concat()	It combines the text of two strings and returns a new string.
indexOf()	It returns the index within the calling String object.
match()	It is used to match a regular expression against a string.

JavaScript's built-in objects

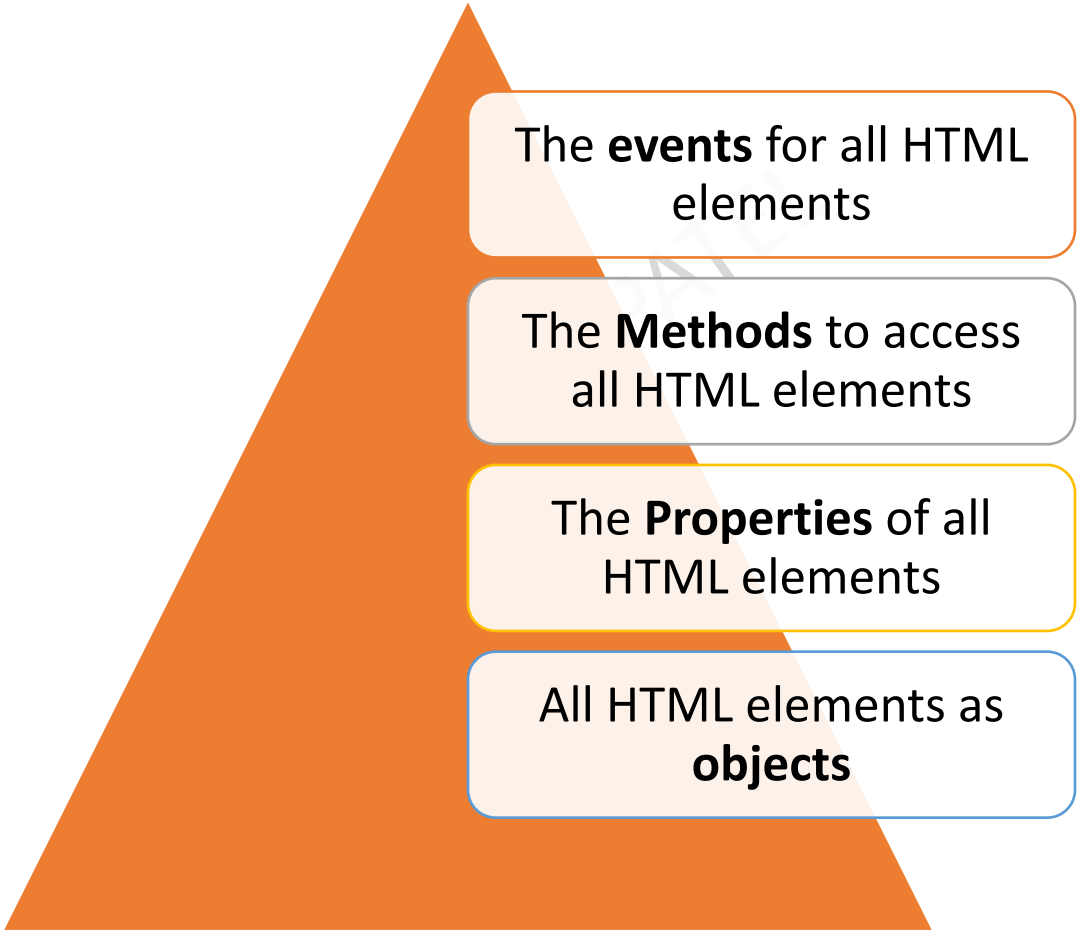
- **String Object**

Methods	Description
replace()	It is used to replace the matched substring with a new substring.
search()	It executes the search for a match between a regular expression.
slice()	It extracts a session of a string and returns a new string.
split()	It splits a string object into an array of strings by separating the string into the substrings.
toLowerCase()	It returns the calling string value converted lower case.
toUpperCase()	Returns the calling string value converted to uppercase.

JavaScript Document Object Model

- **Document Object Model (DOM)** is a standard object model and programming interface for HTML and XML documents.
- It defines a **standard for accessing documents**.
- It is a **Worldwide Consortium standard.**
- The **DOM** represents a **document as a tree of nodes**.
- It provides API that allows you to **add, remove, and modify** parts of the document effectively.
- It is a **platform and language-neutral interface** that allows programs and scripts to **dynamically access and update the content, structure and style of a document**.

What DOM defines?



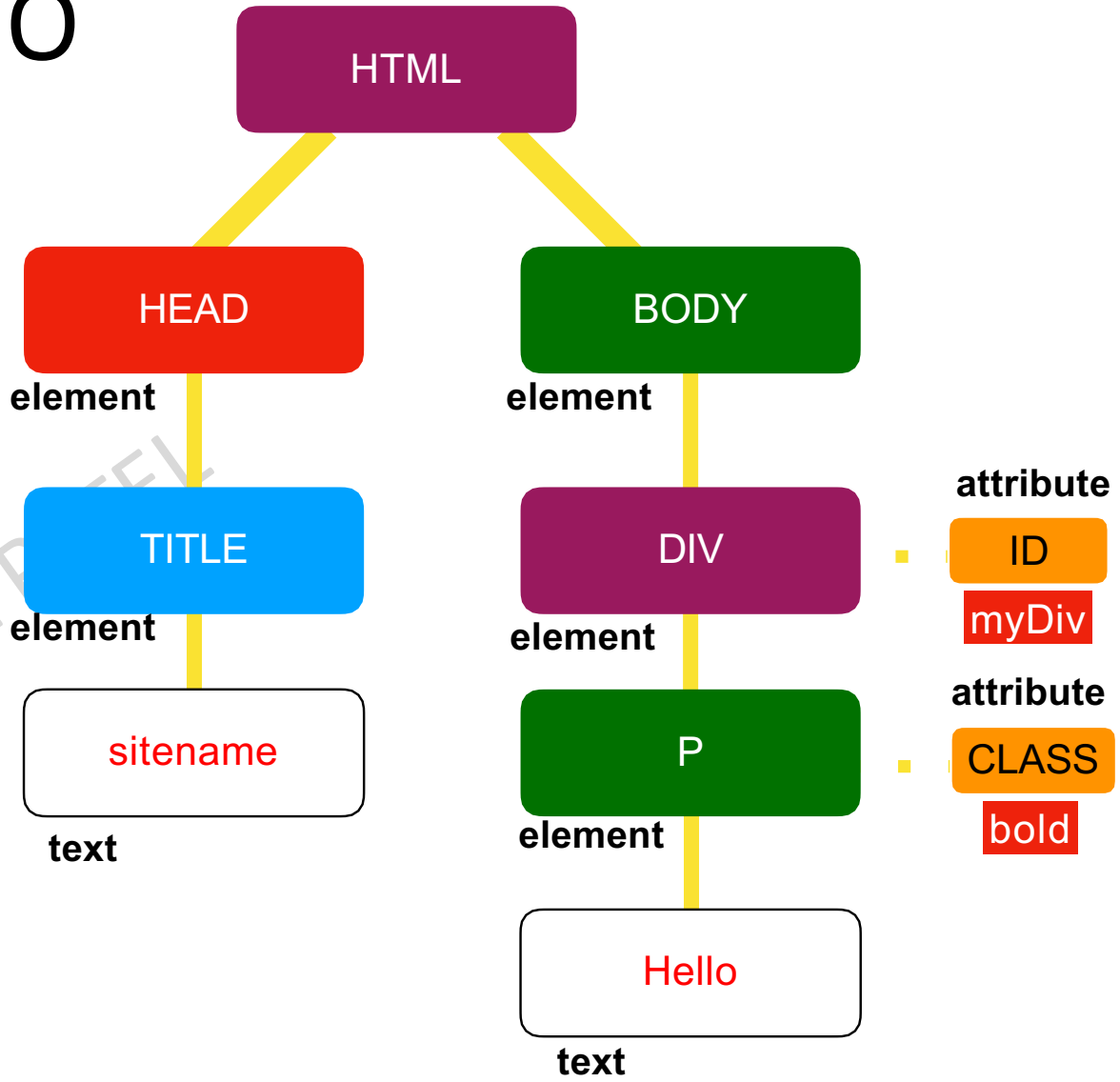
What DOM can do?

Change	JavaScript can change all the HTML elements in the page
Add	JavaScript can add New elements and attributes
React	JavaScript can react to all existing HTML events in the page
Change	JavaScript can change all the attributes in the page
Change	JavaScript can change all the CSS styles in the page
Remove	JavaScript can remove existing HTML elements and attributes
Create	JavaScript can create new HTML events in the page

DO M

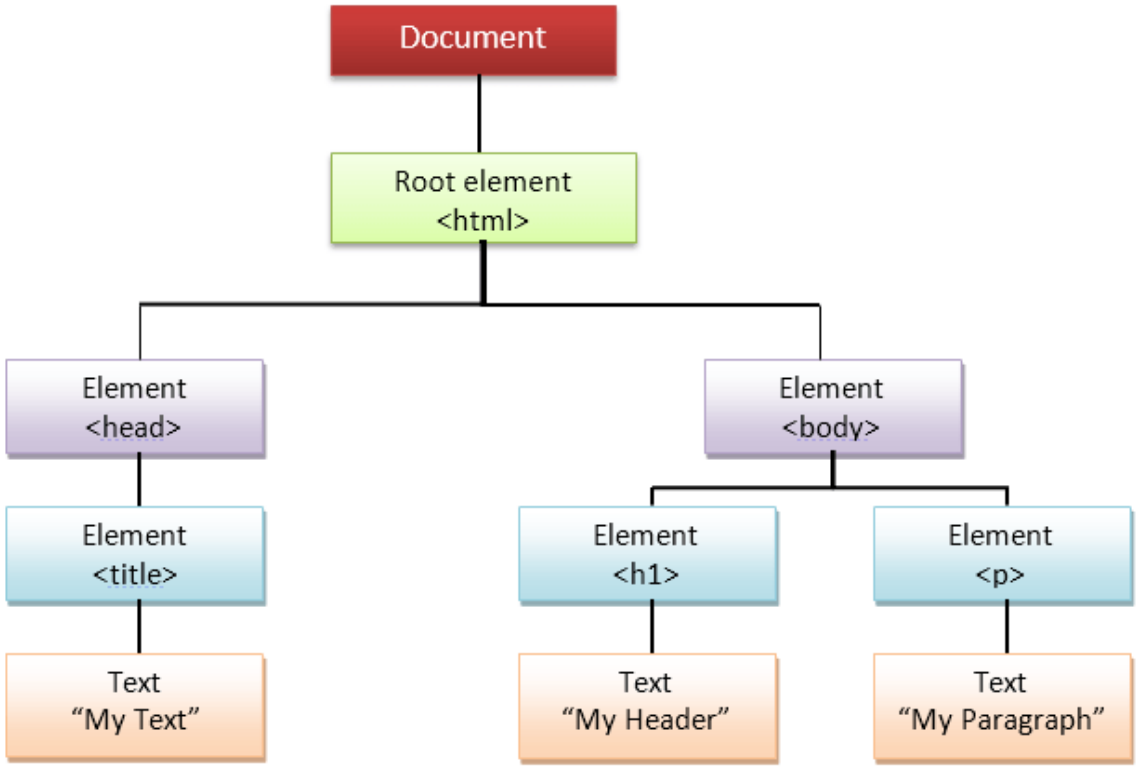
```
<html>
<head>
    <title>sitename<title>
</head>
 <body>
       <div id="myDiv">
           <p class="bold">
               Hello
            </p>
       </div>
 </body>
</html>
```

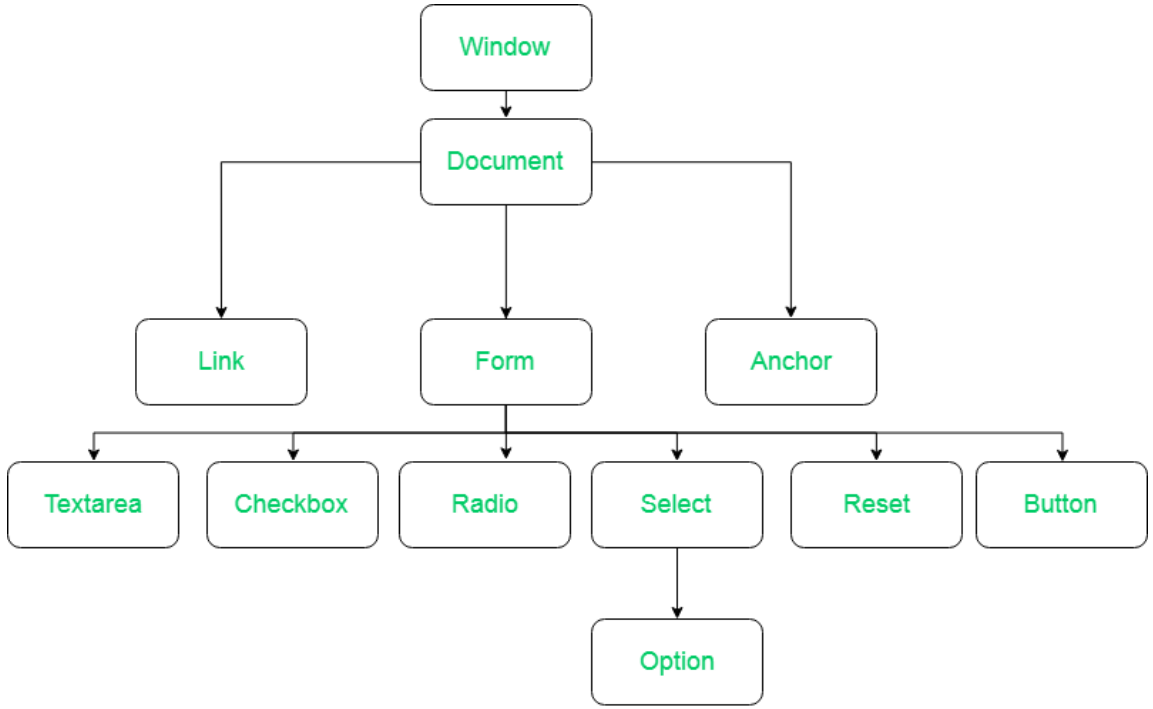
HTML



DOM TREE

JavaScript Document Object Model





DOM Object

- **DOM Properties**

Properties	Description
Cookie	It returns all name or value pairs of cookies in the document.
documentMode	It returns the mode used by the browser to render the document.
Domain	It returns the domain name of the server that loaded the document.
lastModified	It returns the date and time of last modified document.
readyState	It returns the status of the document.
Referrer	It returns the URL of the document that loaded the current document.
Title	It sets or returns the title of the document.
URL	It returns the full URL of the document.

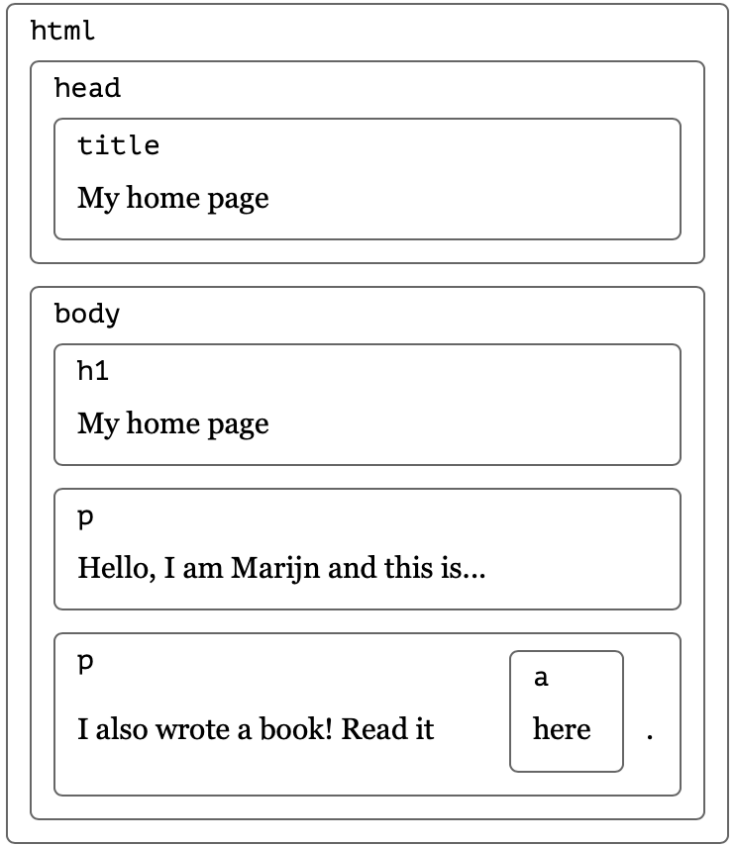
DOM Object

- **DOM Methods**

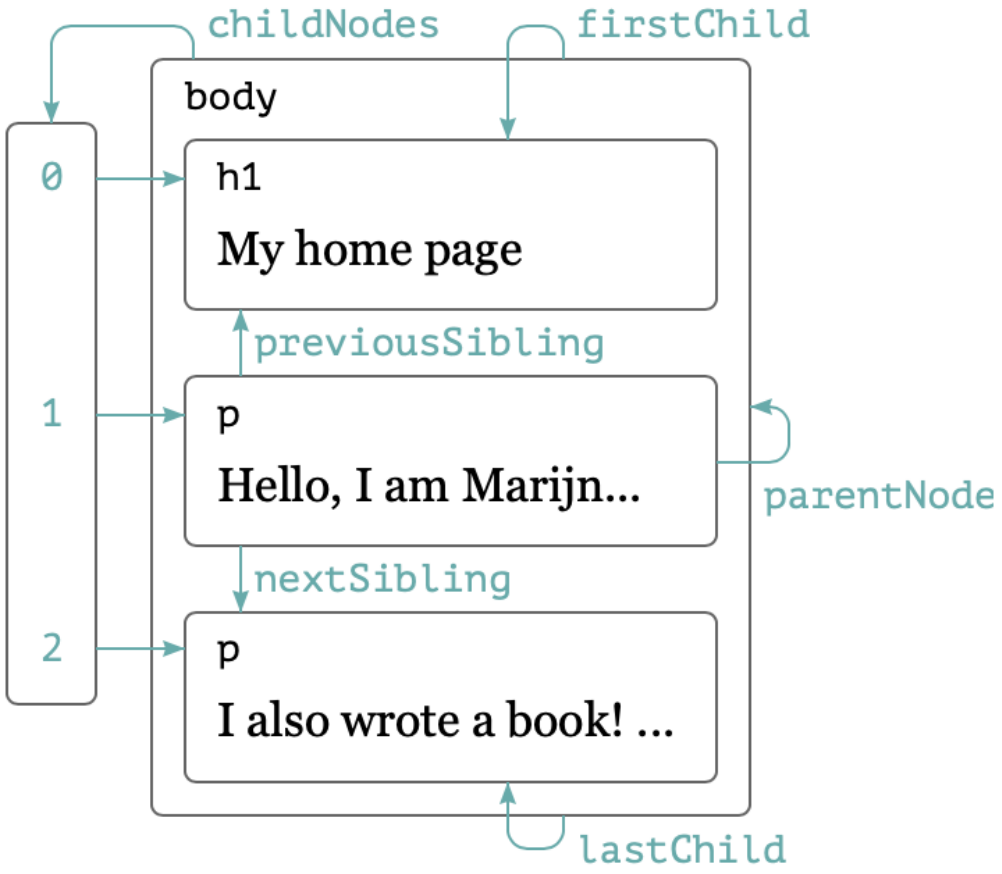
Methods	Description
close()	It closes the output stream previously opened with document.open().
clear()	It clears the document in a window.
getElementById()	It accesses the first element with the specified ID.
getElementByName()	It accesses all the elements with a specified name.
getElementByTagName()	It accesses all elements with a specified tagname.
open()	It opens an output stream to collect the output from document.write().
write()	It writes output (JavaScript code) to a document.
writeln()	Same as write(), but adds a newline character after each statement.

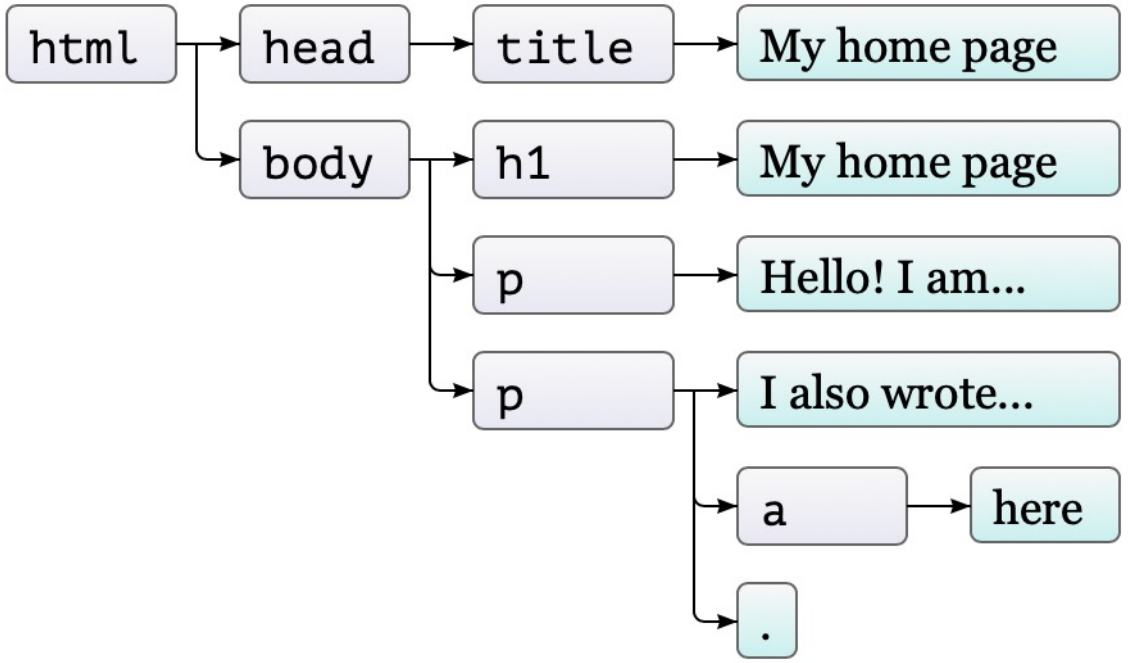
DOM Object

```
<!doctype html>
<html>
        <head>
                <title>My home page</title>
        </head>
        <body>
                <h1>My home page</h1>
                <p>Hello, I am Marijn and this
                        is my home page.</p>
                <p>I also wrote a book! Read it
                        <a
                href="http://eloquentjavascript.net">
                        here
                        </a>.
                </p>
        </body>
</html>
```



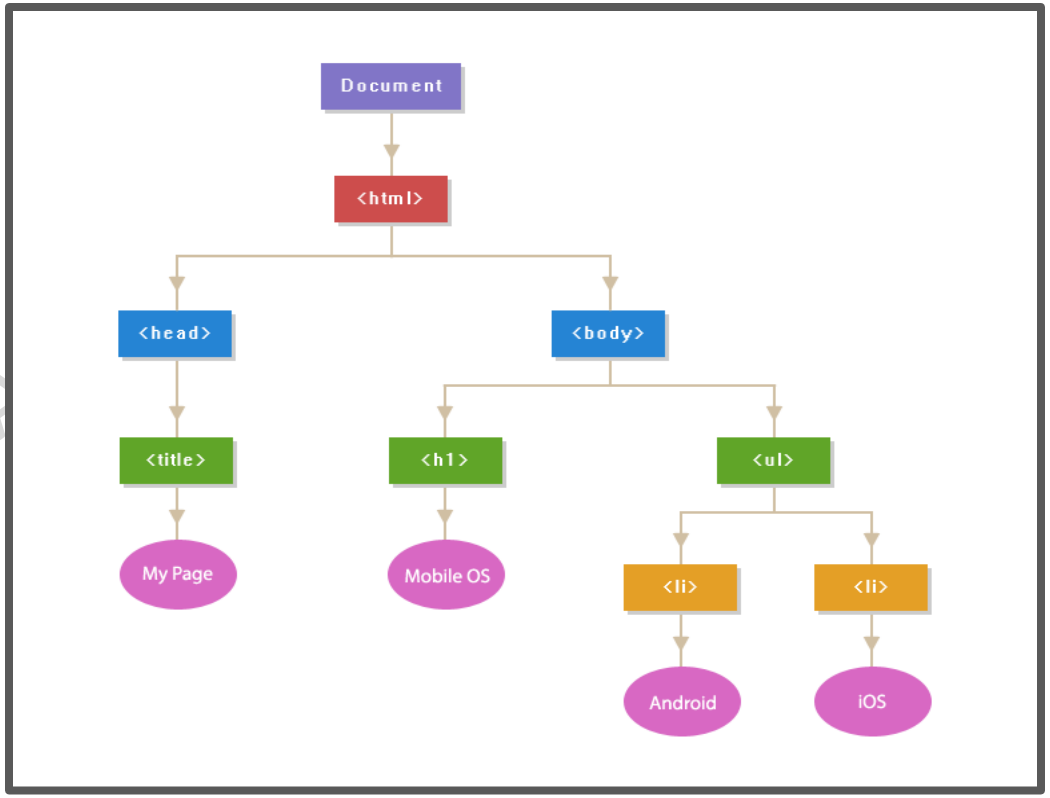
DOM Object


childNodes
firstChild
body
0
h1
My home page
previousSibling
1
p
Hello, I am Marijn...
parentNode
nextSibling
2
p
I also wrote a book! ...
lastChild


html
head
title
My home page
body
h1
My home page
p
Hello! I am...
p
I also wrote...
a
here
.

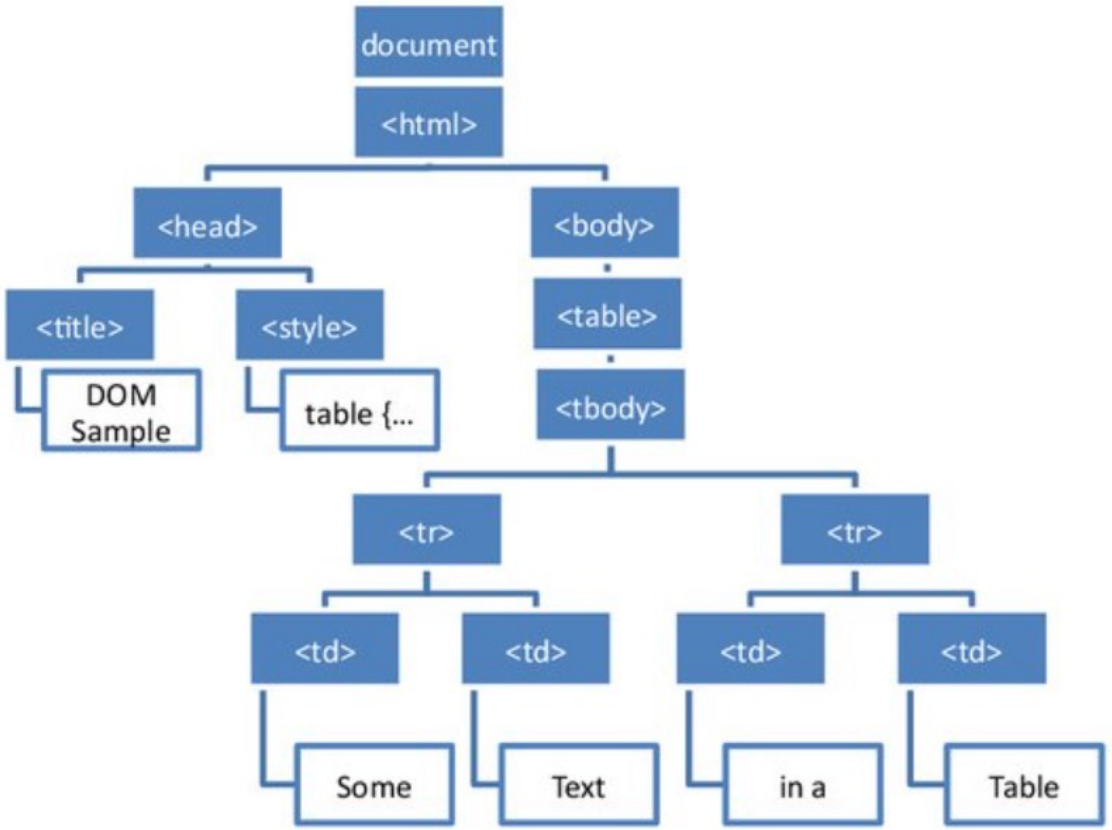
DOM Object

```
<!DOCTYPE html>
<html>
<head>
        <title>My Page</title>
</head>
<body>
        <h1>Mobile OS</h1>
        <ul>
                <li>Android</li>
                <li>iOS</li>
        </ul>
</body>
</html>
```

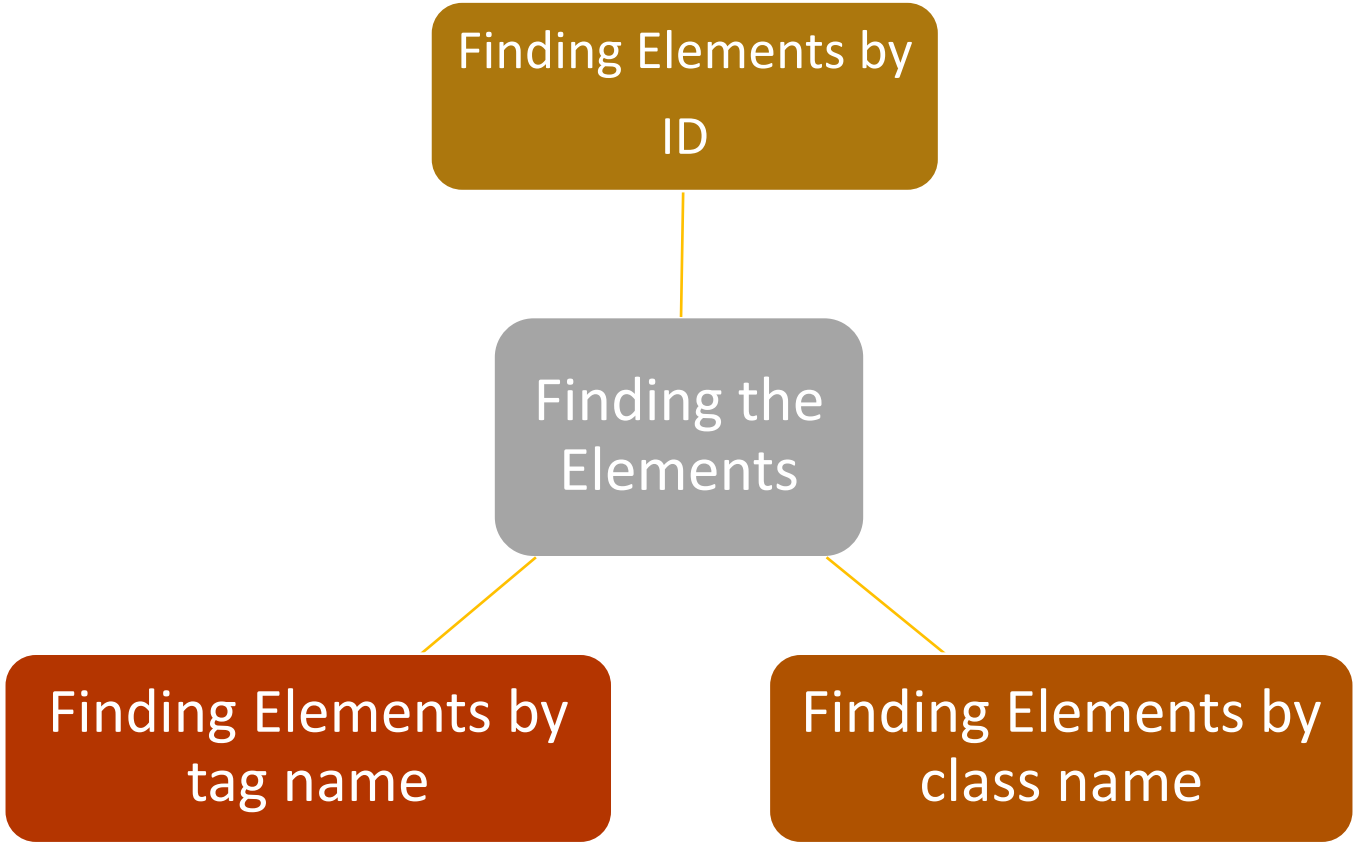


DOM Object

```
<!DOCTYPE html>
<html>
    <head>
        <title>DOM Sample</title>
        <style type="text/css">
            table {
                border: 1px solid black;
            }
        </style>
    </head>
    <body>
        <table>
            <tbody>
                <tr>
                    <td>Some</td>
                    <td>Text</td>
                </tr>
                <tr>
                    <td>in a</td>
                    <td>Table</td>
                </tr>
            </tbody>
        </table>
    </body>
</html>
```

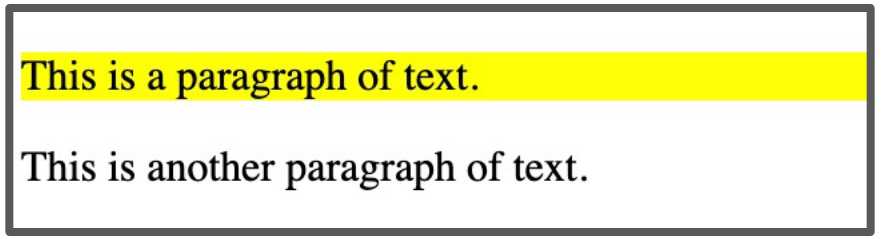


JavaScript DOM Selectors



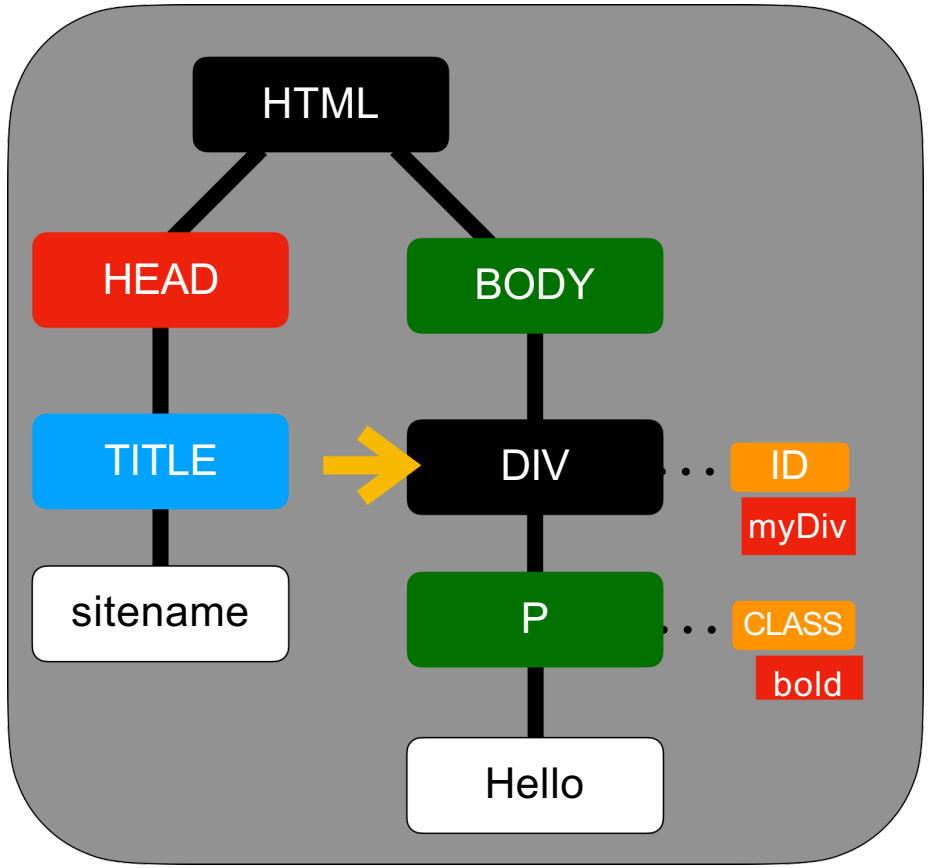
Finding HTML element by ID

```
<body>
        <p id="mark">This is a paragraph of text.</p>
        <p>This is another paragraph of text.</p>
        <script>
                // Selecting element with id mark
                var match = document.getElementById("mark");

                // Highlighting element's background
                match.style.background = "yellow";
        </script>
</body>
```

Select Elements

```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
      <div id="myDiv">
           <p class="bold">
            Hello </p>
      </div>
 </body>
</html>
```



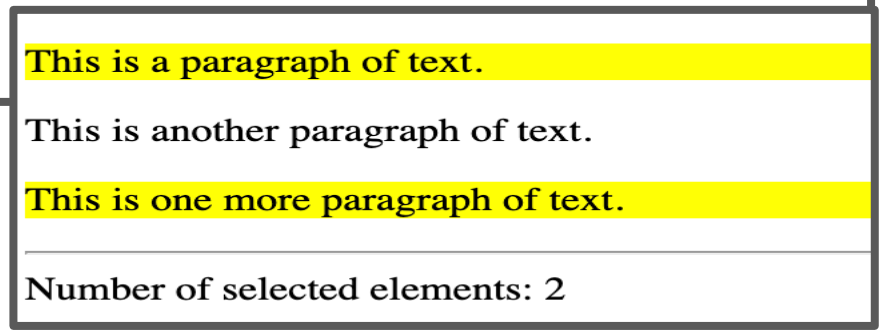
document

document .getElementById ("myDiv")

➡ **HTMLElement**

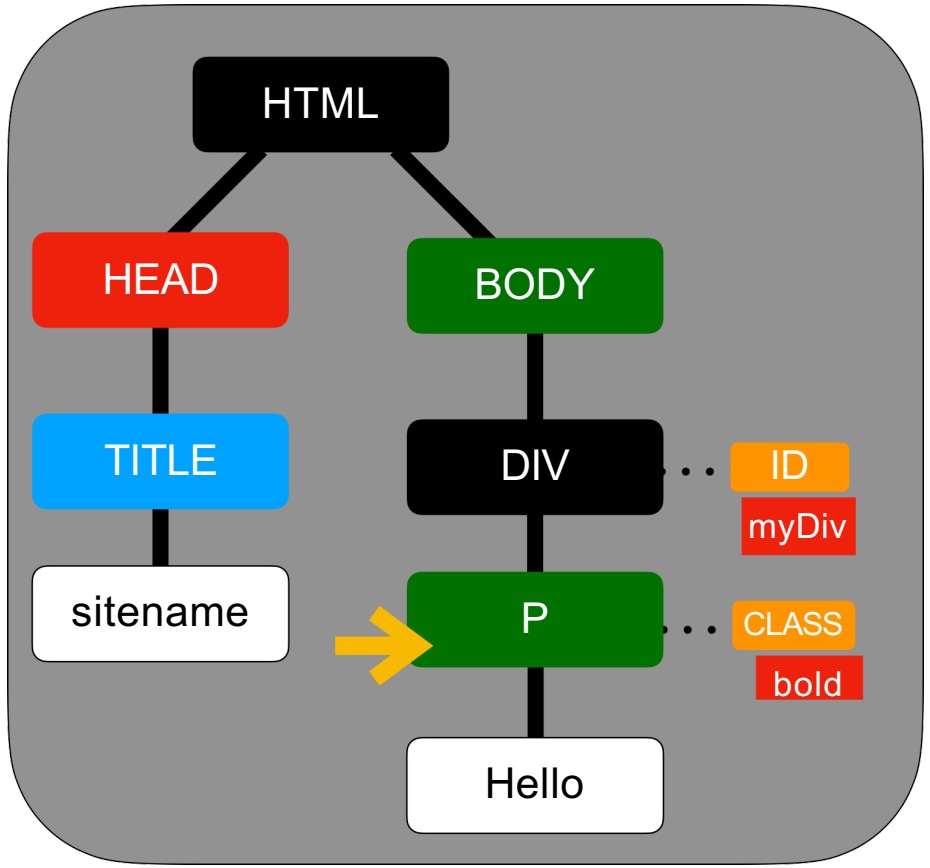
Finding HTML element by Tag Name

```
<body>
        <p>This is a paragraph of text.</p>
        <div class="test">This is another paragraph of text.</div>
        <p>This is one more paragraph of text.</p><hr>
        <script>
                // Selecting all paragraph elements
                let matches = document.getElementsByTagName("p");
                // Printing the number of selected paragraphs
                document.write("Number of selected elements: " + matches.length);
                // Highlighting each paragraph's background through loop
                for(let elem in matches)
                {
                        matches[elem].style.background = "yellow";
                }
        </script>
</body>
```

Select Elements

```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
      <div id="myDiv">
           <p class="bold">
            Hello </p>
      </div>
 </body>
</html>
```



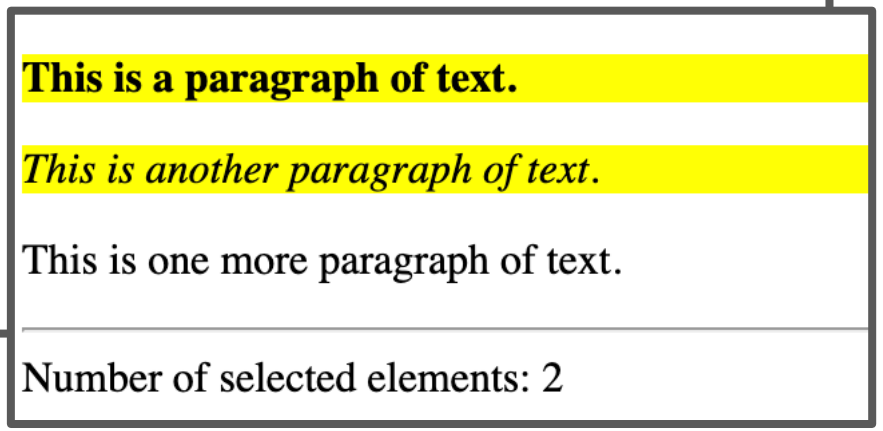
document

document .getElementsByTagName ("p")

HTMLCollection [p] *[HTMLelememt]*

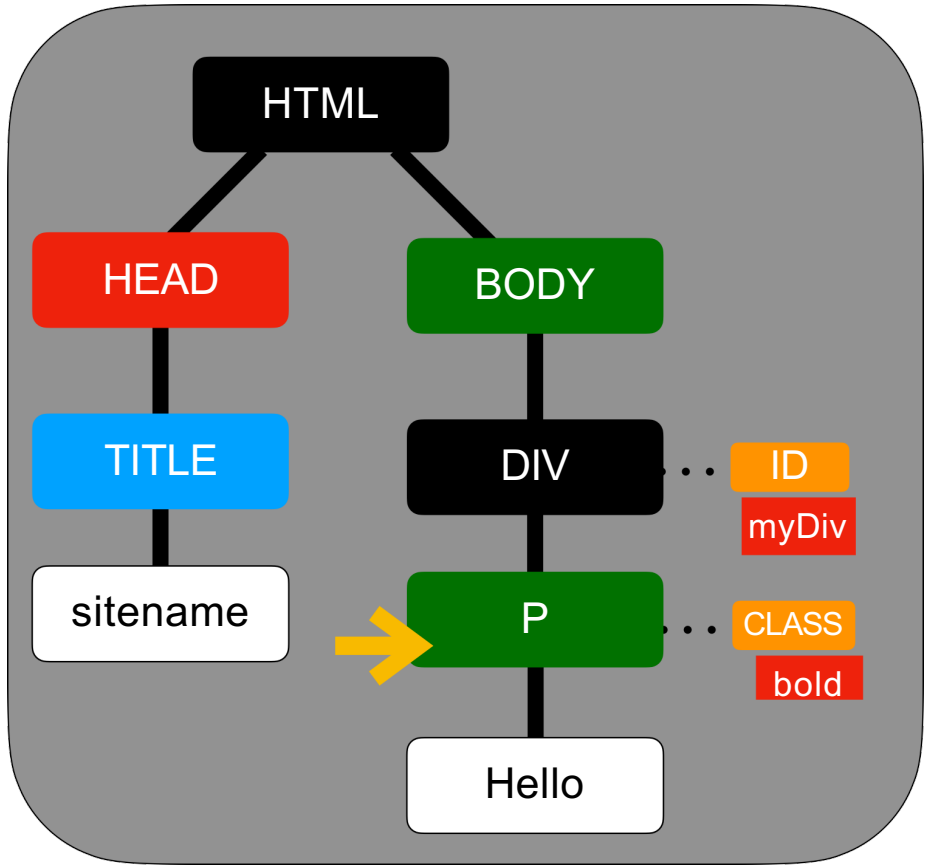
Finding HTML element by Class Name

```
<p class="test">This is a paragraph of text.</p>
<div class="block test">This is another paragraph of text.</div>
<p>This is one more paragraph of text.</p><hr>
<script>
        // Selecting elements with class test
        var matches = document.getElementsByClassName("test");
        // Displaying the selected elements count
        document.write("Number of selected elements: " + matches.length);
        // Applying bold style to first element in selection
        matches[0].style.fontWeight = "bold";
        // Applying italic style to last element in selection
        matches[matches.length - 1].style.fontStyle = "italic";
        // Highlighting each element's background through loop
        for(var elem in matches)
        {
                matches[elem].style.background = "yellow";
        }
</script>
```

Select Elements

```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
      <div id="myDiv">
           <p class="bold">
            Hello </p>
      </div>
 </body>
</html>
```



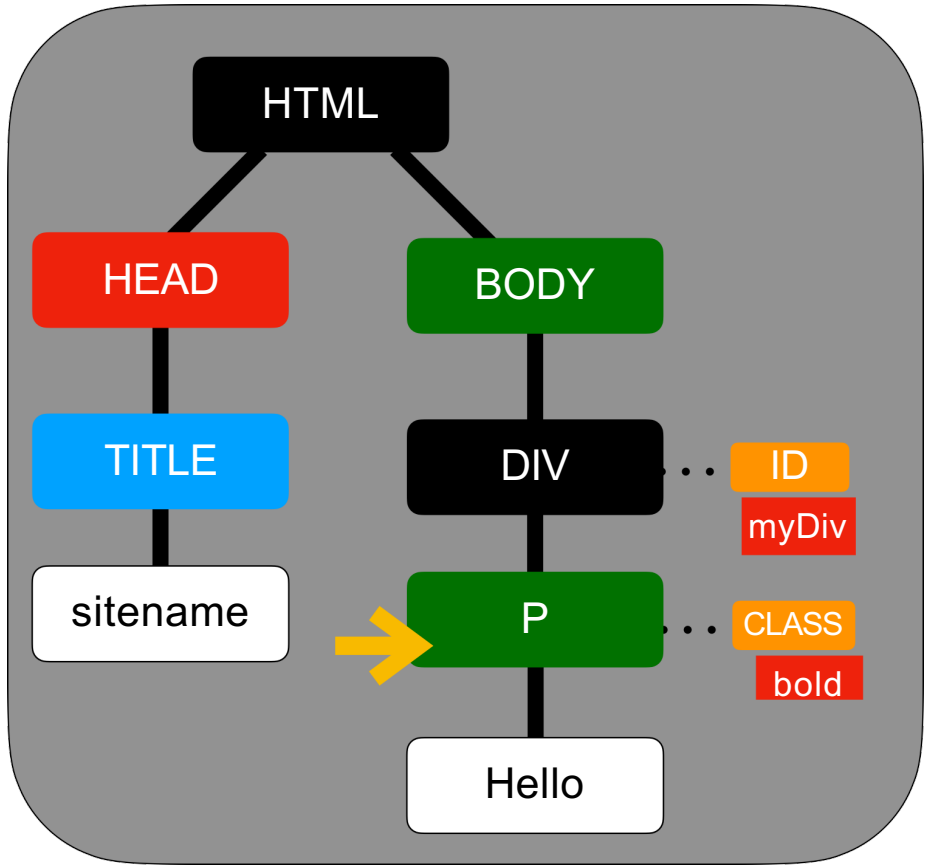
document

document .getElementsByClassName ("bold")

HTMLCollection [p] *[HTMLelememt]*

Select Elements

```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
      <div id="myDiv">
           <p class="bold">
            Hello </p>
      </div>
 </body>
</html>
```



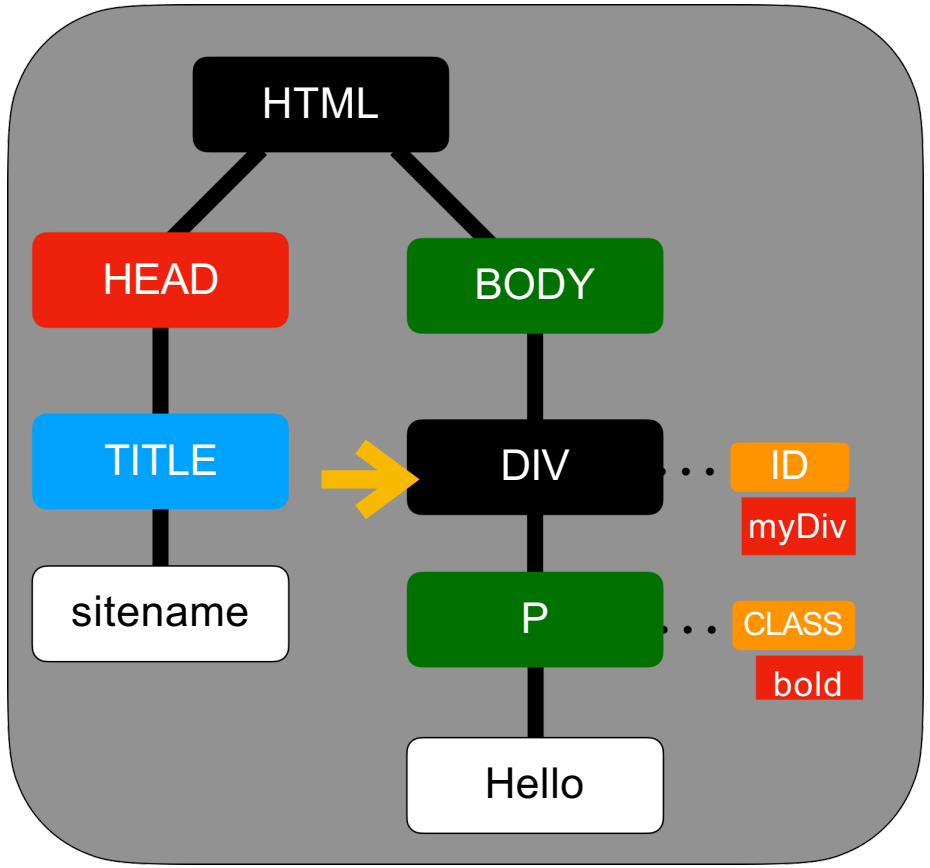
document

document .querySelector (".bold")

➡ **HTMLElement** ***JS Object***

Select Elements

```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
      <div id="myDiv">
           <p class="bold">
            Hello </p>
      </div>
 </body>
</html>
```



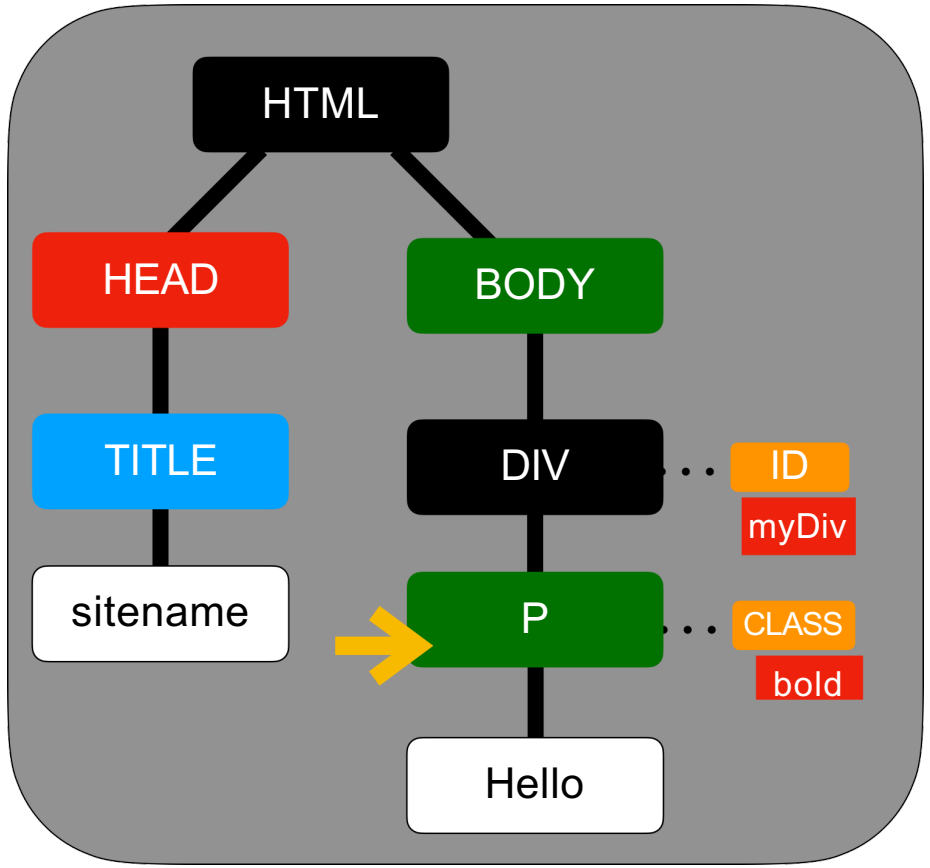
document

document .querySelector ("#myDiv")

➡ **HTMLElement**

Select Elements

```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
      <div id="myDiv">
           <p class="bold">
            Hello </p>
      </div>
 </body>
</html>
```



document

document .querySelectorAll (".bold")

NodeList [p] *[HTMLelememt]*

HTMLElement (reading)

const el = document .querySelector (“#myDiv”)

el.innerText → “”

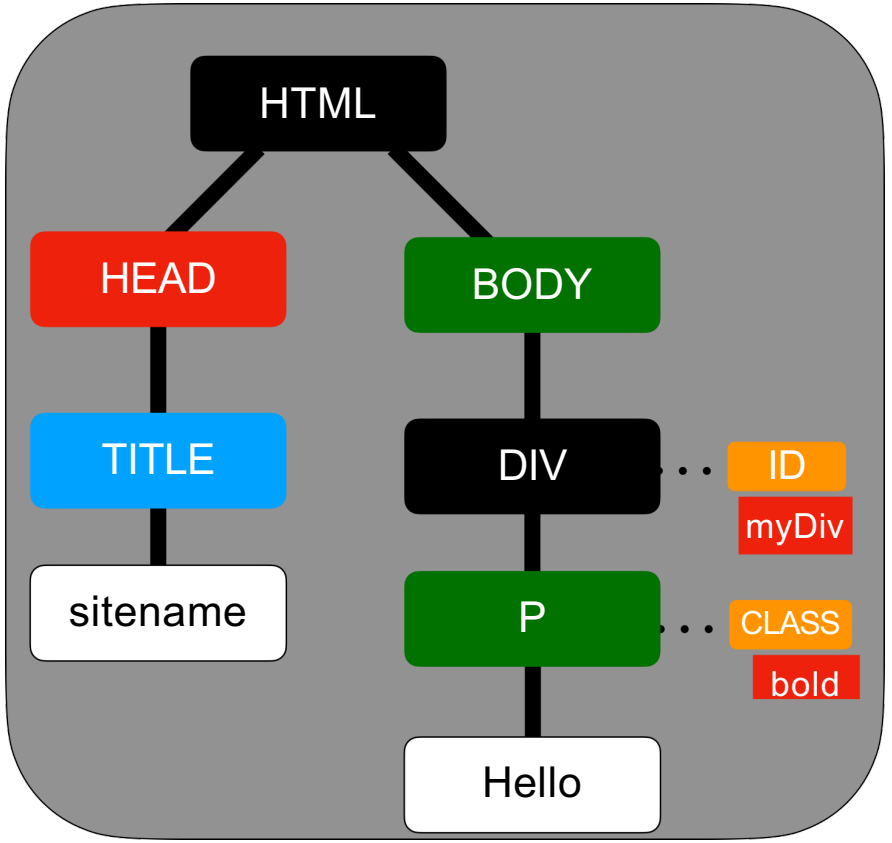
el.innerHTML → Hello

el.id → “myDiv”

const el = document .querySelector (“.bold”)

el.className → “bold”

el.innerText → “Hello”



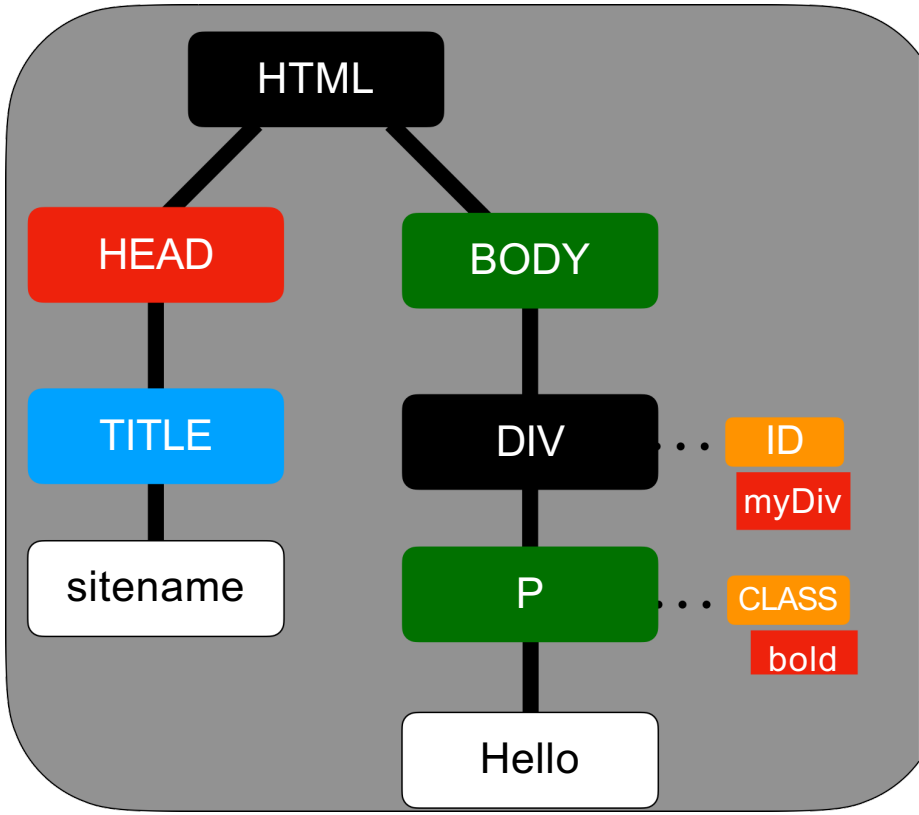

```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
      <div id=“myDiv”>
           <p class=“bold”>
          Hello </p>
      </div>
 </body>
</html>
```

HTMLElement (writing)

```
const el = document  querySelector ( “#myDiv” )
```

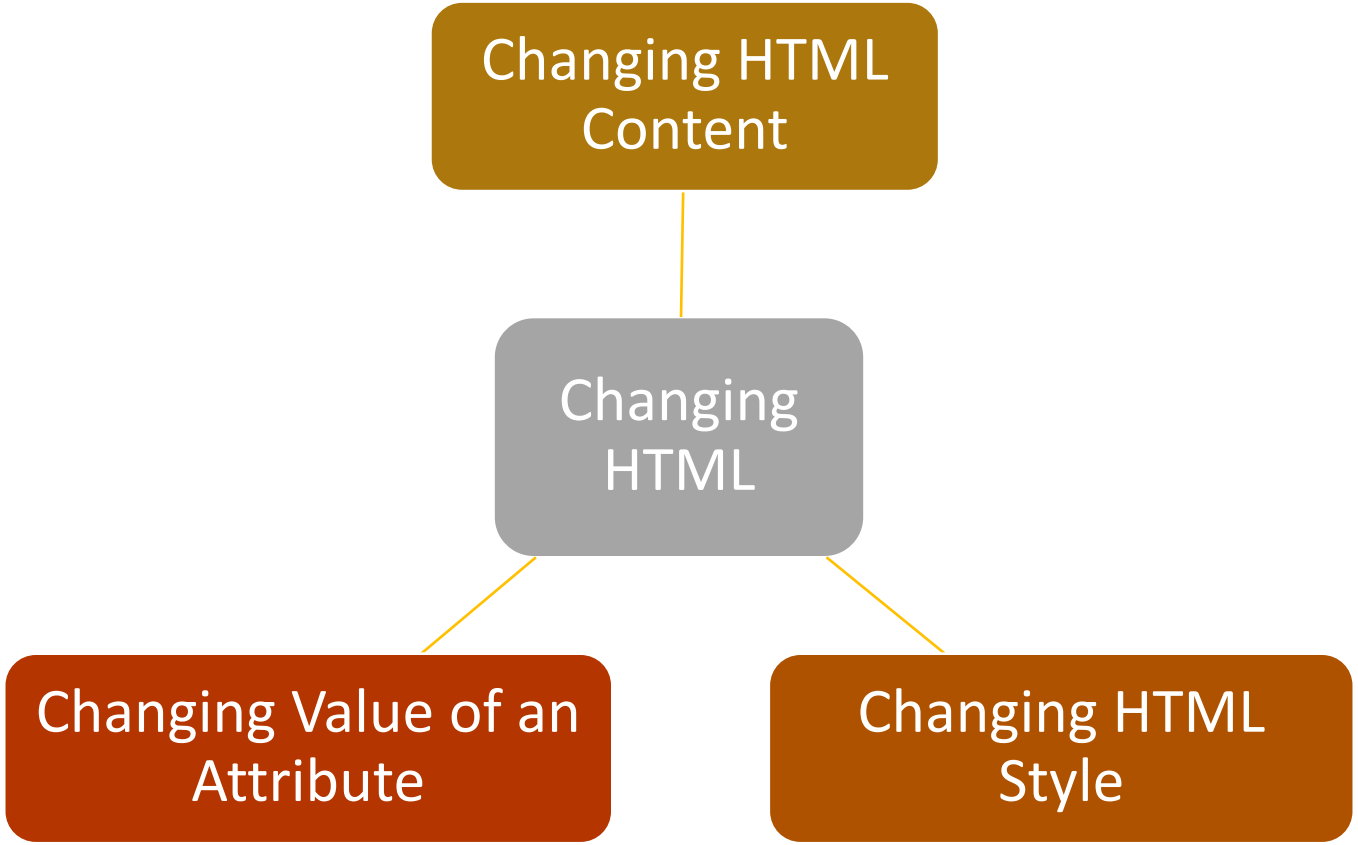
```
el.innerHTML  =  <p class=“bold”> Hey </p>
el.id         =  “myDiv”
```

```
const el = document .querySelector ( “.bold” )
el.className  =  “bold”
el.innerText  =  “Hello”
```



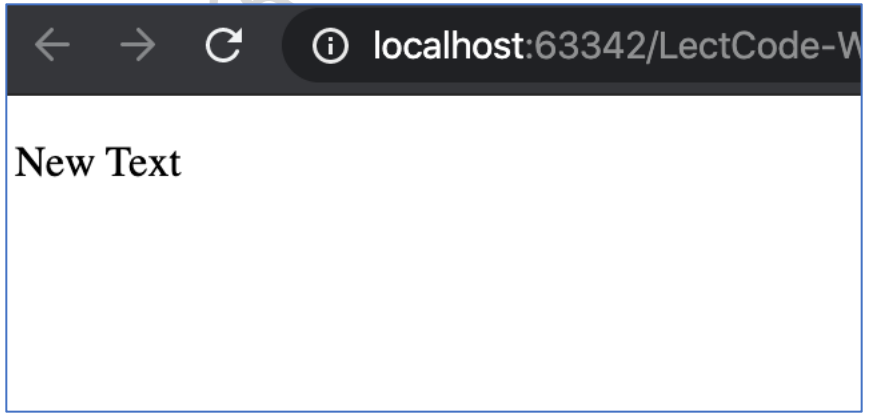

```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
      <div id=“myDiv”>
           <p class=“bold”>
          Hey </p>
      </div>
 </body>
</html>
```

JavaScript DOM Selectors



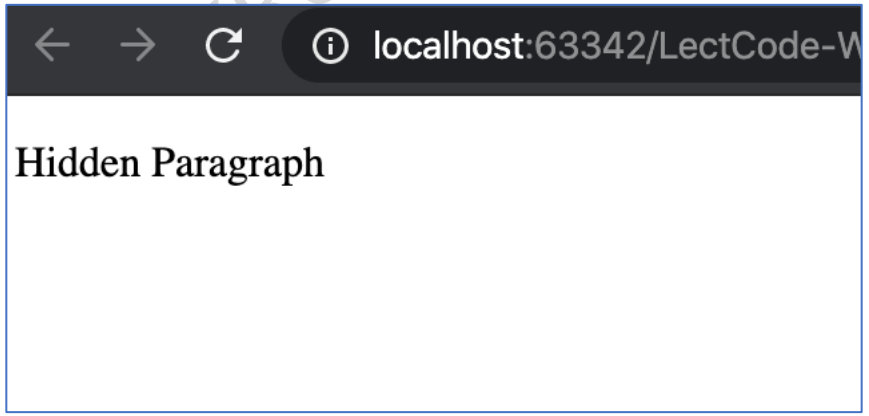
Changing HTML Content

```
<body>
        <p id="target">Old Text </p>
        <script>
                document.getElementById('target').innerHTML= "New Text"
        </script>
</body>
```

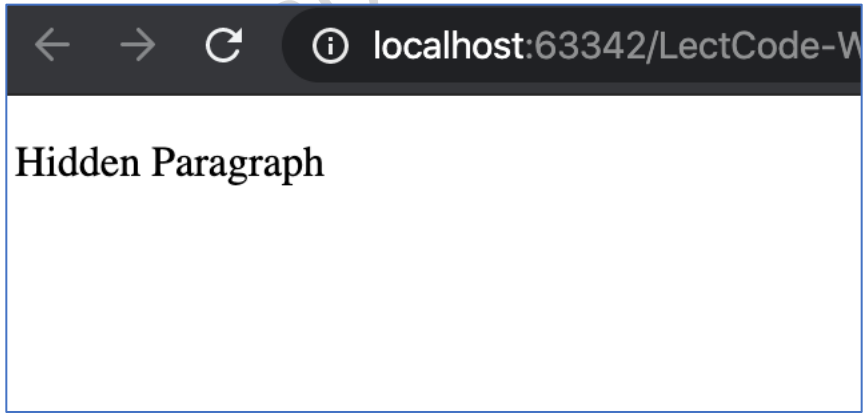
Changing Value of an Attribute

```
<body>
        <p id="target" hidden>Hidden Paragraph</p>
        <script>
                document.getElementById('target').hidden='';
        </script>
</body>
```



Changing HTML Style

```
<body>
        <p id="target" style="display: none">Hidden Paragraph</p>
        <script>
                document.getElementById('target').style.display='';
        </script>
</body>
```

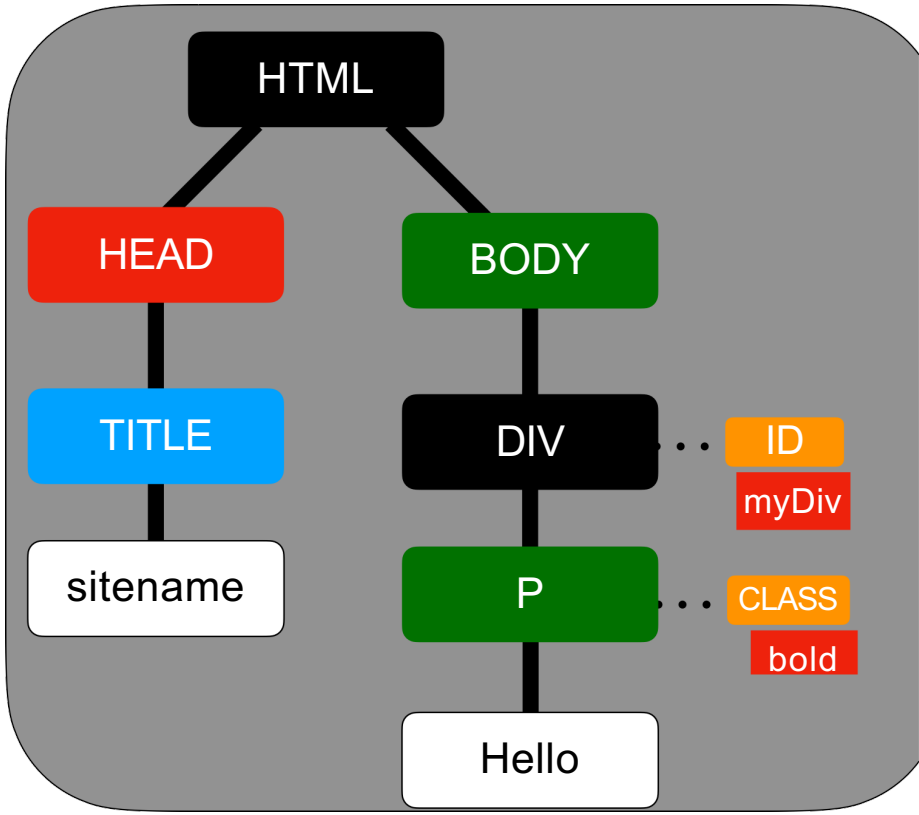



Attributes

```
const el = document .querySelector ( ".bold" )
```

el.getAttribute("class") → "bold"

el.setAttribute("class", "bold dark")

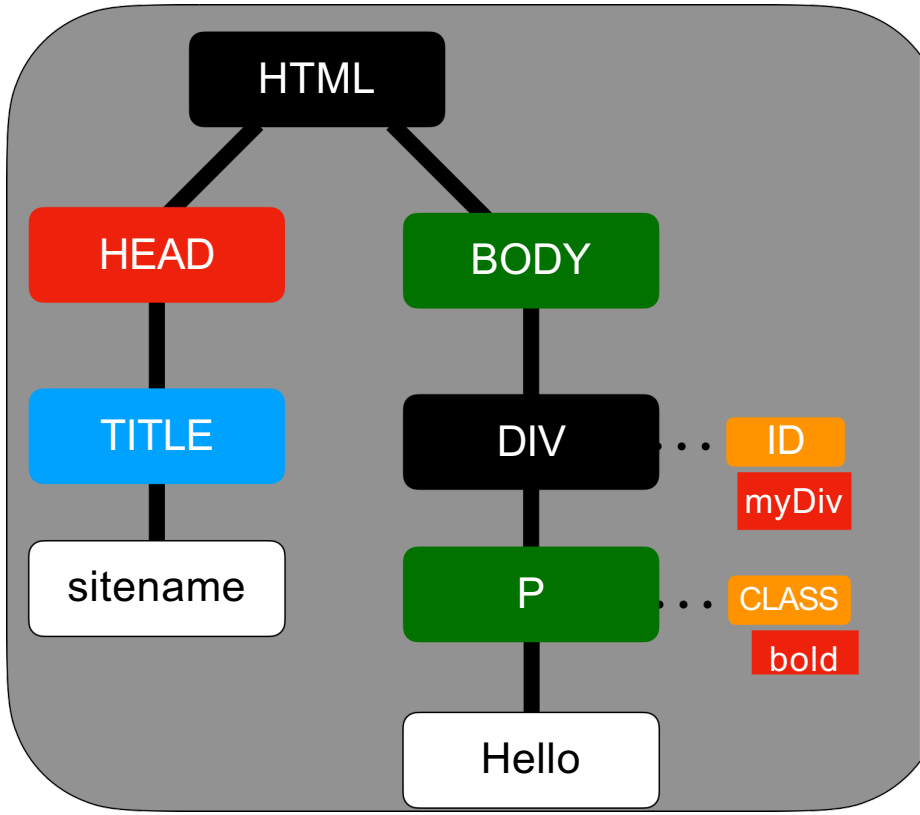



```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
      <div id="myDiv">
          <p class="bold">
         Hello </p>
      </div>
 </body>
</html>
```

CSS Styles

```
const el = document .querySelector ( ".bold" )
```

```
el.style.color  ➡  "black"
el.style.color  =  "blue"
el.style.backgroundColor  =  "yellow"
el.style.border  =  "1px solid red"
```




```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
      <div id="myDiv">
         <p class="bold">
        Hello </p>
      </div>
 </body>
</html>
```

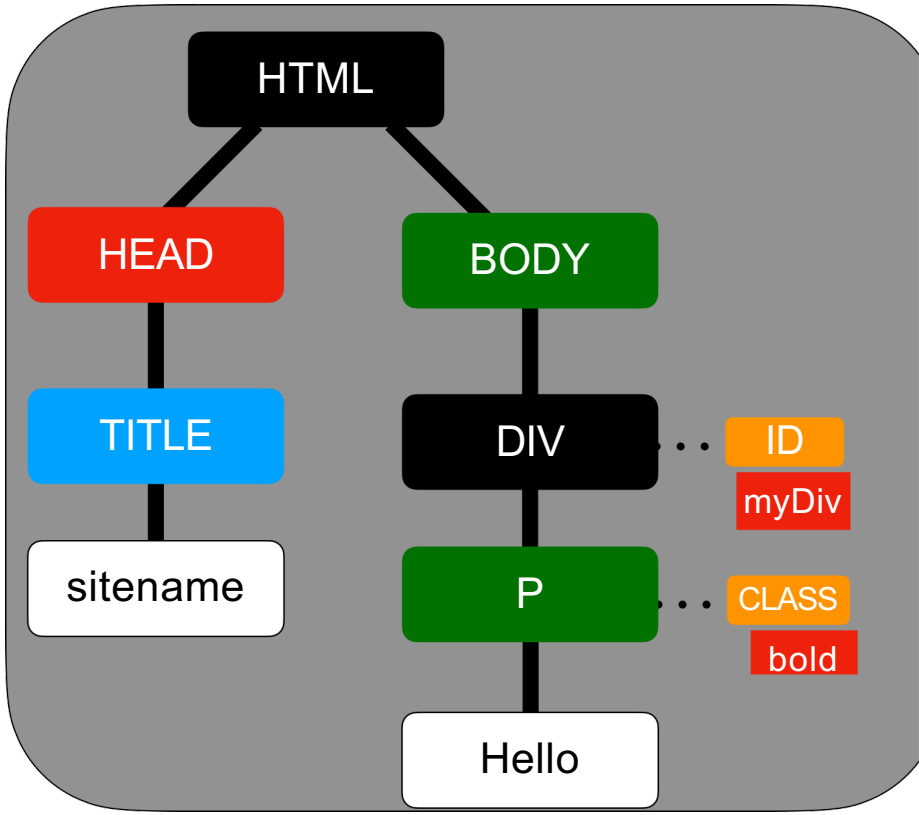
ClassList

```
const el = document .querySelector ( ".bold" )
```

```
el.classList.add("dark")
```

```
el.classList.remove("dark")
```

```
el.classList.replace("bold","dark")
```




```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
      <div id="myDiv">
        <p class="bold">
       Hello </p>
      </div>
 </body>
</html>
```

Children / Parent / Sibling

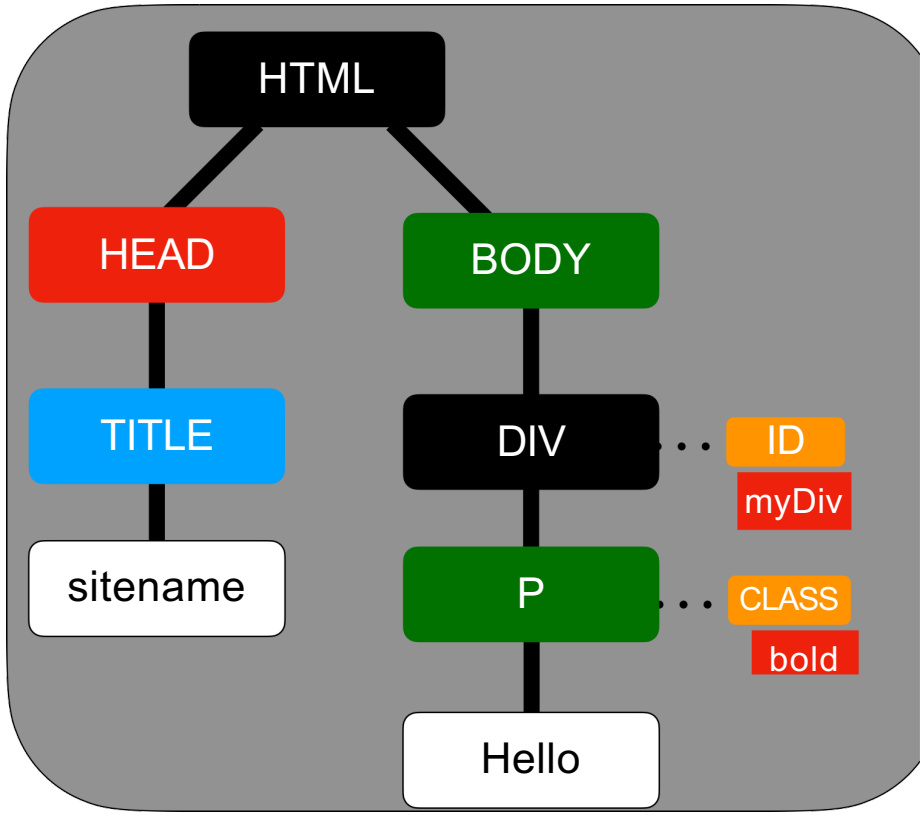
```
const el = document .querySelector ( “#myDiv” )
```

el.children → P

el.parentElement → Body

el.previousSibling → null

el.nextSibling → null




```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
      <div id=“myDiv”>
          <p class=“bold”>
        Hello </p>
</div>
</body>
</html>
```

Window Object

Window object is a **top-level object** in Client-Side JavaScript.

Window object represents the **browser's window.**

It represents an open window in a browser.

It supports all browsers.

The document object is a property of the window object. So, typing **window.document.write** is same as **document.write**.

All global variables are properties and functions are methods of the window object.

Window Object

window object properties:

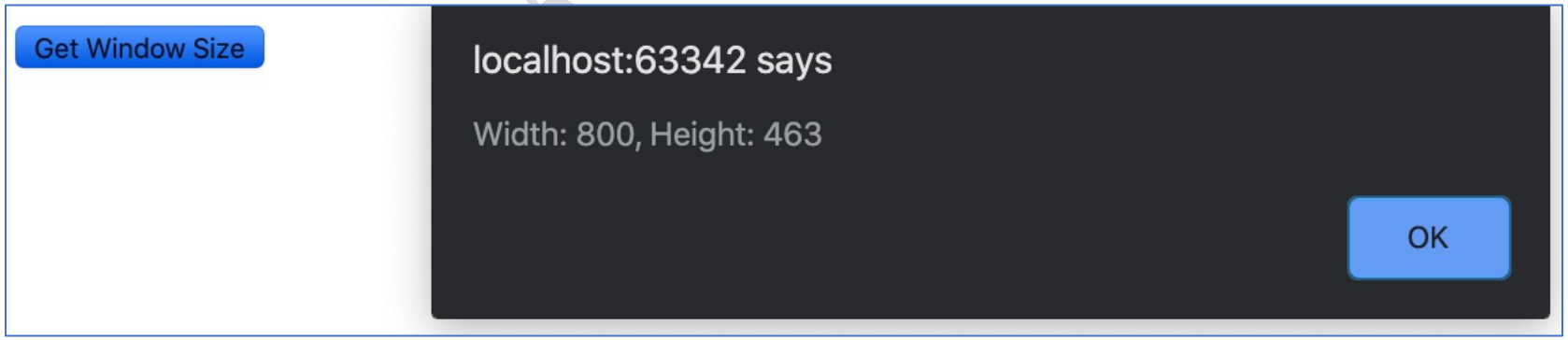
- **location** - Returns the current URL

window object methods:

- **alert()** - Displays some text in an alert box
- **prompt()** - Displays a prompt box that asks the user for some input

Window Object

```
<script>
        function windowSize()
        {
                const w = window.innerWidth;
                const h = window.innerHeight;
                alert("Width: " + w + ", " + "Height: " + h);
        }
</script>
<button type="button" onclick="windowSize();">Get Window Size</button>
```

Navigator Object

- **Navigator object** is the representation of Internet browser.
- It contains all the information about the visitor's (client) browser.
- **Navigator Object Properties**

Property	Description
appName	It returns the name of the browser.
appCodeName	It returns the code name of the browser.
appVersion	It returns the version information of the browser.
cookieEnabled	It determines whether cookies are enabled in the browser.
platform	It returns which platform the browser is compiled.
userAgent	It returns the user agent header sent by the browser to the server.

Navigator Object

- **Navigator Object Method**

Method	Description
javaEnabled()	It specifies whether or not the browser is Java enabled.

Navigator Object

```
<script type="text/javascript">

    document.write("<b>Browser: </b>"+navigator.appName+"<br><br>");

    document.write("<b>Browser Version: </b>"+navigator.appVersion+"<br><br>");

    document.write("<b>Browser Code: </b>"+navigator.appCodeName+"<br><br>");

    document.write("<b>Platform: </b>"+navigator.platform+"<br><br>");

    document.write("<b>Cookie Enabled: </b>"+navigator.cookieEnabled+"<br><br>");

    document.write("<b>User Agent: </b>"+navigator.userAgent+"<br><br>");

    document.write("<b>Java Enabled: </b>"+navigator.javaEnabled()+"<br><br>");

</script>
```

Navigator Object

Browser: Netscape

Browser Version: 5.0 (Macintosh; Intel Mac OS X 10_15_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.117 Safari/537.36

Browser Code: Mozilla

Platform: MacIntel

Cookie Enabled: true

User Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.117 Safari/537.36

Java Enabled: false

Document Object

- **document** - Represents an entire webpage and can be used to access the elements on it.
  - **document object properties:**
    - **domain** - Returns the domain in which a webpage is located
    - **lastModified** - Returns the date and time a webpage was last modified
  - **document object methods:**
    - **write()** - Writes text or content specified with HTML tags to a webpage

Document Object

```
<script type="text/javascript">

    document.write("The domain of this page is " + document.domain);
    document.write("<br />" + "This page was last modified on " + document.lastModified);

</script>
```

The domain of this page is localhost
This page was last modified on 01/13/2020 11:46:07

History Object

- **History object** is a part of the window object.
- It is accessed through the **window.history** property.
- It **contains the information of the URLs visited by the user within a browser window.**
- **History Object Properties**

Property	Description
Length	It returns the number of URLs in the history list.
Current	It returns the current document URL.
Next	It returns the URL of the next document in the History object.
Previous	It returns the URL of the last document in the history object.

History Object

- **History Object Methods**

Method	Description
back()	It loads the previous URL in the history list.
forward()	It loads the next URL in the history list.
go(“URL”)	It loads a specific URL from the history list.

Location Object

Location object is a part of the window object.

It is accessed through the '**window.location**' property.

It **contains the information about the current URL**.

Location Object

- **Location Object Properties**

Property	Description
hash	It returns the anchor portion of a URL.
host	It returns the hostname and port of a URL.
hostname	It returns the hostname of a URL.
href	It returns the entire URL.
pathname	It returns the path name of a URL.
port	It returns the port number the server uses for a URL.
protocol	It returns the protocol of a URL.
search	It returns the query portion of a URL.

Location Object

- **Location Object Methods**

Method	Description
assign()	It loads a new document.
reload()	It reloads the current document.
replace()	It replaces the current document with a new one.

Location Object

```
<script type="text/javascript">

    document.write("<b>Path Name: </b>"+location.pathname+"<br><br>");

    document.write("<b>Href: </b>"+location.href+"<br><br>");

    document.write("<b>Protocol: </b>"+location.protcol+"<br><br>");

</script>
```


Path Name: /LectCode-WTP-2019/JavaScript/LocationObject.html

Href: http://localhost:63342/LectCode-WTP-2019/JavaScript/LocationObject.html?_ijt=rr7mhtrfptrvrs1b4onhi5hi2e

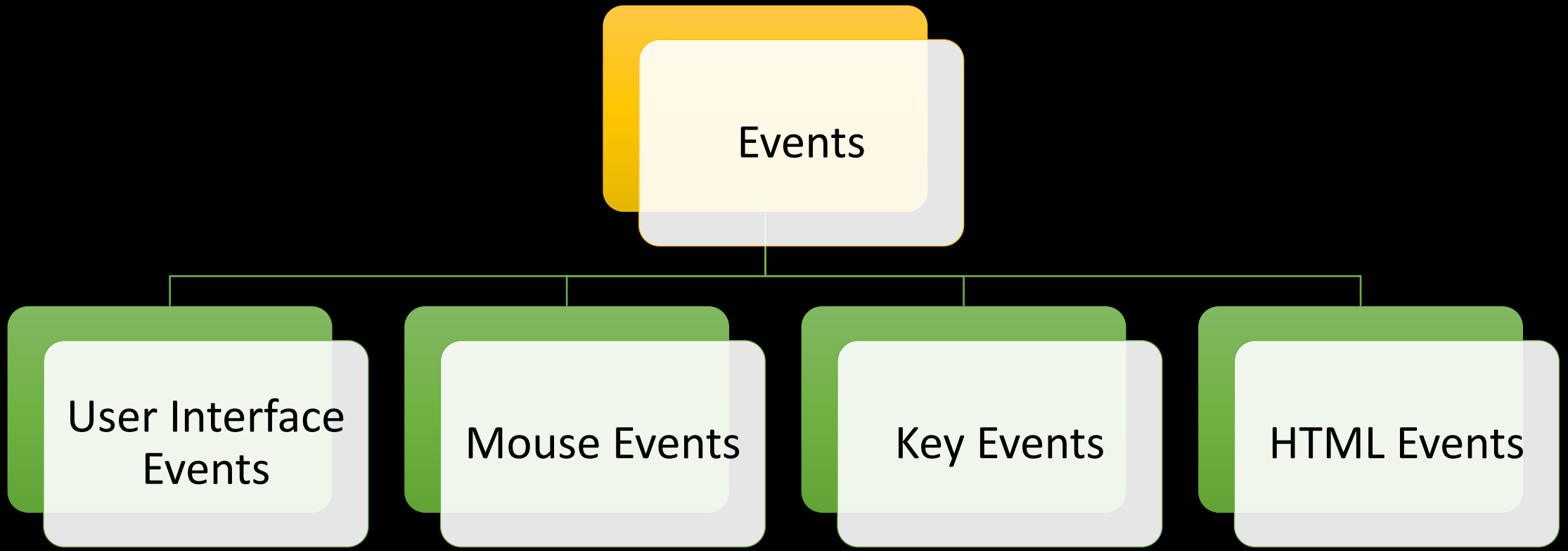
Protocol: undefined

Events

An **E*vent*** is defined as "***something that takes place***" and that is exactly what it means in web programming as well.

An **event handler** is JavaScript code that is designed to run each time a particular event occurs.

Syntax of handling the events <Tag Attributes **event="handler"**>


Events
User Interface Events
Mouse Events
Key Events
HTML Events

Categories of Events

User interface events happen as controls or other objects on a web page gain and lose focus. These events are often caused by other user actions such as a tab key press. They can also happen programmatically as well.

Mouse events occur when the user moves the mouse or presses one of the mouse buttons. These events allow a web page to respond to mouse movements by.

Key events occur when the user presses and/or releases one of the keyboard keys. Only certain HTML elements can capture keyboard events.

Finally, there are several events specific to certain **HTML** elements. They often relate to the browser window itself or to form controls and other objects embedded in a web page.

JavaScript Event Handling

Event Handling is a software routine that processes actions, such as keystrokes and mouse movements.

It is the receipt of an event at some event handler from an event producer and subsequent processes.

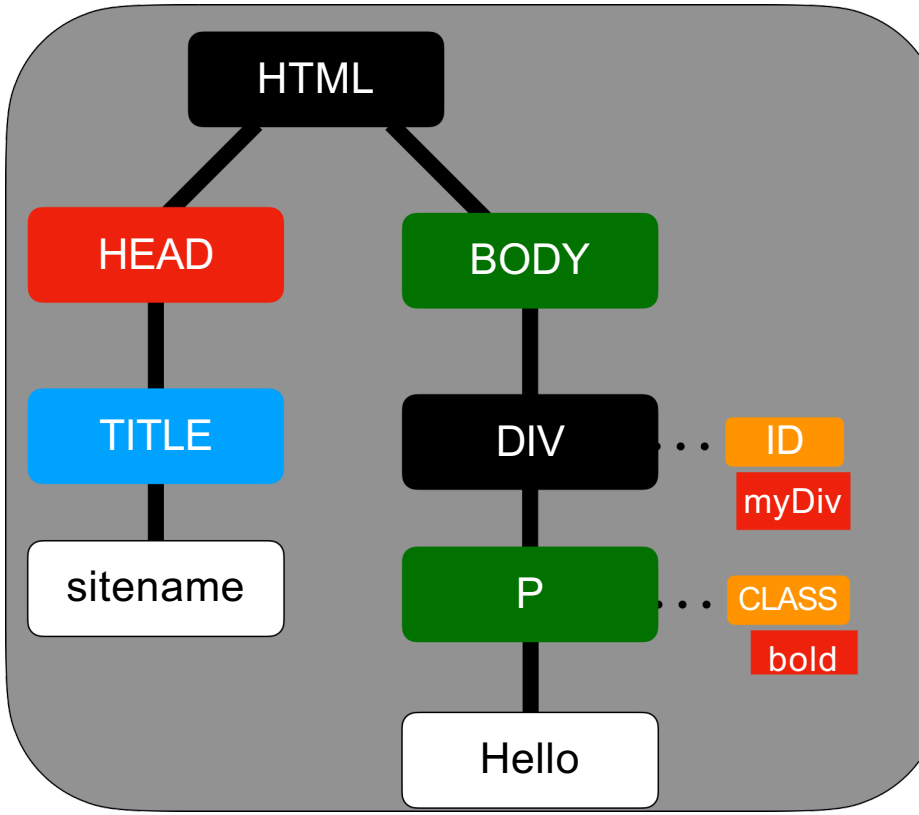
Functions of Event Handling

- Event Handling identifies where an event should be forwarded.
- It makes the forward event.
- It receives the forwarded event.
- It takes some kind of appropriate action in response, such as writing to a log, sending an error or recovery routine or sending a message.
- The event handler may ultimately forward the event to an event consumer.

Events

```
const el = document .querySelector ( ".bold" )

el.addEventListener( "click", function(){

} )
```

runs on every click




```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
       <div id="myDiv">
          <p class="bold">
         Hello </p>
       </div>
 </body>
</html>
```

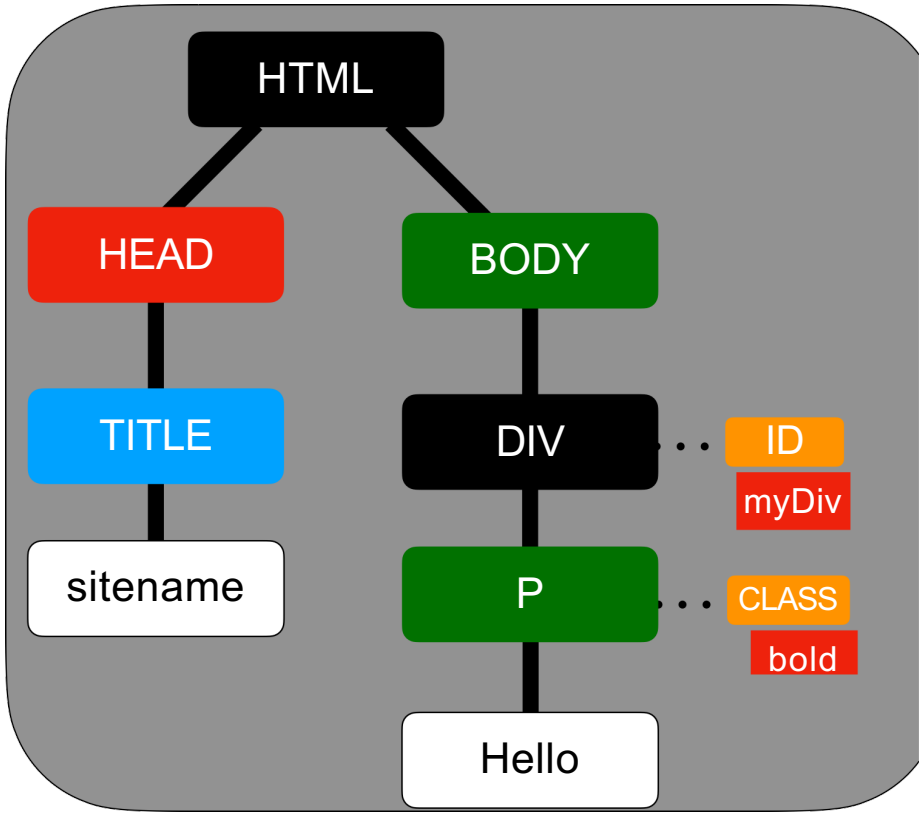
Event Object

```
const el = document .querySelector ( ".bold" )

el.addEventListener( "click", function(e){

} )
```

event Object

```
e.target.innerText
```

target = element




```
<html>
<head>
   <title>sitename<title>
</head>
 <body>
       <div id="myDiv">
          <p class="bold">
         Hello </p>
       </div>
 </body>
</html>
```

Event Handlers

Event Handler	Description
onAbort	It executes when the user aborts loading an image.
onBlur	It executes when the input focus leaves the field of a text, textarea or a select option.
onChange	It executes when the input focus exits the field after the user modifies its text.
onClick	In this, a function is called when an object in a button is clicked, a link is pushed, a checkbox is checked or an image map is selected. It can return false to cancel the action.
onError	It executes when an error occurs while loading a document or an image.
onFocus	It executes when input focus enters the field by tabbing in or by clicking but not selecting input from the field.
onLoad	It executes when a window or image finishes loading.

Event Handlers

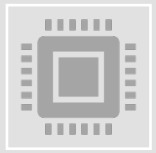
Event Handler	Description
onMouseOver	The JavaScript code is called when the mouse is placed over a specific link or an object.
onMouseOut	The JavaScript code is called when the mouse leaves a specific link or an object.
onReset	It executes when the user resets a form by clicking on the reset button.
onSelect	It executes when the user selects some of the text within a text or textarea field.
onSubmit	It calls when the form is submitted.
onUnload	It calls when a document is exited.

Form

JavaScript provides access to the forms within a HTML document through the **Form** object , which is a child of the **Document** object.

As with all **document** objects, the properties and methods of this object correspond to the various features and attributes of the HTML <**form**>.

Form



Processing of submitted information on the client side is advantage in terms of resources – by not sending the data over to the server

JavaScript can be used to validate data in HTML forms before sending off the content to a server.

Form data that typically are checked by a JavaScript could be:

has the user left required fields empty?

has the user entered a valid e-mail address?

has the user entered a valid date?

has the user entered text in a numeric field?

Properties of the Form object

Property	Description
action	Holds the value of the action attribute indicating the URL to send the form data to whe the user submits.
elements[]	Array of form fields objects representing the form field elements enclosed by this <form>.
method	The value of the method attribute of this <form> tag. Should be either GET or POST.
name	The name of the <form> as defined by its name attribute. You probably should also set the id attribute to hold the same value.
target	The name of the frame in which to display the page resulting from form submission.

Form Methods

- **Form**s also have two form-specific methods.
  - The **reset()** method clears the form's fields, like clicking a button defined by <**input type="reset" /**>
  - The **submit()** method triggers the submission of the form like clicking the button defined by <**input type="submit" /**>.
- In addition to the ability to trigger form reset and submission, we often want to react to these events as well
- So the <**form**> tag supports the corresponding **onreset** and **onsubmit** event handler attributes.
- As with all events, handler scripts can return a value of **false** to cancel the reset or submit.
- Returning **true** (or not returning a value) permits the event to occur normally.

Accessing Forms

- Before manipulation of form fields, we need to have the capable of accessing a **Form** fields.
- Forms can be accessed in three ways:
  - by **number** through **document.forms[].**
  - by **name** through **document.forms[].**
  - by the **regular element retrieval mechanisms** (e.g., **document.*formname***)

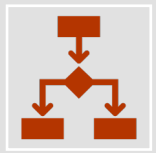
```
<form name="customerform" id="customerform" >
        <input type="text" name="firstname=" id="firstname=" /><br />
        <input type="text" name="lastname=" id="lastname=" /> ...more fields...
</form>
```

- we might use **window.document.forms[0]** (assuming it's the first form in the page), **window.document.forms['customerform']**, or **window.document.customerform**

Accessing Form Fields

Just as the **Document** contains a collection of <**form**> tags, each form contains a collection of form fields that can be accessed through the **elements[]** collection.

window.document.customerform.elements[0] refers to the first field.



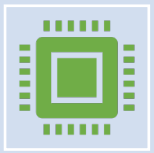
Similarly, we could access the fields by name,

window.document.customerform.elements["firstname"] or
window.document .customerform.firstname

Regular Expression

Regular Expressions, commonly known as "**regex**" or "**RegExp**", are a specially formatted text strings used to find patterns in text.

Regular expressions are one of the most powerful tools available today for **effective and efficient text processing and manipulations**.



For **example**, it can be used to verify whether the format of data i.e. name, email, phone number, etc. entered by the user is correct or not, find or replace matching string within text content, and so on.

Regular Expression

Function	What it Does
exec()	Search for a match in a string. It returns an array of information or null on mismatch.
test()	Test whether a string matches a pattern. It returns true or false.
search()	Search for a match within a string. It returns the index of the first match, or -1 if not found.
replace()	Search for a match in a string and replaces the matched substring with a replacement string.
match()	Search for a match in a string. It returns an array of information or null on mismatch.
split()	Splits up a string into an array of substrings using a regular expression.

Defining Regular Expressions

- In JavaScript, regular expressions are represented by **RegExp** object, which is a native JavaScript object like String, Array, and so on.
- There are two ways of creating a new **RegExp** object
  - **literal syntax**
  - **RegExp() constructor.**
- The literal syntax uses forward slashes (***/pattern/***) to wrap the regular expression pattern, whereas the constructor syntax uses quotes (***"pattern"***).

Defining Regular Expressions

```
// Literal syntax
var regex = /^Mr\./;

// Constructor syntax
var regex = new
RegExp("^Mr\\.");
```


- **Pattern Matching with Regular Expression**
- The characters that are given special meaning within a regular expression, are: `.*?+[](){}^$|\`
- Regular expression patterns include the use of letters, digits, punctuation marks

Character Classes

RegExp	What it Does
[abc]	Matches any one of the characters a, b, or c.
[^abc]	Matches any one character other than a, b, or c.
[a-z]	Matches any one character from lowercase a to lowercase z.
[A-Z]	Matches any one character from uppercase a to uppercase z.
[a-Z]	Matches any one character from lowercase a to uppercase Z.
[0-9]	Matches a single digit between 0 and 9.
[a-z0-9]	Matches a single character between a and z or between 0 and 9.

Predefined Character Classes

Shortcut	What it Does
.	Matches any single character except newline \n.
\d	matches any digit character. Same as [0-9]
\D	Matches any non-digit character. Same as [^0-9]
\s	Matches any whitespace character (space, tab, newline or carriage return character). Same as [\t\n\r]
\S	Matches any non-whitespace character. Same as [^ \t\n\r]
\w	Matches any word character (definned as a to z, A to Z,0 to 9, and the underscore). Same as [a-zA-Z_0-9]
\W	Matches any non-word character. Same as [^a-zA-Z_0-9]

Repetition Quantifiers

RegExp	What it Does
p+	Matches one or more occurrences of the letter p.
p*	Matches zero or more occurrences of the letter p.
p?	Matches zero or one occurrences of the letter p.
p{2}	Matches exactly two occurrences of the letter p.
p{2,3}	Matches at least two occurrences of the letter p, but not more than three occurrences.
p{2,}	Matches two or more occurrences of the letter p.
p{,3}	Matches at most three occurrences of the letter p

Position Anchors

RegExp	What it Does
^p	Matches the letter p at the beginning of a line.
p$	Matches the letter p at the end of a line.

Pattern Modifiers

Modifier	What it Does
g	Perform a global match i.e. finds all occurrences.
i	Makes the match case-insensitive manner.
m	Changes the behavior of ^ and $ to match against a newline boundary (i.e. start or end of each line within a multiline string), instead of a string boundary.
o	Evaluates the expression only once.
s	Changes the behavior of . (dot) to match all characters, including newlines.
x	Allows you to use whitespace and comments within a regular expression for clarity.

JSON

Stands for

JavaScript Object Notation

- **Lightweight data-interchange format**
  - **Simple *textual* representation of data**
- ***Easy for humans* to read and write**
- ***Easy for machines* to parse and generate**
- **Completely independent of any language**

JSON

- **JSON stands for JavaScript Object Notation.**
- **JSON** is extremely lightweight **data-interchange format for data exchange between server and client** which is quick and easy to parse and generate.
- Like XML, **JSON** is also a **text-based format** that's easy to write and easy to understand for both humans and computers, but unlike XML, **JSON data structures occupy less bandwidth than their XML versions.**
- JSON is based on two basic structures:
  - **Object**
  - **Array**

JavaScript Object Notation

JSON

When sending and receiving data in **IOT applications** and **APIs** you will encounter **JSON formatted data**.

Having a good working knowledge of **JSON**, and how to create and use **JSON** data will be very important in developing **IOT applications.**

Data from devices and sensors In IOT applications is normally sent in JSON format.

So, the application program on the sensor must package the data into a **JSON string**, and the receiving application must convert the JSON string into the original data format e.g., object, array etc .

JSON

Object: This is defined as an **unordered collection of key/value pairs (i.e. key : value).**

Each object **begins with a left curly bracket {** and **ends with a right curly bracket }.**

Multiple **key/value** pairs are separated by a comma **,**.

Array: This is defined as an **ordered list of values**.

An array **begins with a left bracket [** and **ends with a right bracket].**

Values are separated by a comma **,**.

JSON Data Types

- **String:** This is defined as an **unordered collection of key/value pairs (i.e. key : value).**

```
{"name":"Admin"}
```

- **Array:** This is defined as an **ordered list of values**.
  - An array **begins with a left bracket [** and **ends with a right bracket].**
  - Values are separated by a comma **,**.

```
{
"user":["Guest","Admin","Root"]
}
```

JSON Data Types

- **Number:**

```
{"age":30}
```

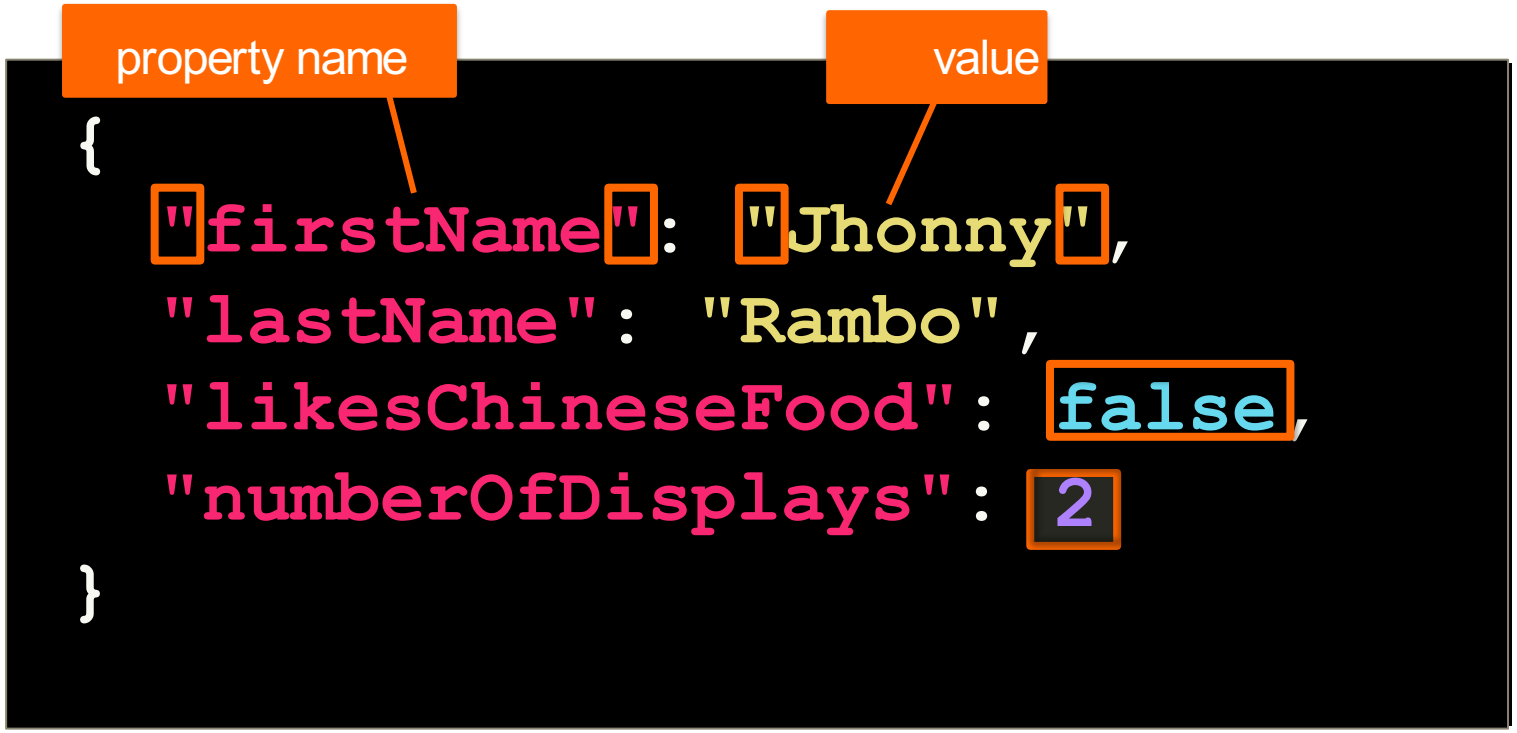
- **Object:**

```
{
    "user" : {
        "name" : "Admin",
        "Age" : 30,
        "City" : "Anand"
    }
}
```

Example

```
{
  "student": [
          {
              "id" : 1,
              "Name" : "Anand",
              "Branch" : "IT"
          },
          {
              "id" : 2,
              "Name" : "Vivek",
              "Branch" : "CP"
          }
      ]
}
```

JSON Example



JSON Example

```
var jsonString =
'{
   "firstName":"Jhonny",
   "lastName": "Rambo",
   "likesChineseFood": false,
   "numberOfDisplays": 2
}';
```