

## UNIT-II: JAVASCRIPT

**The Basic of Java script:** Objects, Primitives, Operations and Expressions, Screen Output and Keyboard Input, Control Statements, Object Creation and Modification, Arrays, Functions, Constructors, Pattern Matching using Regular Expressions

**DHTML:** Positioning Moving and Changing Elements

---

### ORIGINS:

JavaScript was originally developed at Netscape by Brendan Eich. Initially named Mocha but soon after was renamed LiveScript. In late 1995, LiveScript became a joint venture of Netscape and Sun Microsystems and its name changed again to JavaScript. A language standard for JavaScript was developed by European Computer Manufacturers Association (ECMA) as ECMA – 262. This standard is now in Version 6. Since 2015 new versions are named by year (ECMAScript 2016 / 2017 / 2018). The official name of this standard language is ECMA Script, but it becomes popular with the name “JavaScript”.

JavaScript can be divided into three parts:

#### 1. The Core

The core is the heart of the language includes its operators, expressions, statements and sub – programs.

#### 2. Client Side

Client – Side script is a collection of objects that supports the control of a browser and interactions with users. For example, with JavaScript, an HTML document can respond to the user inputs such as mouse clicks and keyboard events.

#### 3. Server Side

Server – Side Script is a collection of objects that makes language useful on a web server. For example, to communicate with database management system

### USES OF JAVASCRIPT:

- ☐ The original goal of JavaScript was to provide the programming capability at both server and client ends of web connection.
- ☐ Client Side Script can serve as an alternative to perform the tasks done at the server – side. It reduces computational tasks on server side. But it cannot replace all server side computing which include file operations, database access, networking, etc.
- ☐ Interaction with the users through form elements such as buttons and menus can be conveniently performed with the use of JavaScript.
- ☐ JavaScript is event driven; it is meant that an HTML document with embedded JavaScript is capable to respond to the user actions such as mouse clicks, button presses, keyboard strokes, etc.

- ☐ DOM is the main capability of JavaScript that makes static HTML Documents as highly dynamic, which allows accessing and modifying the style properties and content of the elements of HTML document.
- ☐ JavaScript can be used to create cookies.
- ☐ JavaScript is used to validate the data on the web page before submitting it to the server.

## DIFFERENCES BETWEEN JAVA & JAVASCRIPT

Java	Java script
1. Java is a Programming language	1. Java script is a scripting language
2. Java is an object-oriented language	2. Java script is an object-based language
3. Objects in Java are Static. i.e., the number of data members and method are fixed at compile time.	3. In JavaScript the objects are dynamic. i.e., we can change the total data members and method of an object during execution.
4. Java is strongly typed language, and type checking is done at compile time.	4. Java script is loosely typed language. And type checking is done at run time.
5. Difficult to understand	5. Java script is comparatively easier to understand

## HTML – JAVASCRIPT DOCUMENTS

If an HTML document does not include embedded scripts, the browser reads the lines of the document and renders its window according to the tags, attributes and the content it finds. When a JavaScript script is encountered in the document then the browser uses its JavaScript interpreter to “execute” the script.

There are two ways to embed the script in an HTML document, either implicitly or explicitly. In explicit embedding, the JavaScript code physically resides in the HTML document. We can embed the JavaScript code as follows:

```
<script type= “text/javascript”>
-----
-----
-----
</script>
```

In implicit embedding, the JavaScript code can be placed in a separate **.js** file, and linked to the HTML document. This can be done as follows:

```
<script type= “text/javascript” src= “Mypage.js”>
</script>
```

Implicit embedding has the advantage of hiding the script from the browser user. When scripts are embedded in HTML Document, they can appear in either head or body part of the HTML document depending on the purpose. Scripts that produce the content only when as a response to the user actions are placed in head section of the document. Scripts that are to be interpreted just once, only when the interpreter finds are placed in the body section of the document.

## **PRIMITIVE DATA TYPES**

Java script has five primitive types: Number, String, Boolean, Undefined and Null. Undefined and Null are often called trivial types.

- All numerical literals are primitive values of type Number. The Number type values are represented internally in double – precision floating point form. Numerical Literals in a script can be integers or floating point values.
- A string literal is a sequence of characters delimited by either single quotes or double quotes. String literals can also include escape characters such as \t, \n,.....
- The only values of type Boolean are true and false. These values are usually computed as the result of evaluating a relational or Boolean expression.
- If a variable has been explicitly declared but not assigned a value. It has value **undefined**.
- Null indicates no value. A variable is null if it has not been explicitly declared. If an attempt is made to use a variable whose value is null, it causes a runtime error.

## **DECLARING VARIABLES**

In JavaScript, variables are dynamic typed this means that the variable can be used for any type. Variables are not typed. A variable can have the value of any primitive type or it can be a reference to any object.

The type of the value of the variable in a program can be determined by the interpreter. Interpreter converts the type of a value to whatever is needed for the context.

A variable can be declared either by assigning it a value, then the interpreter implicitly declares it or declare it explicitly using the keyword **var**.

Examples:

```
var counter;  
var index=0;  
var pi=3.14159265;  
var status=true;  
var color= "green";
```

A variable that has been declared but not assigned a value has the value **undefined**.

## JAVASCRIPT RESERVED WORDS

JavaScript has 25 reserved words. Reserved words have some meaning and it is allowed to use for that purpose only.

break	delete	function	Return	typeof
case	Do	if	Switch	var
catch	Else	in	This	void
continue	finally	instanceof	Throw	while
default	For	new	Try	with

## SCREEN OUTPUT AND KEYBOARD INPUT

### ALERT BOX:

Alert box is a very frequently useful to send or write cautionary messages to end user screen. Alert box is created by alert method of window object as shown below.

**Syntax: - window.alert ("message");**

When alert box is popup, the user has to click ok to continue browsing or to perform any further operations.

### Example:

```
<html>
<head>
<title> alert box </title>
  <script language="JavaScript">
    function add( )
    {
      a=20;
      b=40;
      c=a+b;
      window.alert("This is for addition of 2 no's");
      document.write("Result is: "+c);
    }
  </script>
</head>
<body onload="add( )">
</body>
</html>
```

### Output:

Result is 60



### CONFIRM POPUP BOX:

This is useful to verify or accept something from user. It is created by confirm method of window object as shown below.

**Syntax:- window.confirm ("message?");**

When the confirm box pop,s up, user must click either ok or cancel buttons to proceed. If user clicks ok button it returns the boolean value true. If user clicks cancel button, it returns the boolean value false.

#### Example:

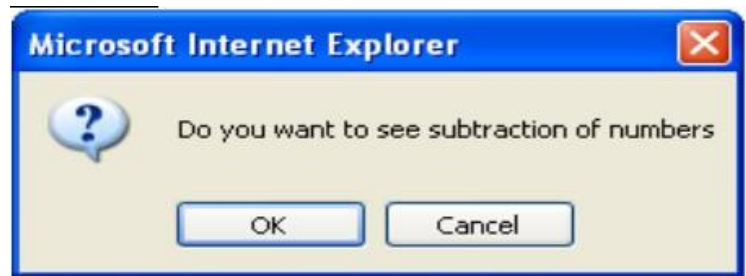
```
<HTML>
<HEAD>
<TITLE> Confirm </TITLE>
<script>
function sub( )
{
    a=50;
    b=45;
    c=a-b;
    x=window.confirm("Do you want to see subtraction of numbers")
    ;if(x==true)
    {
        document.write("result is :"+c);
    }
    else
    {
        document.write("you clicked cancel button");
    }
}
</script>
</HEAD>
<BODY onload="sub( )">
To see the o/p in pop up box:
</BODY>
```

</HTML>

**Output:**

To see the o/p in pop up box:

result is :5



**PROMPT POPUP BOX:-**

It is useful to accept data from keyboard at runtime. Prompt box is created by prompt method of window object.

**Syntax: window.prompt (“message”, “default text”);**

When prompt dialog box arises user will have to click either ok button or cancel button after entering input data to proceed. If user click ok button it will return input value. If user click cancel button the value —null will be returned.

**Example:**

<HTML>

<HEAD>

<TITLE> Prompt </TITLE>

<script>

function fact( )

{

var b=window.prompt("enter +ve integer :", "enter here");

var c=parseInt(b);

a=1;

for(i=c;i>=1;i--)

{

a=a\*i;

}

window.alert("factorial value :"+a);

}

</script>

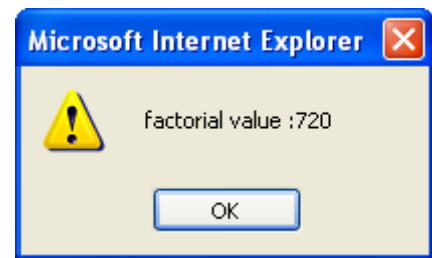
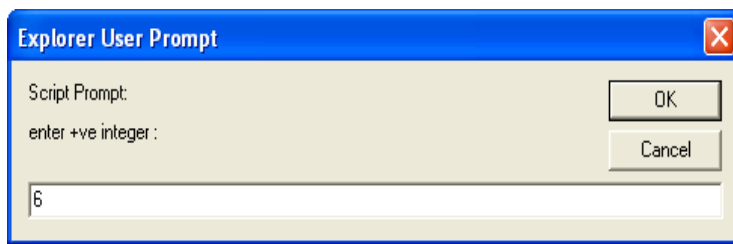
</HEAD>

<BODY onload="fact( )">

</BODY>

</HTML>

## Output:



## Write ( ) Method:

The **write( ) method** in HTML is used to write some content or JavaScript code in a Document.

**Syntax:** document. write (exp1, exp2, exp3, ...);

Here, exp1, exp2, exp3 ..... are all optional arguments, these arguments are appended to the document in order of occurrence

The `writeln()` method is identical to the `document.write()` method, with the addition of writing a newline character after each statement.

### Example:

```
<!DOCTYPE html>
<html>
<body>
<p>Note that write() does NOT add a new line after each statement:</p>
<script>
    document.write("Hello World!");
    document.write("Have a nice
    day!");
</script>
<p>Note that writeln() add a new line after each statement:</p>
<script>
    document.writeln("Hello World!");
    document.writeln("Have a nice
    day!");
</script>
</body>
</html>
```

## Output:

Note that `write()` does NOT add a new line after each statement:

Hello World!Have a nice day!

Note that `writeln()` add a new line after each statement:

Hello World!  
Have a nice day!

## **OPERATORS IN JAVASCRIPT**

In JavaScript, an operator is a special symbol used to perform operations on operands (values and variables).

For example,  $2 + 3$ ; `// 5`

Here `+` is an operator that performs addition, and 2 and 3 are operands.

JavaScript supports the following types of operators.

- ☐ Arithmetic Operators
- ☐ Comparison (or Relational) Operators
- ☐ Logical Operators
- ☐ Assignment Operators
- ☐ Conditional (or ternary) Operators

### **ARITHMETIC OPERATORS**

- ☐ JavaScript supports the following arithmetic operators –
- ☐ Assume variable A holds 10 and variable B holds 20, then –

S. No.	Operator & Description
1	<b>+ (Addition)</b> Adds two operands <b>Ex:</b> $A + B$ will give 30
2	<b>- (Subtraction)</b> Subtracts the second operand from the first <b>Ex:</b> $A - B$ will give -10
3	<b>* (Multiplication)</b> Multiply both operands <b>Ex:</b> $A * B$ will give 200
4	<b>/ (Division)</b> Divide the numerator by the denominator <b>Ex:</b> $B / A$ will give 2



5	<b>% (Modulus)</b> Outputs the remainder of an integer division <b>Ex:</b> B % A will give 0
6	<b>++ (Increment)</b> Increases an integer value by one <b>Ex:</b> A++ will give 11
7	<b>-- (Decrement)</b> Decreases an integer value by one <b>Ex:</b> A-- will give 9

## COMPARISON OPERATORS

- ☐ Comparison Operators also called relational operators, used to write conditions in the statements like if, if – else, switch, loop statements. Expressions that include these set of operators evaluated to true or false.
- ☐ JavaScript supports the following comparison operators –
- ☐ Assume variable A holds 10 and variable B holds 20, then –

Sr.No.	Operator & Description
1	<b>== (Equal)</b> Checks if the value of two operands are equal or not, if yes, then the condition becomes true. <b>Ex:</b> (A == B) is not true.
2	<b>!= (Not Equal)</b> Checks if the value of two operands are equal or not, if the values are not equal, then the condition becomes true. <b>Ex:</b> (A != B) is true.
3	<b>&gt; (Greater than)</b> Checks if the value of the left operand is greater than the value of the right operand, if yes, then the condition becomes true. <b>Ex:</b> (A > B) is not true.
4	<b>&lt; (Less than)</b> Checks if the value of the left operand is less than the value of the right operand, if yes, then the condition becomes true. <b>Ex:</b> (A < B) is true.
5	<b>&gt;= (Greater than or Equal to)</b> Checks if the value of the left operand is greater than or equal to the value of the right operand, if yes, then the condition becomes true. <b>Ex:</b> (A >= B) is not true.
6	<b>&lt;= (Less than or Equal to)</b> Checks if the value of the left operand is less than or equal to the value of the right operand, if yes, then the condition becomes true. <b>Ex:</b> (A <= B) is true.

7	<b>=== (is strictly equal to)</b> This operator checks for the equality of two operands in terms of its value as well as its type. If two operands have same type and have equal value then it returns true otherwise false
8	<b>!== (is strictly not equal to)</b> This operator returns true if the both operands do not have equal value and of different data types.

## LOGICAL OPERATORS

- ☐ Logical Operators are used to combine two or more conditions into single condition. For these operators, and the logical expressions are evaluated to either true or false.
- ☐ JavaScript supports the following logical operators –

S.No.	Operator & Description
1	<b>&amp;&amp; (Logical AND)</b> If both the operands are non-zero, then the condition becomes true. <b>Ex:</b> (A && B) is true.
2	<b>   (Logical OR)</b> If any of the two operands are non-zero, then the condition becomes true. <b>Ex:</b> (A    B) is true.
3	<b>! (Logical NOT)</b> Reverses the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false. <b>Ex:</b> !(A && B) is false.

## ASSIGNMENT OPERATORS

- ☐ Assignment operators are used to assign a literal value or value of an expression to variable.
- ☐ JavaScript supports the following assignment operators –

Sr.No.	Operator & Description
1	<b>= (Simple Assignment)</b> Assigns values from the right side operand to the left side operand <b>Ex:</b> C = A + B will assign the value of A + B into C
2	<b>+= (Add and Assignment)</b> It adds the right operand to the left operand and assigns the result to the left operand. <b>Ex:</b> C += A is equivalent to C = C + A
3	<b>-= (Subtract and Assignment)</b> It subtracts the right operand from the left operand and assigns the result to the left operand. <b>Ex:</b> C -= A is equivalent to C = C – A
4	<b>*= (Multiply and Assignment)</b> It multiplies the right operand with the left operand and assigns the result to the left operand. <b>Ex:</b> C *= A is equivalent to C = C * A

5	<b>/= (Divide and Assignment)</b> It divides the left operand with the right operand and assigns the result to the left operand. <b>Ex:</b> $C /= A$ is equivalent to $C = C / A$
6	<b>%= (Modules and Assignment)</b> It takes modulus using two operands and assigns the result to the left operand. <b>Ex:</b> $C \% = A$ is equivalent to $C = C \% A$

## CONDITIONAL OPERATOR (? :):-

The conditional operator first evaluates an expression for a true or false value and then executes one of the two given statements depending upon the result of the evaluation of condition.

### Usage:

***Conditional - Expression? Expression 1: Expression 2;***

If the condition is evaluated to true then the expression1 is executed, otherwise expression 2 is executed.

## typeof OPERATOR

- ☐ The **typeof** operator is a unary operator used to check the type of variable given to it.
- ☐ The *typeof* operator evaluates to "number", "string", or "boolean" if its operand is a number, string, or boolean.
- ☐ Here is a list of the return values for the **typeof** Operator.

Type	String Returned by typeof
Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"
Function	"function"
Undefined	"undefined"
Null	"object"

## STRING CONCATENATION OPERATOR

Joining multiple strings together is known as concatenation. To join two or more strings together into a single string, the concatenation operator (+) is used. It concatenates two or more string values together and returns another string which is the union of the two operand strings.

### Example 1:

```
var txt1 =
"John";var txt2
= "Doe";
var txt3 = txt1 + " " + txt2;// now txt3 contains "John Doe"
```

### Example 2:

```
var txt1 = "What a very ";  
txt1 += "nice day"; // now txt1 contains "What a very nice day"
```

- ☐ If a number and a string is added then the result will be the string

### Example:

```
var x = 5 + 5; // Now x has 10  
var y = "5" + 5; // Now the variable y has 55  
var z = "Hello" + 5; // Now the variable z has Hello5
```

## TYPE CONVERSIONS

There are two types of type conversions (or) typecasting,

- ☐ Implicit Conversion (casting) done by JavaScript Interpreter
- ☐ Explicit Conversion (casting) is done by the developer

### IMPLICIT TYPE CONVERSION

Converting one type of data value to another type is called typecasting. If required JavaScript engine automatically converts one type of value into another type. It will try to convert it to the right type.

#### Example:

1. "Survey\_number"+10  
The result of above operation will get a string  
"Survey\_number 10"
2. 5\*"12"  
Second operand which is a String will automatically converted to number.  
The result is 60.
3. 5\*"Hello"  
Second operand which is a String cannot be converted to number.  
The result is NaN –Not a number.

When implicit type conversion is made, the result is always not as expected

Examples:

- |  |   |
|--|---|
| <input type="checkbox"/> 5 + null // returns 5         | <input type="checkbox"/> because null is converted to 0               |
| <input type="checkbox"/> "5" + null // returns "5null" | <input type="checkbox"/> because null is converted to "null"          |
| <input type="checkbox"/> "5" + 2 // returns "52"       | <input type="checkbox"/> because 2 is converted to "2"                |
| <input type="checkbox"/> "5" - 2 // returns 3          | <input type="checkbox"/> because "5" is converted to 5                |
| <input type="checkbox"/> "5" * "2" // returns 10       | <input type="checkbox"/> because "5" and "2" are converted to 5 and 2 |

## EXPLICIT TYPE CONVERSION

In explicit type conversion, the programmer forcefully converts the one data type to another data type using the following functions:

- ☐ `Number ()` – it converts the given number into the binary, octal, and hexadecimal
- ☐ `Boolean ()` – converts any type of given value into true or false (or) 0 and 1
- ☐ `parseInt ()` – converts the numerical string to an integer value
- ☐ `parseFloat ()` – converts the numerical string to a float value
- ☐ `String ()` – it converts any given type of value into string type
- ☐ `toString ()` – it converts any given type of value into string type

### **Number ( ):**

**Number()** function is used to convert a value to a Number. It can convert any numerical text and boolean value to a Number. In case of strings of non-numerics, it will convert it to a **NaN** (Not a Number)

#### **Example:**

```
var s = "144";  
var n = Number(s); // now n contain 144(Number)
```

```
var s = true;  
var n = Number(s); // now n contain 1(Number)
```

### **Boolean ( ):**

It converts any type of value to true or false

#### **Example:**

```
var s = 1;  
var n = Boolean(s); // now n contain true(Boolean)
```

```
var s = "0";  
var n = Boolean(s); // now n contain false(Boolean)
```

```
var s = "20";  
var n = Boolean(s); // now n contain true(Boolean)
```

```
var s = "twenty";  
var n = Boolean(s); // now n contain true(Boolean)
```

```
var s = ""; //empty string  
var n = Boolean(s); // now n contain false(Boolean)
```

### **parseInt ( ):**

This function is used to convert a numerical string into integer, if string contains non – numerical values then it returns NaN (Not a Number).

**Example:**

```
var s = "20";  
var n = parseInt(s); // now n contain 20 (integer)  
  
var s = "twenty";  
var n = parseInt(s); // now n contain NaN (Not a Number)  
  
var s = "19twenty";  
var n = parseInt(s); // now n contain 19 (integer)
```

Here, this function parses up to numerical characters when it finds non – numerical character then it stops parsing.

**parseFloat ( )**

The **parseFloat()** function is used to convert the string into a floating-point number. If the string does not contain a numeral value or if the first character of the string is not a Number then it returns **NaN** i.e, not a number. It actually returns a floating-point number parsed up to that point where it encounters a character that is not a Number.

**Example:**

```
var s = "20.56";  
var n = parseFloat(s); // now n contain 20.56 (float value)
```

**toString( )**

The **toString()** method in Javascript is used with a number and converts the number to a string. It is used to return a string representing the specified Number object.

**Syntax:**

```
num.toString(base)    // base specifies the number system
```

**Example:**

```
var num=12;  
document.write("Output : " + num.toString(2));
```

Here, it displays 1100 in binary number system

```
var num=12;  
document.write("Output : " + num.toString(10));
```

Here it displays 12 in decimal number systems

```
var num=12;  
document.write("Output : " + num.toString(8));
```

Here, it displays 14 in octal number systems

# **CONTROL STATEMENTS**

Control Structures are used to control sequential flow of program execution in desired manner to solve the problem.

## **SELECTION STATEMENTS**

Selection Statements are used to select one alternative among the two or more available set of statements. These conditional statements allow you to take different actions depending upon different conditions. There are three conditional statements:

- ☐ if statement, which is used when you want the script to execute if a condition is true
- ☐ if...else statement, which is used when you want to execute one set of code if a condition is true and another if it is false
- ☐ switch statements, which are used when you want to select one block of code from many depending on a situation

### **IF STATEMENT:**

This statement is used to decide whether a block of code is to be executed or not. If the condition is true then the code in the curly braces is executed. Here is the syntax of IF statement:

```
if (condition)
{
    Statements
}
```

### **IF . . . ELSE STATEMENT**

When you have two possible situations and you want to react differently for each, you can use **if...else** statement. This means: “If the conditions specified are met, run the first block of code; otherwise run the second block”. The syntax is as follows:

```
if (condition)
{
    Statements
}
else
{
    Statements
}
```

**Example:**

```

<html>
  <head>
    <title>if/else</title>
  </head>
  <body>
    <script type="text/javascript">
      var a,b,c;
      a=10;b=20;c=30;
      if(a>b)
      {
        if(a>c)
          document.write("a is largest number"+a);
        else
          document.write("c is largest number"+c);
      }
      else
      {
        if(b>c)
          document.write("b is largest number"+b);
        else
          document.write("c is largest number"+c);
      }
    </script>
  </body>
</html>

```

**SWITCH STATEMENT**

A switch statement allows you to deal with several possible results of a condition. You have a single expression. The value of this expression is then compared with the values for each case in the structure. If there is a match, the block of code will execute.

**Syntax:**

```

switch (expression)
{
    case value1: statement(s)
                  break;
    case value2: statement(s)
                  break;
    .....
    .....
    default:      statement(s)
                  break;
}

```

You use the break to prevent code from running into the next case automatically.



**Ex:**

```
<html>
  <head>
    <title>switch</title>
  </head>
  <body>
    <script type="text/javascript">
      var d=new Date();
      ch=d.getMonth();
      switch(ch)
      {
        case 0:document.write("January");
              break;
        case 1:document.write("february");
              break;
        case 2:document.write("march");
              break;
        case 3:document.write("april");
              break;
        case 4:document.write("may");
              break;
        case 5:document.write("june");
              break;
        case 6:document.write("July");
              break;
        case 7:document.write("august");
              break;
        case 8:document.write("september");
              break;
        case 9:document.write("october");
              break;
        case 10:document.write("november");
              break;
        case 11:document.write("december");
              break;
        default: document.write("Invalid choice");
      }
    </script>
  </body>
</html>
```

## **LOOP CONTROL STATEMENTS**

Loop statements are used to execute the same block of code repeatedly to required number of times based on the truth value of a condition. JavaScript supports three looping statements

- A while loop that runs the same block of code while or until a condition is true.
- A do while loop that runs once before the condition is checked. If the condition is true, it will continue to run until the condition is false. The difference between **while** and **do-while** loop is that do while runs once whether the condition is true or false.
- A for loop that runs the same block of code a specified number of times

### **WHILE STATEMENT**

In a while loop, a code block is executed repeatedly until the condition becomes false. The syntax is as follows:

```
while (condition)
{
    Block of statements
}
```

**Ex:**

```
<html>
<head>
    <title>while</title>
</head>
<body>
    <script type="text/javascript">
        var i=1;
        while(i<=10)
        {
            document.write("Number"+i+"It's square"+(i*i)+"<br/>")
            i++;
        }
    </script>
</body>
</html>
```

### **DO ... WHILE STATEMENT**

A **do ... while** loop executes a block of code once and then checks a condition. As long as the condition is true, it continues to loop. So, without evaluating the condition, the statements in the loop runs at least once. Here is the syntax:

```
do
{
    Statements
} while (condition);
```

**Ex:**

```

<html>
<head>
    <title>do - while</title>
</head>
<body>
    <script type= "text/javascript">
        var i=1;
        do
        {
            document.write("Number"+i+"It's square"+(i*i)+"<br/>")
            i++;
        } while(i<=10);
    </script>
</body>
</html>

```

**FOR STATEMENT**

The **FOR LOOP** statement executes a block of code a specified number of times. You use it when you want to specify how many number of times body of the loop is executed.

Here is the syntax:

```

for (initialization; test condition; increment/decrement)
{
    Statements
}

```

**Ex:**

```

<html>
<head>
    <title>FOR LOOP STATEMENT </title>
</head>
<body>
    <script type="text/javascript">
        var i;
        for(i=1;i<=10;i++)
        {
            document.write("Number"+i+"It's square"+(i*i)+"<br/>")
        }
    </script>
</body>
</html>

```

## BREAK STATEMENT

In general, a loop statement is terminated when the condition becomes false. The break statement is used to break the loop forcefully even though the loop condition is true. And the **break** statement is allowed to use with the conditional statement only.

**Ex:**

```
<html>
<head>
    <title>break</title>
</head>
<body>
    <script type="text/javascript">
        var i;
        for(i=10;i>=1;i--)
        {
            if(i==5)
                break;
        }
        document.write("My lucky Number is:"+i)
    </script>
</body>
</html>
```

## CONTINUE STATEMENT

The continue statement is used in a loop to stop the execution of loop statements for the current iteration and continue with the remaining iterations. And the **continue** statement is allowed to use with the conditional statement only

```
<html>
<head>
    <title>continue</title>
</head>
<body>
    <script type="text/javascript">
        var i;
        for(i=10;i>=1;i--)
        {
            if(i==5)
            {
                x=i;
                continue;
            }
            document.write(i+"<br/>");
        }
    </script>
</body>
</html>
```

```

        document.write("My missed Number is:"+x")
</script>
</body>
</html>

```

## **FUNCTIONS**

A function is a piece of code that performs a specific task. The function will be executed by an event or by a call to that function. We can call a function from anywhere within the page (or even from other pages if the function is embedded in an external **.js** file). JavaScript has a lot of built – in functions.

### **Defining functions:**

JavaScript function definition consists of the **function** keyword, followed by the name of the function. A list of arguments to the function are enclosed in parentheses and separated by commas. The statements within the function are enclosed in curly braces { }.

### **Syntax:**

**function** function\_name(var1,var2,...,varX)

```

{
    -----
    -----
    -----
}

```

### **Parameter Passing:**

The parameters values that appear in a call to a function are called actual parameters. The parameters those receives the actual parameters in the function definition when it calls are called formal parameters.

Not every function accepts parameters. When a function receives a value as a parameter, that value is given a name and can be accessed using that name in the function. The names of parameters are taken from the function definition and are applied in the order in which parameters are passed in.

The number of actual parameters in a function call is not checked against the number of formal parameters. In function if actual parameters are excess in number then they are ignored, if the lesser in number, then the excess formal parameters are undefined.

Primitive data types are passed by value in JavaScript. This means that a copy is made of a variable when it is passed to a function, so any modifications made to that parameter does not affect the original value of the variable passed.

Unlike primitive data types, composite types such as arrays and objects are passed by reference rather than value. If references are passed then only changes made in the function affects the actual parameters.

JavaScript uses pass – by – value parameter passing method. However, if a reference to an object is passed to a function and the function made any changes to the formal parameter then the change has no effect on the actual parameter.

A value can also be returned from the function using the keyword **return**. This return statement can be written at the end of the function usually as the control flow goes back to the calling statement.

**Example:**

```
<html>
  <head>
    <title>Defining a function</title>
    <script type="text/javascript">
      function name()
      {
        str= "vignan";
        return str;
      }
    </script>
  </head>
  <body>
    <script type="text/javascript">
      document.write("Department of IT");
      var x=name();
      document.writeln("Welcome to" + x);
    </script>
  </body>
</html>
```

**Examining the function call:**

In JavaScript parameters are passed as arrays. Every function has two properties that can be used to find information about the parameters:

**arguments** - This is an array of parameters that have been passed

**arguments.length** - This is the number of parameters that have been passed into the function

**Example:**

```
<html>
<head>
<script type="text/javascript">
function params(a,b)
{
  document. writeln("<br/>No of parameters passed:"+arguments.length);
  document.writeln("<br/>Parameter Values are:")
  for(var i=0;i<arguments.length;i++)
```

```

        {
        }
        return;
    }

document.writeln("<br/>Parameter" + (i + 1) + " " + arguments[i]);
</script>
</head>
<body>
    <script type="text/javascript">
        params("Ajay", "Arun");
        params("Ajay", "Arjun", "Anand");
    </script>
</body>
</html>

```

### **Output:**

No of parameters  
 passed:2Parameter  
 Values are:  
 Parameter1  
 Ajay  
 Parameter2  
 Arun  
 No of parameters  
 passed:3Parameter  
 Values are:  
 Parameter1 Ajay  
 Parameter2 Arjun  
 Parameter3  
 Anand

## **SCOPE OF VARIABLES**

The scope of a variable is the range of statements over which it is visible. When the JavaScript is embedded in an HTML document, the scope of the variable is the range of lines of the document over which the variable is visible.

- ☐ Variables that are declared explicitly using the keyword **var**, inside a function have LOCAL SCOPE and they can be allowed to access in that function only.
- ☐ Variables that are declared explicitly using the keyword **var**, outside to all functions have GLOBAL SCOPE and they can be allowed to access anywhere.
- ☐ Variables that are not declared explicitly, and implicitly declared by JavaScript interpreter, without using the keyword **var** have GLOBAL SCOPE and they are accessed anywhere in the HTML document. Even though, if such an implicit declaration has made inside a function, its scope is GLOBAL.

- If a variable that is defined both as a Local Variable and Global Variable then the local variable has more precedence, then the global variable is hidden.

## **JAVASCRIPT OBJECTS**

In JavaScript, Objects are collection of properties, which may be either a data property or a function or a method.

- Data properties are appear in two categories: primitive values or references to other objects.
  - Sub programs that are called through objects are called methods.
  - Sub programs that are not called through objects are called functions.
- All objects in a JavaScript are indirectly accessed through variables. The properties of an object are referenced by attaching the name of the property to the variable that references the object. For example, if **car** is the variable that refers an object that has the property **engine**, this **engine** property can be accessed as **car.engine**.
  - The root object in JavaScript is **Object**. It is ancestor to all the objects.
  - A JavaScript Object can appear as a list of property – value pairs. The properties are names and values are data values.
  - Properties of objects can be added or deleted at any time during execution. So they are Dynamic.

### **OBJECT CREATION AND MODIFICATION**

The web designer can create an object and can set its properties as per his requirements. The object can be created using **new** expression. Initially the object with no properties can be set using following statements.

```
obj=new Object();
```

Then by using dot operator we can set the properties for that object. The object can then be modified by assigning values to that object.

#### ***Example:***

```
<html>
<head>
<title>Object creation</title>
</head>
<body>
<script type="text/javascript">var
student;
student=new Object(); student.id=10;
student.name="Vignan";
```



```
document.write("The ID is:"+student.id);
document.write("The Name is:"+student.name);
</script>
</body>
</html>
```

## JAVASCRIPT CONSTRUCTORS

- Objects in Java Script are created using the constructors too.
- Constructors are like regular functions, but we use them with the new keyword.

There are two types of constructors:

1. Built-in constructors such as Array and Object, which are available automatically in the execution environment at runtime
2. Custom constructors, which define properties and methods for your own type of object.

A constructor is useful when you want to create multiple similar objects with the same properties and methods. It's a convention to capitalize the starting letter in the name of constructor to distinguish them from regular functions.

Consider the following code:

```
function Book()
{
// unfinished code
}
var myBook = new Book();
```

The above line of the code creates an instance of Book and assigns it to a variable.

- Although the Book constructor doesn't do anything, **myBook** is still an instance of it. And it is possible to add the properties to the instance **myBook** dynamically. This can be illustrated in the following example:

```
<html>
<body><script type= "text/javascript">
    function Book()
    {
        this. title= "Programming WWW";
        this. price = 500.00;
        this. author= "Tim Berners Lee";
    }
var myBook = new Book(); Document.writeln("Book
Title:"+myBook.title); Document.writeln("Book
Author"+myBook.author);Document.writeln("Book
Price:"+myBook.price);
```

</script> </body> </html>

### **Determining the type of an instance:**

- To find out whether an object is an instance of another one, we use the instanceof operator:

```
myBook instanceof Book // true
```

```
myBook instanceof String // false
```

- Note that if the right side of the instanceof operator isn't a function, it will throw an error:

```
myBook instanceof {};
```

```
// TypeError: invalid 'instanceof' operand ({})
```

- Another way to find the type of an instance is to use the constructor property.

Consider the following code fragment:

```
myBook.constructor === Book; // true
```

The constructor property of myBook points to Book, so the strict equality operator returns true.

## **STRING OBJECT**

String is a set of characters enclosed in a pair of single quotes or double quotes. In JavaScript

using string object, many useful string related functionalities can be done. Some commonly used methods of string object are concatenating two strings, converting the string to uppercase or lowercase, finding the substring of a given string and so on.

A String object can be declared as follows:

```
var str = new String("Vignan LARA");
```

## **PROPERTY:**

### **□ Length**

This property of string returns the length of the string

#### **Example:**

```
var str = new String("Vignan LARA");  
str.length //gives 5
```

## **METHODS:**

### **1. charAt(index)**

This method returns the character specified by index

#### **Example:**

```
var str = new String("Vignans LARA");  
str.charAt(2); // this gives the character at the index position 2 i.e. g
```

### **2. indexOf(substring [, offset])**

This method returns the index of substring if found in the main string. If the substring is not found then it returns **-1**. By default the **indexOf( )** function starts searching the substring from index **0**, however, an optional offset may be specified, so that the search starts from that position.

#### **Example:**

```
var str = new String("Vignan LARA");  
str.indexOf("LARA"); // This gives the index position 7 as it finds the "LARA" there.
```

### **3. lastIndexOf(substring [,offset])**

This method returns the index of substring if found in the main string (i.e. last occurrence). If the substring is not found then it returns **-1**. By default the **lastIndexOf()** function starts from the last index i.e. from backwards, however, an optional offset may be specified, so that the search starts from that position in backwards.

#### **Example:**

```
var str = new String("Vignan LARA");  
str.lastIndexOf("LARA");
```

#### 4. **str1.concat(str2 [,str3 ..strN])**

This method is used to concatenate strings together.

For example, `s1.concat(s2)` returns the concatenated string of `s1` and `s2`. The original strings don't get altered by this operation.

```
var s1="Hello"; var  
s2="Vignan";  
var s3 = s1.concat(s2);
```

Now `s3` contains the string `Hello Vignan`

#### 5. **substring(start [,end] )**

This method returns the substring specified by **start** and **end** indices (upto **end** index, but not the character at **end** index). If the **end** index is not specified, then this method returns substring from **start** index to the end of the string.

**Example:**

```
var str = new String("Vignan LARA");  
str.substring(7); // This gives the string from the index position 7 to the end of the string.  
str.substring(0, 7); // This gives the string from the index position 0 to 6
```

#### 6. **substr(index [,length])**

This method returns substring of specified number of characters (**length**), starting from **index**. If the length is not specified it returns the entire substring starting from **index**.

**Example:**

```
"Vignan LARA".substr(7,4);
```

It returns the sub string of 4 characters length from index position 7 i.e. `LARA`

#### 7. **toLowerCase( )**

This method returns the given string converted into lower case. The original string is not altered by this operation.

**Example:**

```
var str = new String("Vignan LARA");  
str.toLowerCase();
```

#### 8. **toUpperCase( )**

This method returns the given string converted into upper case. The original string is not altered by this operation.

**Example:**

```
var s="Vignan LARA";  
s.toUpperCase();
```

### 9. `split(separator [,limit] )`

Splits the string based on the *separator* specified and returns that array of substrings. If the **limit** is specified only those number of substrings will be returned

#### **Example1:**

```
var s="VLITS#LARA#GNT#AP";  
var t=s.split("#");
```

Now, the variable **t** contains an array of sub strings – VLITS, LARA, GNT, AP.

### **MATH OBJECT**

The Math Object provides a collection of properties of Number Objects and the methods that operate on Number Objects. This Math object has methods for the trigonometric functions such as `sin`, `cos` as well as other commonly used operations such as `sqrt ()`, `round ()`, `floor ()`, `ceil ()`, `abs ()`, `log ()`, etc.

The following example rounds PI to the nearest whole number (integer) and writes it to the screen.

```
var numPI = Math.PI;  
numPI = Math.round(numPI);  
document.write (numPI);
```

#### **Example:**

```
<html>  
<head>  
<title>Math object</title>  
</head>  
<body>  
<script type="text/javascript">  
var number=100;  
document.write("Square root of number is:"+Math.sqrt(num));  
</script>  
</body>  
</html>
```

For performing the mathematical computations there are some useful methods available from math object.

- a. `sqrt(num)` – returns the square root of given number where  $\text{num} > 0$
- b. `abs(num)` – returns the absolute value of num i.e it ignores the sign of the number
- c. `ceil(num)` – returns the integer which is equal to the smallest of the integers those are greaterthan num.
- d. `floor(num)` – returns the integer which is equal to the largest of the integers those are smallerthan num.
- e. `pow(a,b)` – returns the a to the power of b value
- f. `min(a,b)` – returns the minimum of a and b.
- g. `max(a,b)` – returns the maximum of a and b.
- h. `sin(num)` – returns the sine value of num.

- i. `cos(num)` – returns the cosine value of num.
- j. `tan(num)` – returns the tan value of the given num.
- k. `exp(num)` – returns the e to the power of num.

## **DATE OBJECT**

The Date object is used when the information about the current date and time is useful in the program. The Date object contains a collection of methods that are used to perform the manipulation operations on date and time values.

A Date object is created as follows:

**`var today = new Date();`**

Here, today is the object of type Date that contains the current system date and time.

This Date and Time value is in the form of computer's local time (system's time) or it can be in the form of UTC(Universal Coordinated Time), formerly known as GMT (Greenwich Mean Time).

It is also possible to create a date object that set to a specific date or time, in which case you need to pass it one of four parameters:

1. One can pass milliseconds – this value should be the number of milliseconds from 01/01/1970.

**`var bday = new Date(milliseconds);`**

2. One can pass the date in the form of string

**`var bday = new Date(string);`**

3. One can pass year, month, and day.

**`var bday = new Date(year, month, day);`**

4. One can pass year, month, day, hour, minutes and seconds

**`var bday = new Date(year, month, day, hour, minutes, seconds);`**

The following table is the list of some of the methods of Date Object:

5. `getDate()` – returns the day of the month.
6. `getMonth()` – returns the month of the year, ranges from 0 to 11
7. `getDay()` – returns the day of the week ranges from 0 to 6
8. `getFullYear` – returns the year
9. `getTime ()` – returns the number of milliseconds from Jan 1, 1970
10. `getHours()` – returns the number of hours range from 0 to 23
11. `getMinutes` – returns the number of minutes range from 0 to 59
12. `getSeconds` – returns the number of seconds range from 0 to 59
13. `getMilliseconds` – returns the number of milliseconds range from 0 to 999.

**Example:**

```
<html>
<head>
<title>Date object</title>
</head>
<body>
    <script type="text/javascript">
        var mydate=new Date();
        document.write("The Date is:"+mydate.toString()+"<br/>");
        document.write("Today's Date is:"+mydate.getDate()+"<br/>");
        document.write("Current year is:"+mydate.getFullYear()+"<br/>");
        document.write("Minutes is:"+mydate.getMinutes()+"<br/>");
        document.write("Seconds is:"+mydate.getSeconds()+"<br/>");
    </script>
</body>
</html>
```

**Output:**

The Date is: Fri May 07 2021 19:22:10 GMT+0530  
Today Date is: 7  
Current year is:  
2021 Minutes is: 22  
Seconds is: 10

## **ARRAYS**

Array is a collection of similar type of elements which can be referred by a common name. Any element in an array is referred by an array name. The particular position of an element in an array is called "Index" or "Subscript".

In java script the array can be created using array object.

**Create an Array:**

An array can be created in following ways:

1. var arrName= ["a","b","c"];
2. var arrName= new Array(3); // 3 is the size

arrName[0]= "a";;

arrName[1]= "b";

arrName[2]= "c";

3. `var arrName=new Array("a", "b", "c");`

Here "a", "b", "c" are elements of the array

```
<html>
<head>
<title>Defining a function</title>
</head>
<body>
<script type="text/javascript">
var a=new Array(5);
for(i=0;i<=5;i++)
{
a[i]=i; document.write(a[i]+"<br/>");
}
</script>
</body>
</html>
```

4. One property that is frequently used to determine number of elements in an array is **length**.

Example:

```
var list = [11,22,33,44,55];
document.writeln("Length of the list is"+list.length); // displays the length 5
```

### **Array Methods:**

Here is a list of the methods of the Array object along with their description.

S. No.	Method & Description
1	<b>concat()</b> Returns a new array comprised of this array joined with other array(s) and/or value(s).  <code>var arr1 = ["Cecilie", "Lone"];</code> <code>var arr2 = ["Emil", "Tobias", "Linus"];</code> <code>var arr3 = ["Robin", "Morgan"];</code> <code>var myChildren = arr1.concat(arr2, arr3); // Concatenates arr1 with arr2 and arr3</code>
2	<b>indexOf()</b> Returns the first (least) index of an element within the array equal to the specified value, or -1 if none is found.  <code>var fruits = ["Banana", "Orange", "Apple", "Mango", "Orange"];</code> <code>document.writeln(fruits.indexOf("Orange")); // it gives 1</code>



3	<b>lastIndexOf()</b> Returns the last (greatest) index of an element within the array equal to the specified value, or -1 if none is found. <pre>var fruits = ["Banana", "Orange", "Apple", "Mango", "Orange"]; document.writeln(fruits.lastIndexOf("Orange")); // it gives 4</pre>
4	<b>pop()</b> Removes the last element from an array and returns that element. <pre>var fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.pop();           // Removes the last element ("Mango") from fruits</pre>
5	<b>push()</b> Adds one or more elements to the end of an array and returns the new length of the array. <pre>var fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.push("Kiwi");    // Adds a new element ("Kiwi") to fruits</pre>
6	<b>reverse()</b> Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first. <pre>var fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.reverse(); // it gives "Mango", "Apple", "Orange", "Banana"</pre>
7	<b>shift()</b> Removes the first element from an array and returns that element. <pre>var fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.shift();      // Removes the first element "Banana" from fruits</pre>
8	<b>unshift()</b> Adds one or more elements to the front of an array and returns the new length of the array. <pre>var fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.unshift("Lemon"); // Adds a new element "Lemon" to fruits</pre>
9	<b>slice()</b> Extracts a section of an array and returns a new array. <pre>var fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.slice(2); // it extracts a new array starting from index 2 i.e. Apple</pre>
10	<b>sort()</b> Sorts the elements of an array <pre>var fruits = ["Banana", "Orange", "Apple", "Mango"]; fruits.sort() // it gives "Apple", "Banana", "Mango", "Orange"</pre>

11	<b>toString()</b> Returns a string with array elements separated by Commas  var fruits = ["Banana", "Orange", "Apple", "Mango"]; document.writeln(fruits.toString()); // it gives "Banana, Orange, Apple, Mango"
----	--

### **Passing An Array to Function:**

We can pass an entire array as a parameter to the function. When an array is passed to the function the array name is simply passed.

#### ***Example:***

```

<html>
<head> <title>passing array to a function</title>
<script type="text/javascript">
function display(a)
{
    document.write("The array elements are:");
    i=0;
    while(i < a.length)
    {
        document.write(a[i]+"<br/>");
        i++;
    }
}
</script>
</head>
<body>
<script type="text/javascript">var
arr=[11, 22, 33, 44, 55];
display(arr);
</script>
</body>
</html>

```

### **PATTERN MATCHING USING REGULAR EXPRESSIONS**

A regular expression is a sequence of characters that forms a **search pattern**. When you search for data in a text, you can use this search pattern to describe what you are searching for.

A regular expression can be a single character, or a more complicated pattern. Regular expressions can be used to perform all types of **text search** and **text replace** operations.

EXAMPLE:

### **Static Creation of Regular Expression**

```
var patt = /w3schools/i;
```

**/w3schools/i** is a regular expression.

**w3schools** is a pattern (to be used in a search).

**i** is a modifier (modifies the search to be case-insensitive).

```
var str = " Visit W3Schools";
```

```
str.match (pattern);
```

### **Dynamic Creation of Regular Expression:**

```
var pattern = new RegExp ("/w3schools/i");var str
```

```
= " Visit W3Schools";
```

```
pattern.exec (str);
```

JavaScript has the below given Regular Expression Grammar:

Token	Description
^	Match at the start of the input string
\$	Match at the end of the input string
*	Match 0 or more times
+	Match 1 or more times
?	Match 0 or 1 time
a/b	Match „a“ or „b“
{n}	Match the string „n“ times
\d	Match a digit
\o	Match anything except for digits
\w	Match any alpha numeric character or the underscore
\W	Match anything except alpha numeric characters or underscore
\s	Match a white space character

<code>\S</code>	Match anything except for white space characters
<code>[---]</code>	Create a set of characters, one of which must match, if the operation is to be successful. If we need to specify a range of characters, then separate the first & last with Hyphen (-) Ex: <code>[0-9]</code> & <code>[D-G]</code> .
<code>[^---]</code>	Creates a set of characters which must not match. If any character in the set matches then operation has failed. Ex: <code>[^d-q]</code>

### ***Regular Expressions using String Methods:***

In JavaScript, regular expressions are often used with the **string methods**:

`search()` and `replace()`.

5. The **`search()`** method uses an expression to search for a match, and returns the position of the match.
6. The **`replace()`** method returns a modified string where the pattern is replaced.`

### **RegExp class functions**

#### **1. `exec (string)`**

It executes a search for the matching pattern in its parameter string. It returns an array holding the results of the operation. If the search fails then it returns the null value.

#### **2. `test (string)`**

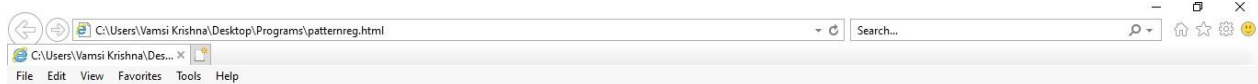
Search for a match in its parameter string. Returns true if a match is found otherwise it returns false

### **Using String `search()` With a Regular Expression**

Use a regular expression to do a case-insensitive search for "w3schools" in a string:

```
<html>
<body>
<h2>JavaScript Regular Expressions</h2>
<p>Search a string for "w3Schools", and display the position of the match:</p>
<script>
    var str = "Visit W3Schools!";
    var n = str.search(/w3Schools/i);
    document.writeln(n);
</script>
</body>
</html>
```

## Outcome:



### JavaScript Regular Expressions

Search a string for "w3Schools", and display the position of the match:

6

### Use String replace() With a Regular Expression

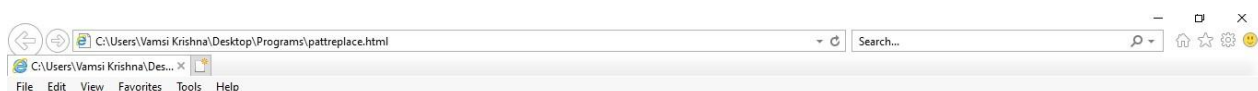
Use a case insensitive regular expression to replace Microsoft with W3Schools in a string:

```
<html>
<body>
<h2>JavaScript Regular Expressions</h2>
<p>Replace "microsoft" with "W3Schools" in the paragraph below:</p>
<button onclick="myFunction()">Try it</button>
<p id="demo">Please visit Microsoft and Microsoft!</p>
<script>

    function myFunction()
    {
        var str = document.getElementById("demo").innerHTML;
        var txt = str.replace(/microsoft/i, "W3Schools");
        document.getElementById("demo").innerHTML = txt;
    }

</script>
</body>
</html>
```

## Outcome:



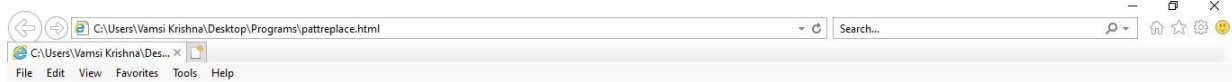
### JavaScript Regular Expressions

Replace "microsoft" with "W3Schools" in the paragraph below:

Try it

Please visit Microsoft and Microsoft!

On clicking the **Try it** button, Web page is displayed as follows:



## JavaScript Regular Expressions

Replace "microsoft" with "W3Schools" in the paragraph below:

[Try it](#)

Please visit W3Schools and Microsoft!

## Introduction to Angular JS

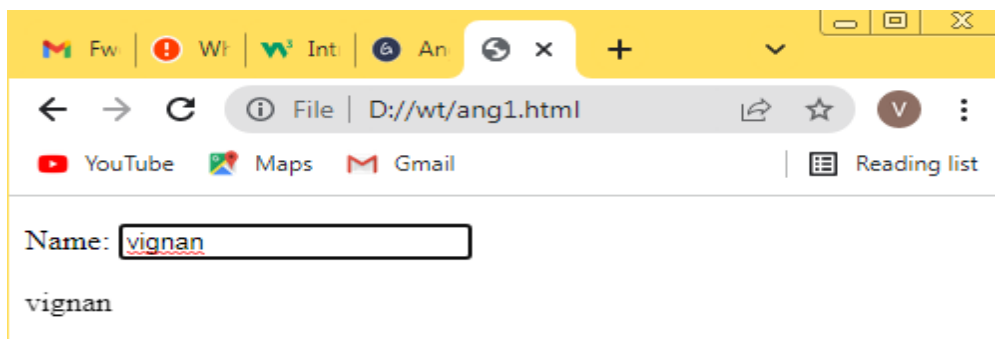
- AngularJS extends HTML with new attributes.
- AngularJS is perfect for Single Page Applications (SPAs).
- AngularJS is easy to learn.
- AngularJS is a **JavaScript framework** written in JavaScript. It can be added to an HTML page with a `<script>` tag.
- AngularJS extends HTML attributes with **Directives**, and binds data to HTML with **Expressions**.
- AngularJS is distributed as a JavaScript file, and can be added to a web page with a script tag:  
`<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>`
- AngularJS extends HTML with **ng-directives**.
- The **ng-app** directive defines an AngularJS application.
- The **ng-model** directive binds the value of HTML controls (input, select, textarea) to application data.
- The **ng-bind** directive binds application data to the HTML view.

### AngularJS Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="">
  <p>Name: <input type="text" ng-model="name"></p>
  <p ng-bind="name"></p>
</div>

</body>
</html>
```



**Example explanation:**

AngularJS starts automatically when the web page has loaded.

The **ng-app** directive tells AngularJS that the <div> element is the "owner" of an AngularJS **application**.

The **ng-model** directive binds the value of the input field to the application variable **name**.

The **ng-bind** directive binds the content of the <p> element to the application variable **name**.

**AngularJS Directives**

AngularJS directives are HTML attributes with an **ng** prefix.

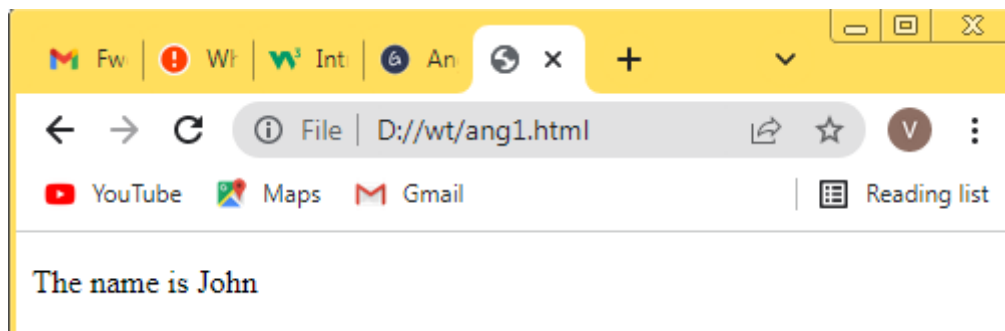
The **ng-init** directive initializes AngularJS application variables.

**AngularJS Example**

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>
<div ng-app="" ng-init="firstName='John'">

<p>The name is <span ng-bind="firstName"></span></p>

</div>
</body>
</html>
```



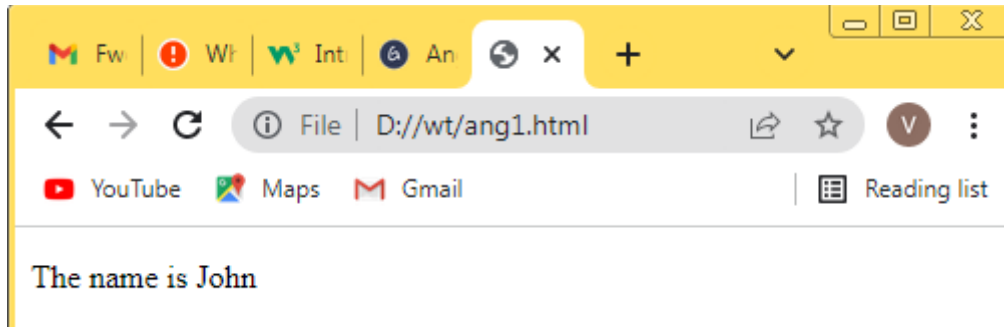
Alternatively with valid HTML:

**AngularJS Example**

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>
```



```
<div data-ng-app="" data-ng-init="firstName='John'">
<p>The name is <span data-ng-bind="firstName"></span></p>
</div>
```



You can use **data-ng-**, instead of **ng-**, if you want to make your page HTML valid.

## AngularJS Expressions

AngularJS expressions can be written inside double braces: **{{ expression }}**.

AngularJS expressions are used to bind data to HTML the same way as the ng-bind directive. AngularJS displays the data exactly at the place where the expression is placed.

AngularJS expressions can also be written inside a directive: **ng-bind="expression"**.

AngularJS will resolve the expression, and return the result exactly where the expression is written.

**AngularJS expressions** are much like **JavaScript expressions**: They can contain literals, operators, and variables.

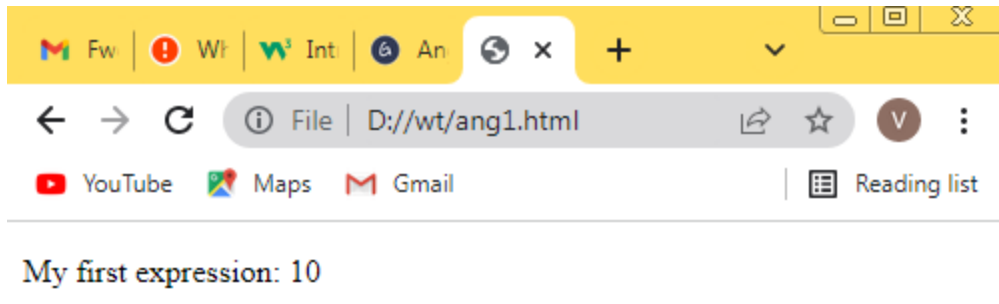
Example **{{ 5 + 5 }}** or **{{ firstName + " " + lastName }}**

Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="">
  <p>My first expression: {{ 5 + 5 }}</p>
</div>

</body>
</html>
```



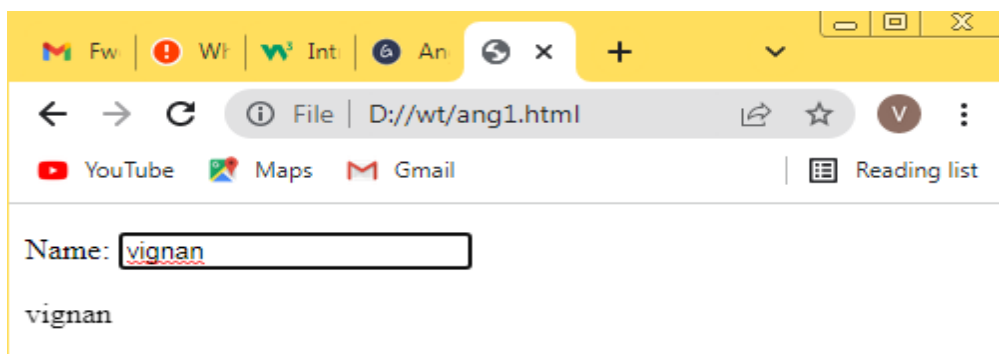
AngularJS expressions bind AngularJS data to HTML the same way as the **ng-bind** directive.

### AngularJS Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="">
  <p>Name: <input type="text" ng-model="name"></p>
  <p>{{ name }}</p>
</div>

</body>
</html>
```



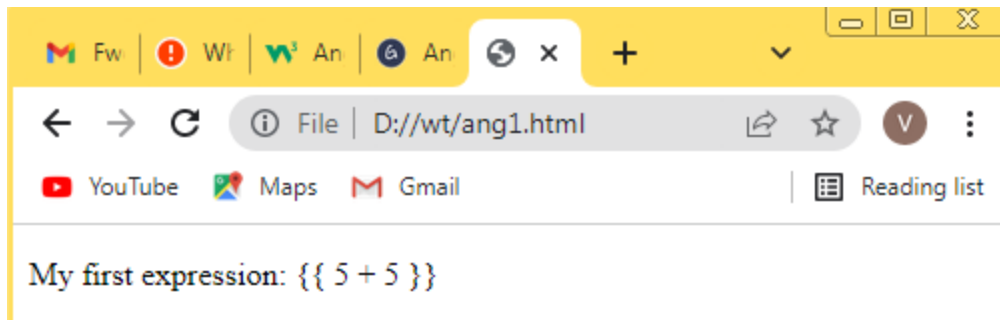
If you remove the **ng-app** directive, HTML will display the expression as it is, without solving it:

### Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div>
  <p>My first expression: {{ 5 + 5 }}</p>
</div>
```

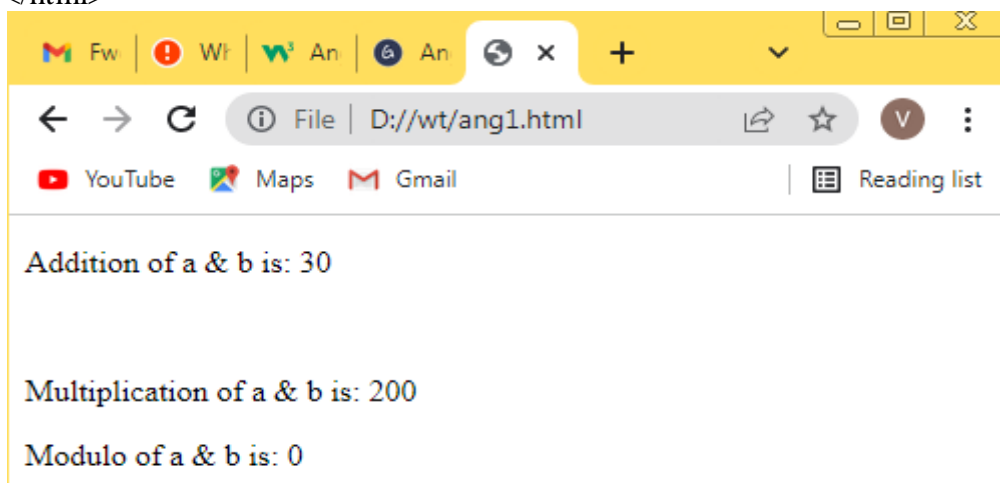
```
</body>  
</html>
```



You can write expressions wherever you like, AngularJS will simply resolve the expression and return the result.

### Arithmetic operations Example:

```
<!DOCTYPE html>  
<html>  
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>  
<body>  
  
<div ng-app="" ng-init="a=10;b=20">  
  
<p>Addition of a & b is: {{ a+b }}</p><br>  
<p>Multiplication of a & b is: {{ a*b }}</p>  
<p>Modulo of a & b is: {{ b%a }}</p>  
  
</div>  
</body>  
</html>
```



**Example:** Let AngularJS change the value of CSS properties.

Change the color of the input box below, by changing its value:

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="" ng-init="myCol='lightblue'">

<input style="background-color:{{ myCol }}" ng-model="myCol">

</div>
</body>
</html>
```

## AngularJS Numbers

Expressions can be used to work with numbers as well. AngularJS numbers are like JavaScript numbers:

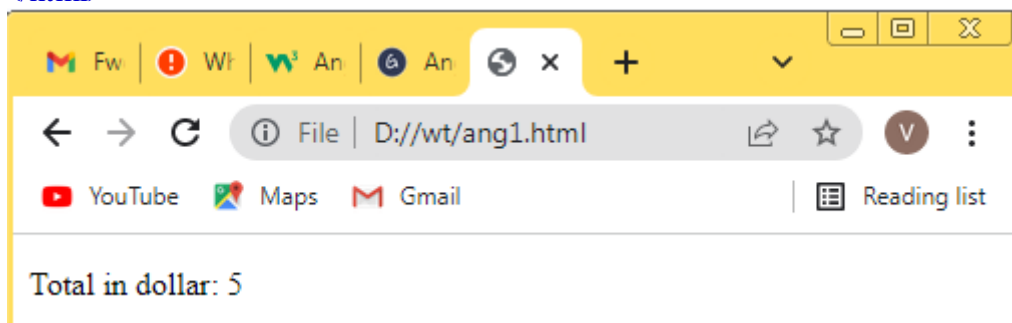
Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="" ng-init="quantity=1;cost=5">

<p>Total in dollar: {{ quantity * cost }}</p>

</div>
</body>
</html>
```



1. The ng-init directive is used in angular.js to define variables and their corresponding values in the view itself. It's somewhat like defining local variables to code in any programming language. In this case, we are defining 2 variables called quantity and cost and assigning values to them.
2. We are then using the 2 local variables and multiplying their values.

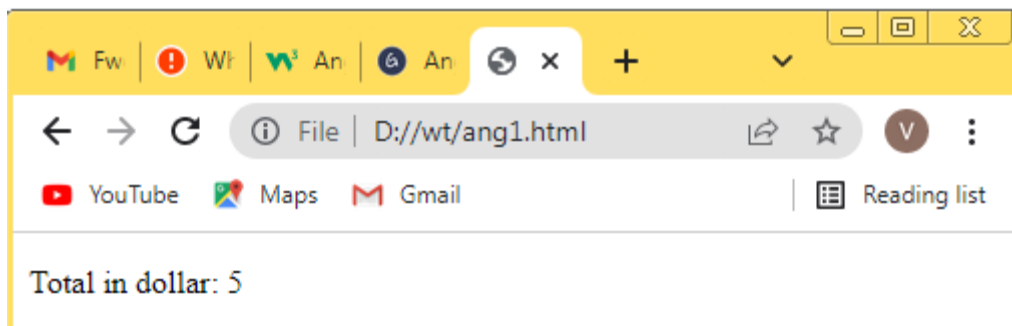
Same example using **ng-bind**:

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="" ng-init="quantity=1;cost=5">

<p>Total in dollar: <span ng-bind="quantity * cost"></span></p>

</div>
</body>
</html>
```



## AngularJS Strings

Expressions can be used to work with strings as well. AngularJS strings are like JavaScript strings.

In this example, we are going to define 2 strings of “firstName” and “lastName” and display them using expressions accordingly.

### Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

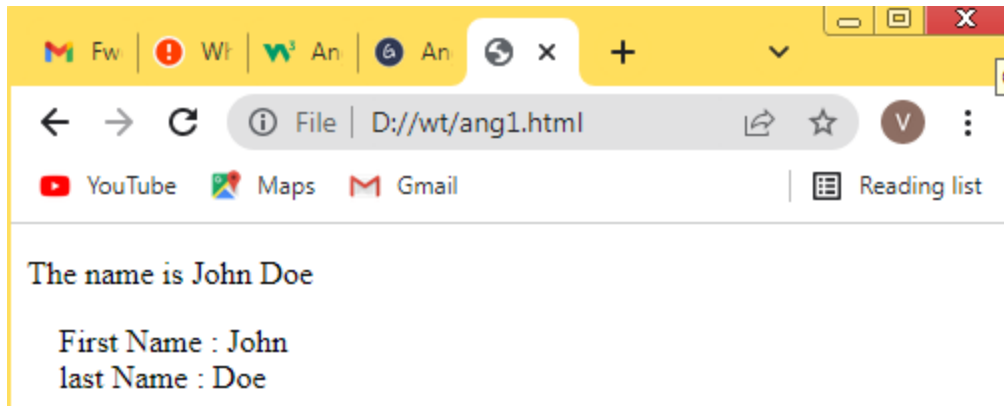
<div ng-app="" ng-init="firstName='John';lastName='Doe'">

<p>The name is {{ firstName + " " + lastName }}
```

```

<div>
  First Name : {{firstName}}<br>
  last Name : {{lastName}}
</div>
</body>
</html>

```



1. The ng-init directive is used to define the variables firstName with the value “Guru” and the variable lastName with the value of “99”.
2. We are then using expressions of {{firstName}} and {{lastName}} to access the value of these variables and display them in the view accordingly.

### Same example using **ng-bind**:

```

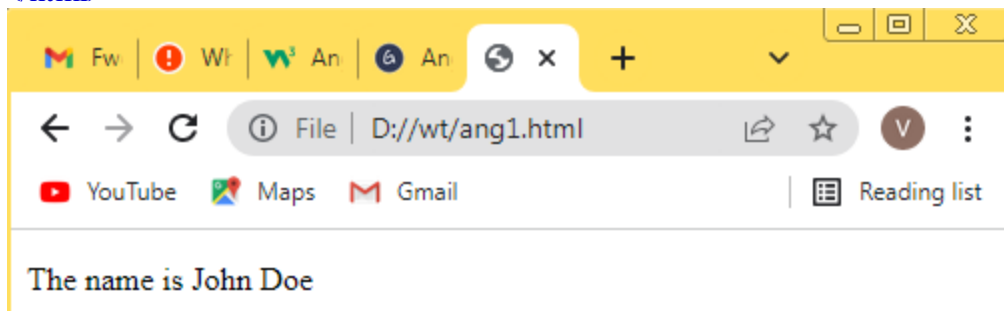
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="" ng-init="firstName='John';lastName='Doe'">

<p>The name is <span ng-bind="firstName + ' ' + lastName"></span></p>

</div>
</body>
</html>

```



## AngularJS Objects

Expressions can be used to work with [JavaScript](#) objects as well. A javascript object consists of a name-value pair.

Below is an example of the syntax of a javascript object.

### Syntax:

```
var car = {type:"Ford", model:"Explorer", color:"White"};
```

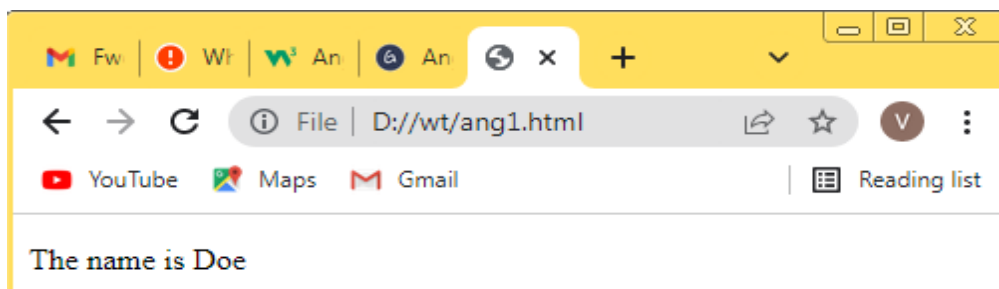
In this example, we are going to define one object as a person object which will have 2 key value pairs of “firstName” and “lastName”.

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="" ng-init="person={ firstName:'John',lastName:'Doe'}">

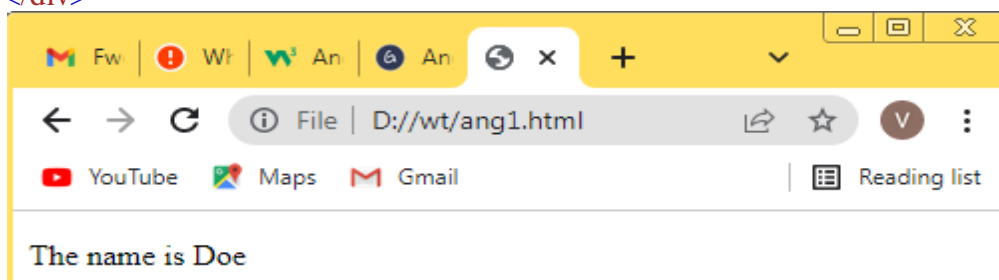
<p>The name is {{ person.lastName }}</p>

</div>
</body>
</html>
```



### Same example using ng-bind:

```
<div ng-app="" ng-init="person={ firstName:'John',lastName:'Doe'}">
<p>The name is <span ng-bind="person.lastName"></span></p>
</div>
```



1. The ng-init directive is used to define the object person which in turn has key value pairs of firstName with the value “John” and the variable lastName with the value of “Doe”.
2. We are then using expressions of {{person.firstName}} and {{person.lastName}} to access the value of these variables and display them in the view accordingly. Since the actual member variables are part of the object person, they have to access it with the dot (.) notation to access their actual value.

### AngularJS Arrays

Array is defined as a set of values with the similar data items. We can use [index] to access each element of the array.

AngularJS arrays are like JavaScript arrays:

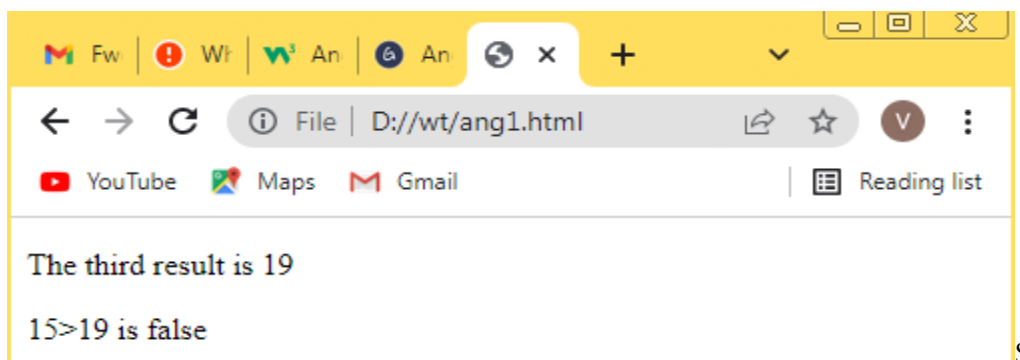
#### Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="" ng-init="points=[1,15,19,2,40]">

<p>The third result is {{ points[2] }}</p>
<p>15>19 is {{ points[1]>points[2] }}</p>

</div>
</body>
</html>
```



#### Same example using ng-bind:

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
```

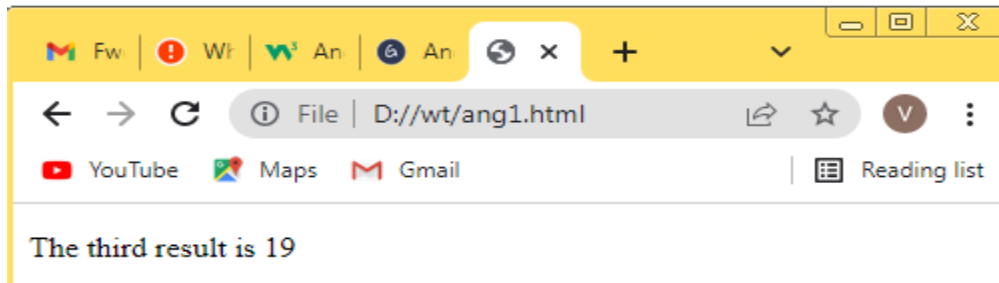


```
<body>

<div ng-app="" ng-init="points=[1,15,19,2,40]">

<p>The third result is <span ng-bind="points[2]"></span></p>

</div>
</body>
</html>
```



### AngularJS Expressions vs. JavaScript Expressions

- Like JavaScript expressions, AngularJS expressions can contain literals, operators, and variables.
- Unlike JavaScript expressions, AngularJS expressions can be written inside HTML.
- AngularJS expressions do not support conditionals, loops, and exceptions, while JavaScript expressions do.
- AngularJS expressions support filters, while JavaScript expressions do not.

### AngularJS Forms

Forms in AngularJS provides data-binding and validation of input controls.

#### **Input Controls:**

Input controls are the HTML input elements:

- input elements
- select elements
- button elements
- textarea elements

#### **Data-Binding**

Input controls provides data-binding by using the **ng-model** directive.

```
<input type="text" ng-model="firstname">
```

The application does now have a property named **firstname**.

The **ng-model** directive binds the input controller to the rest of your application.

The property **firstname**, can be referred to in a controller:

### Example

```
<script>
var app = angular.module('myApp', []);
app.controller('formCtrl', function($scope) {
  $scope.firstname = "John";
});
</script>
```

It can also be referred to elsewhere in the application:

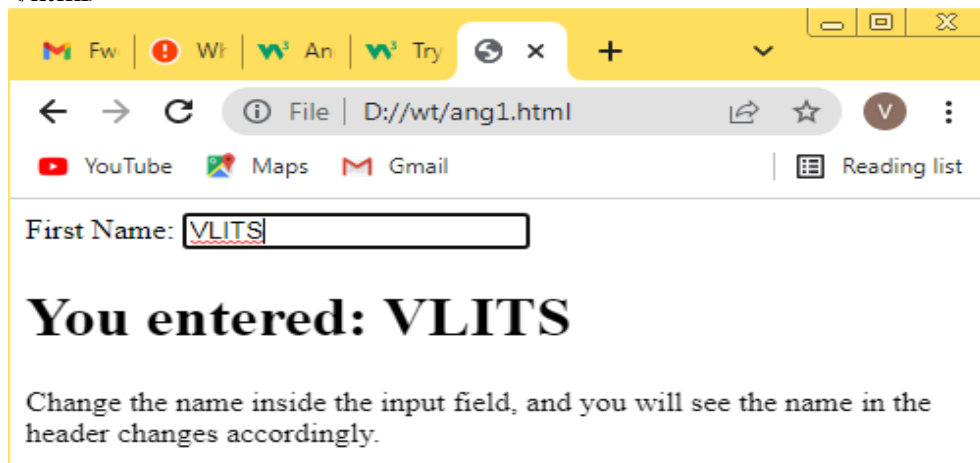
### Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>
```

```
<div ng-app="">
  <form>
    First Name: <input type="text" ng-model="firstname">
  </form>
  <h1>You entered: {{firstname}}</h1>
</div>
```

<p>Change the name inside the input field, and you will see the name in the header changes accordingly.</p>

```
</body>
</html>
```



### Checkbox

A checkbox has the value **true** or **false**. Apply the **ng-model** directive to a checkbox, and use its value in your application.

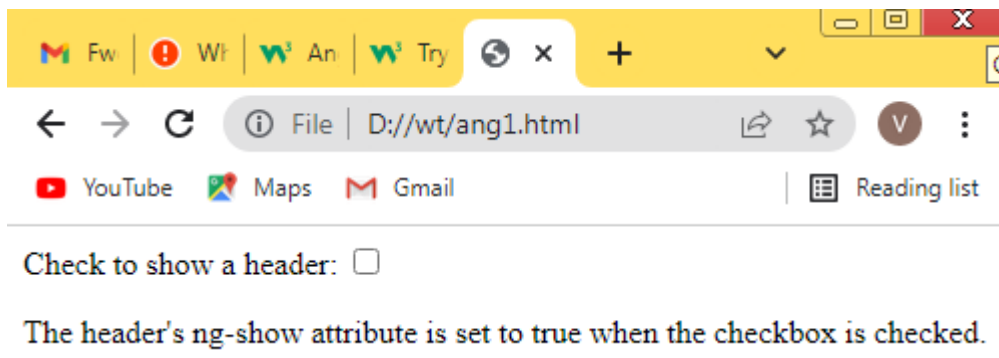
### Example

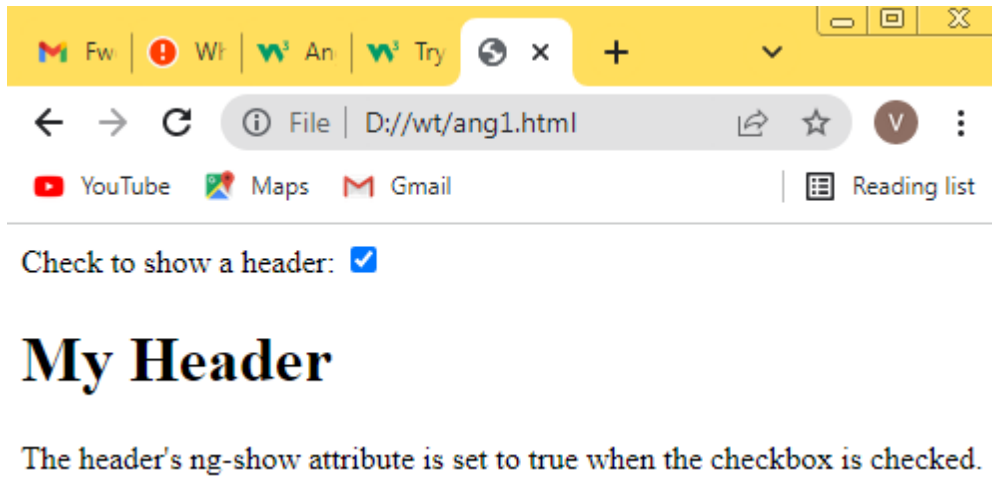
Show the header if the checkbox is checked:

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="">
  <form>
    Check to show a header:
    <input type="checkbox" ng-model="myVar">
  </form>
  <h1 ng-show="myVar">My Header</h1>
</div>

<p>The header's ng-show attribute is set to true when the checkbox is checked.</p>
</body>
</html>
```





### Radiobuttons:

Bind radio buttons to your application with the **ng-model** directive.

Radio buttons with the same **ng-model** can have different values, but only the selected one will be used.

### Example

Display some text, based on the value of the selected radio button:

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body ng-app="">

<form>
  Pick a topic:
  <input type="radio" ng-model="myVar" value="dogs">Dogs
  <input type="radio" ng-model="myVar" value="tuts">Tutorials
  <input type="radio" ng-model="myVar" value="cars">Cars
</form>

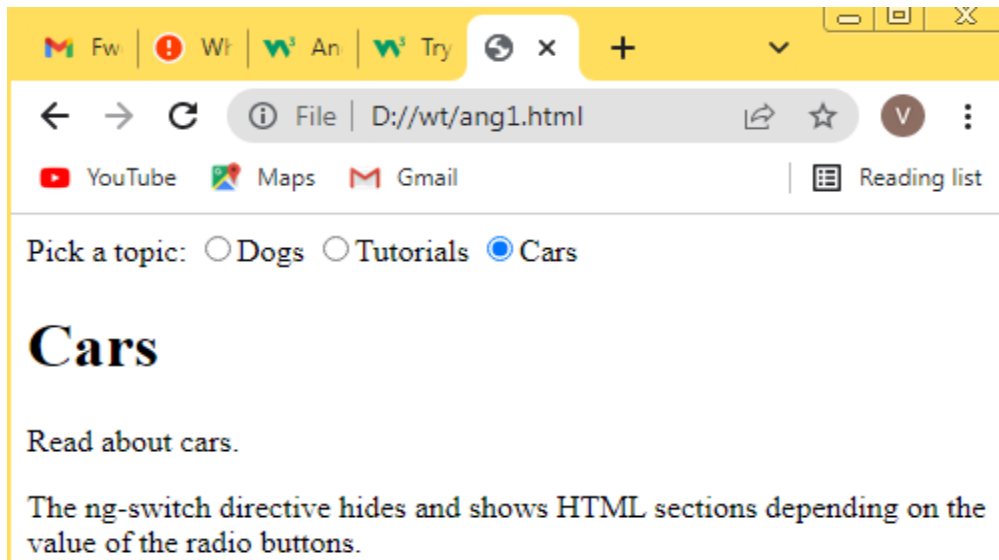
<div ng-switch="myVar">
  <div ng-switch-when="dogs">
    <h1>Dogs</h1>
    <p>Welcome to a world of dogs.</p>
  </div>
  <div ng-switch-when="tuts">
    <h1>Tutorials</h1>
    <p>Learn from examples.</p>
  </div>
  <div ng-switch-when="cars">
    <h1>Cars</h1>
    <p>Read about cars.</p>
  </div>
</div>
```

```
</div>
</div>
```

<p>The ng-switch directive hides and shows HTML sections depending on the value of the radio buttons.</p>

```
</body>
</html>
```

The value of myVar will be either **dogs**, **tuts**, or **cars**.



### Selectbox:

Bind select boxes to your application with the **ng-model** directive.

The property defined in the **ng-model** attribute will have the value of the selected option in the selectbox.

### Example

Display some text, based on the value of the selected option:

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body ng-app="">

<form>
  Select a topic:
  <select ng-model="myVar">
    <option value="">
    <option value="dogs">Dogs
    <option value="tuts">Tutorials
```

```

    <option value="cars">Cars
  </select>
</form>

<div ng-switch="myVar">
  <div ng-switch-when="dogs">
    <h1>Dogs</h1>
    <p>Welcome to a world of dogs.</p>
  </div>
  <div ng-switch-when="tuts">
    <h1>Tutorials</h1>
    <p>Learn from examples.</p>
  </div>
  <div ng-switch-when="cars">
    <h1>Cars</h1>
    <p>Read about cars.</p>
  </div>
</div>

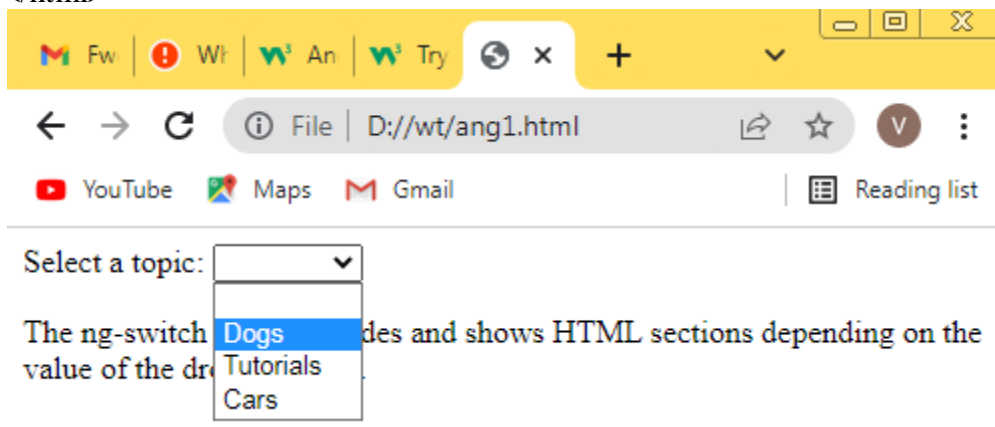
```

<p>The ng-switch directive hides and shows HTML sections depending on the value of the dropdown list.</p>

```

</body>
</html>

```



The value of myVar will be either **dogs**, **tuts**, or **cars**.

### An AngularJS Form Example

First Name:

Last Name:

**RESET**

```
form = { "firstName": "John", "lastName": "Doe" }
```

```
master = { "firstName": "John", "lastName": "Doe" }
```

**Application Code:**

```
<!DOCTYPE html>
<html lang="en">
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

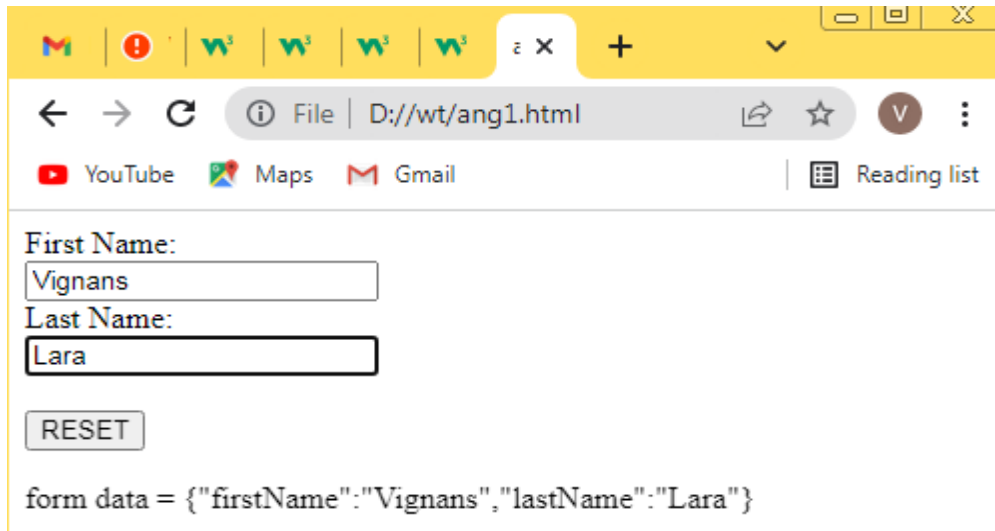
<div ng-app="myApp" ng-controller="formCtrl">
  <form novalidate>
    First Name:<br>
    <input type="text" ng-model="user.firstName"><br>
    Last Name:<br>
    <input type="text" ng-model="user.lastName">
    <br><br>
    <button ng-click="reset()">RESET</button>
  </form>
  <p>form data = { { user } } </p>

</div>

<script>
var app = angular.module('myApp', []);
app.controller('formCtrl', function($scope) {

  $scope.reset = function() {
    $scope.user = angular.copy($scope.master);
  };
  $scope.reset();
});
</script>

</body>
</html>
```



First Name:

Last Name:

form data = {"firstName":"Vignans","lastName":"Lara"}

### Example Explained

The **ng-app** directive defines the AngularJS application.

The **ng-controller** directive defines the application controller.

The **ng-model** directive binds two input elements to the **user** object in the model.

The **formCtrl** controller sets initial values to the **master** object, and defines the **reset()** method.

The **ng-click** directive invokes the **reset()** method, only if the button is clicked.

### AngularJS Form Validation

AngularJS can validate input data.

#### Form Validation

AngularJS offers client-side form validation.

AngularJS monitors the state of the form and input fields (input, textarea, select), and lets you notify the user about the current state.

AngularJS also holds information about whether they have been touched, or modified, or not.

Client-side validation cannot alone secure user input. Server side validation is also necessary.

#### Required

Use the HTML5 attribute **required** to specify that the input field must be filled out:

#### Example



The input field is required:

```
<form name="myForm">
  <input name="myInput" ng-model="myInput" required>
</form>
```

```
<p>The input's valid state is:</p>
<h1>{{ myForm.myInput.$valid }}</h1>
```

### E-mail:

Use the HTML5 type **email** to specify that the value must be an e-mail:

### Example

The input field has to be an e-mail:

```
<form name="myForm">
  <input name="myInput" ng-model="myInput" type="email">
</form>
```

```
<p>The input's valid state is:</p>
<h1>{{ myForm.myInput.$valid }}</h1>
```

### Form State and Input State:

AngularJS is constantly updating the state of both the form and the input fields.

Input fields have the following states:

- **\$untouched** The field has not been touched yet
- **\$touched** The field has been touched
- **\$pristine** The field has not been modified yet
- **\$dirty** The field has been modified
- **\$invalid** The field content is not valid
- **\$valid** The field content is valid

They are all properties of the input field, and are either **true** or **false**.

Forms have the following states:

- **\$pristine** No fields have been modified yet
- **\$dirty** One or more have been modified
- **\$invalid** The form content is not valid
- **\$valid** The form content is valid
- **\$submitted** The form is submitted

They are all properties of the form, and are either **true** or **false**.

You can use these states to show meaningful messages to the user. Example, if a field is required, and the user leaves it blank, you should give the user a warning:

### Example

Show an error message if the field has been touched AND is empty:

```
<input name="myName" ng-model="myName" required>
<span ng-show="myForm.myName.$touched && myForm.myName.$invalid">The name is
required.</span>
```

### Validation Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>
```

```
<h2>Validation Example</h2>
```

```
<form ng-app="myApp" ng-controller="validateCtrl"
name="myForm" novalidate>
```

```
<p>Username:<br>
<input type="text" name="user" ng-model="user" required>
<span style="color:red" ng-show="myForm.user.$dirty && myForm.user.$invalid">
<span ng-show="myForm.user.$error.required">Username is required.</span>
</span>
</p>
```

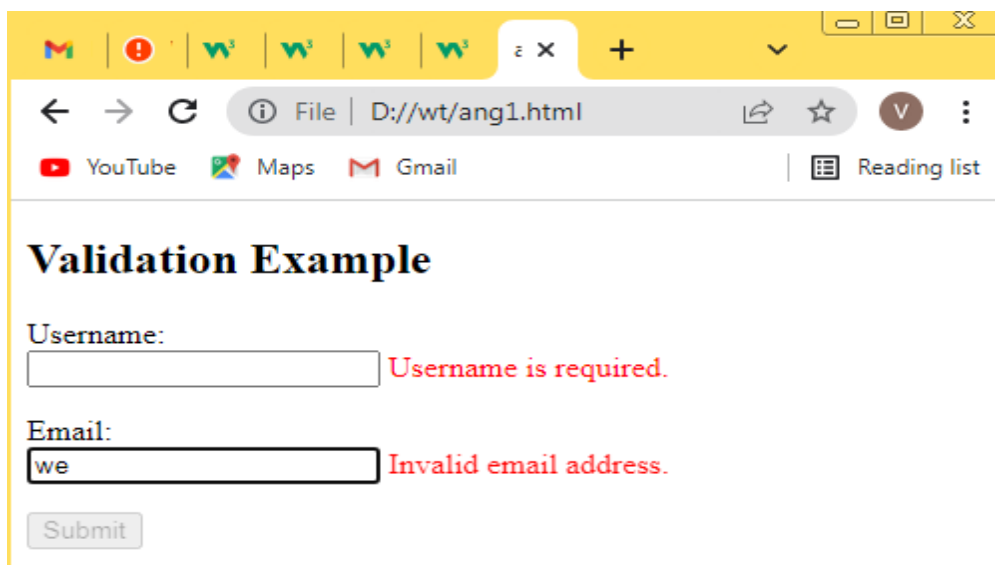
```
<p>Email:<br>
<input type="email" name="email" ng-model="email" required>
<span style="color:red" ng-show="myForm.email.$dirty && myForm.email.$invalid">
<span ng-show="myForm.email.$error.required">Email is required.</span>
<span ng-show="myForm.email.$error.email">Invalid email address.</span>
</span>
</p>
```

```
<p>
<input type="submit"
ng-disabled="myForm.user.$dirty && myForm.user.$invalid ||
myForm.email.$dirty && myForm.email.$invalid">
```

```
</p>  
</form>
```

```
<script>  
var app = angular.module('myApp', []);  
app.controller('validateCtrl', function($scope) {  
  $scope.user = 'John Doe';  
  $scope.email = 'john.doe@gmail.com';  
});  
</script>
```

```
</body>  
</html>
```



### Example Explanation:

The AngularJS directive **ng-model** binds the input elements to the model.

The model object has two properties: **user** and **email**.

Because of **ng-show**, the spans with `color:red` are displayed only when user or email is **\$dirty** and **\$invalid**.

## **Node.js Introduction**

- Node.js is an open source server environment
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the server

### **Why Node.js?**

#### **Node.js uses asynchronous programming!**

A common task for a web server can be to open a file on the server and return the content to the client. Here is how PHP or ASP handles a file request:

1. Sends the task to the computer's file system.
2. Waits while the file system opens and reads the file.
3. Returns the content to the client.
4. Ready to handle the next request.

Here is how Node.js handles a file request:

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

Node.js eliminates the waiting, and simply continues with the next request.

Node.js runs single-threaded, non-blocking, asynchronous programming, which is very memory efficient.

### **What Can Node.js Do?**

- Node.js can generate dynamic page content
- Node.js can create, open, read, write, delete, and close files on the server
- Node.js can collect form data
- Node.js can add, delete, modify data in your database

### **What is a Node.js File?**

- Node.js files contain tasks that will be executed on certain events
- A typical event is someone trying to access a port on the server
- Node.js files must be initiated on the server before having any effect
- Node.js files have extension ".js"

## **Advantages of Node.js**

**1. Cross-platform compatibility:** NodeJS is compatible with multiple platforms including Windows, Unix, Linux, Mac OS X, and mobile platforms. It can be bundled with the correct package into an executable that is entirely self-reliant.

## **2. The convenience of using one coding language**

NodeJs can be learned quickly by those who are already well-versed with JavaScript. With NodeJs, developers can use the same coding language for both front-end and back-end development. Therefore, JavaScript is the most preferred language for full-stack development.

Thus, it offers convenience to programmers as they don't need to switch between multiple coding languages and they also need to deal with fewer files.

## **3. V8 Engine**

Originally developed for Chrome, V8 Engine has now been adapted to suit web app development purposes. The V8 Engine is one of the most superior engines that can translate JavaScript to general machine coding language with the help of C++. Thus, the V8 Engine is ultimately helpful for servers and all machine language-based products.

## **4. Facilitates quick deployment and microservice development**

[NodeJS](#) is a lightweight tool that aids in the faster development and deployment of applications.

It also assists in the development of microservices. This is because NodeJs is capable of fast data processing and provides non-locking algorithms which are extremely beneficial for the development of microservices.

Additionally, NodeJs can also handle concurrent requests simultaneously. This is ultimately crucial for microservices as they need to constantly and quickly communicate with each other.

## **5. Scalable**

Most businesses nowadays prefer scalable software.

First of all, NodeJs takes care of concurrent requests. The second reason that makes NodeJs popular is that it has a cluster module that handles load balance for all running CPU cores.

The third and most interesting feature of NodeJs is its ability to split software horizontally. It accomplishes this with the help of child processes. This means that businesses can present different app versions to different target audiences which helps them address the personalization preferences of customers.

## **6. Commendable data processing ability**

Next, NodeJs employs an event-based software development approach in which there is no defined output order and the output is delivered solely based upon the user's inputs.

## **7. Active open-source community**

Being an open-source solution, NodeJs offers an extensive global community. The advantage of having a larger community is that developers can seek help from community members to get responses for their queries instantly. Community members share tools, modules, packages, and frameworks among each other completely free of cost.

## 8. Additional functionality of NPM

The official package ecosystem of NodeJs is the node package manager (NPM) with a dynamic repository of multiple tools and modules which are used by developers for app development. NPM is like a free marketplace for developers.

NPM can help in file upload management, download updates, and establish connectivity to MySQL databases.

## 9. Advanced hosting ability of NodeJs

NodeJs reduces the number of servers needed to host the application and ultimately cuts down the page load time by 50%.

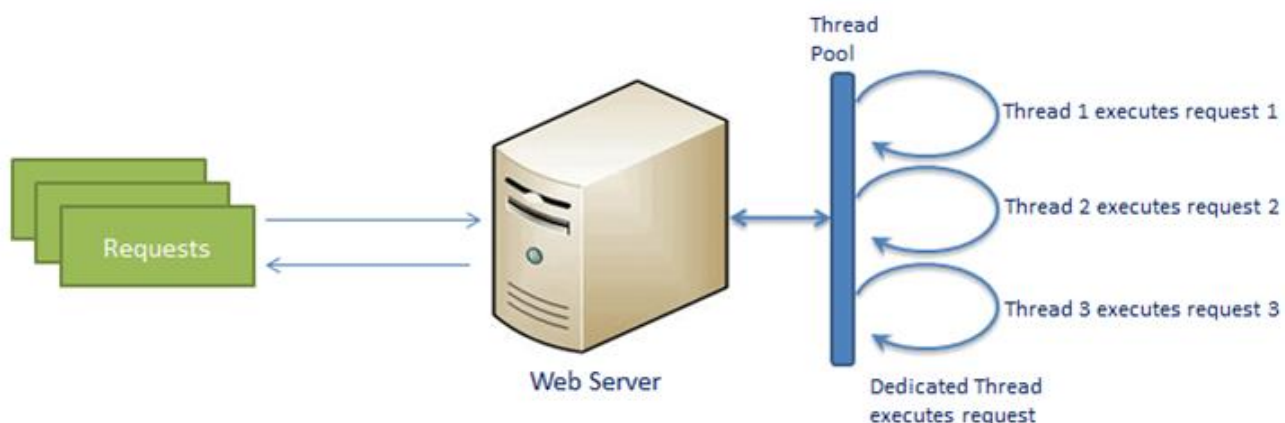
## 10. Fast data streaming

When data travels as different streams, processing them consumes a lot of time. So, [NodeJS](#) saves the time that goes into processing data by processing a file simultaneously while it is being uploaded.

### Node.js Process Model

#### Traditional Web Server Model

In the traditional web server model, each request is handled by a dedicated thread from the thread pool. If no thread is available in the thread pool at any point of time then the request waits till the next available thread. Dedicated thread executes a particular request and does not return to thread pool until it completes the execution and returns a response.



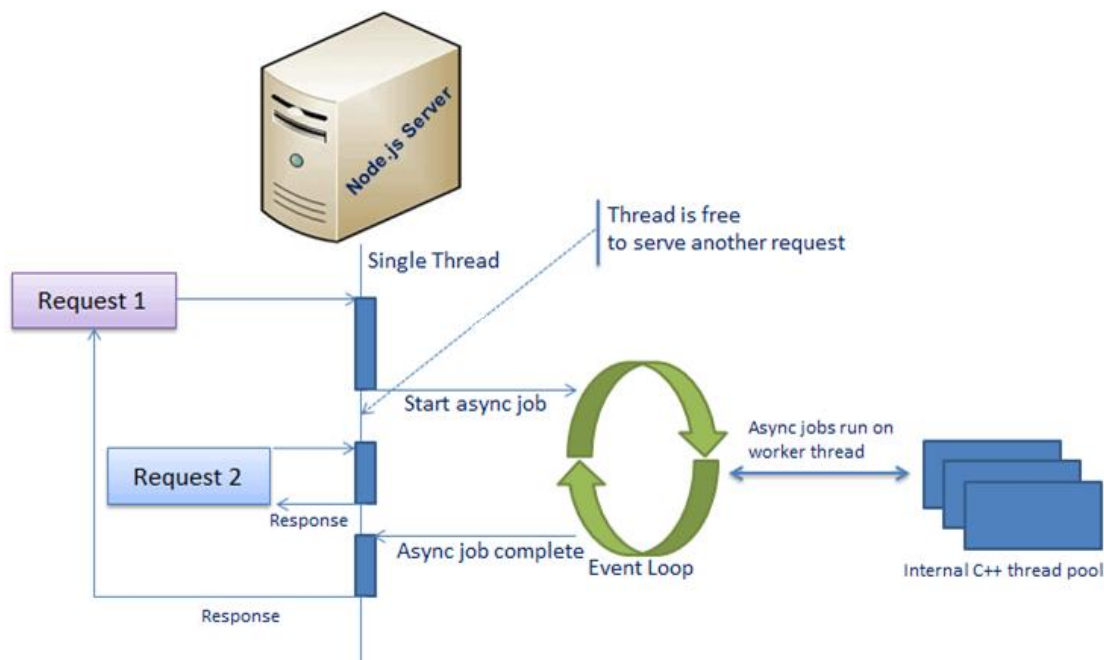
#### Traditional Web Server Model

## Node.js Process Model

Node.js processes user requests differently when compared to a traditional web server model. Node.js runs in a single process and the application code runs in a single thread and thereby needs less resources than other platforms. All the user requests to your web application will be handled by a single thread and all the I/O work or long running job is performed asynchronously for a particular request. So, this single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response.

An event loop is constantly watching for the events to be raised for an asynchronous job and executing callback function when the job completes. Internally, Node.js uses [libev](#) for the event loop which in turn uses internal C++ thread pool to provide asynchronous I/O.

The following figure illustrates asynchronous web server model using Node.js.



## Node.js Process Model

Node.js process model increases the performance and scalability with a few caveats. Node.js is not fit for an application which performs CPU-intensive operations like image processing or other heavy computation work because it takes time to process a request and thereby blocks the single thread.

## Node.js Modules

### What is a Module in Node.js?

Consider modules to be the same as JavaScript libraries.

A set of functions you want to include in your application.

## Built-in Modules

Node.js has a set of built-in modules which you can use without any further installation.

## Include Modules

To include a module, use the `require()` function with the name of the module:

```
var http = require('http');
```

Now your application has access to the HTTP module, and is able to create a server:

```
http.createServer(function (req, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
  res.end('Hello World!');  
}).listen(8080);
```

## Create Your Own Modules

You can create your own modules, and easily include them in your applications.

The following example creates a module that returns a date and time object:

### Example

Create a module that returns the current date and time:

```
exports.myDateTime = function ()  
  return Date();  
};
```

Use the `exports` keyword to make properties and methods available outside the module file.

Save the code above in a file called "myfirstmodule.js"

## Include Your Own Module

Now you can include and use the module in any of your Node.js files.

### Example

Use the module "myfirstmodule" in a Node.js file:

```
var http = require('http');  
var dt = require('./myfirstmodule');
```



```
http.createServer(function (req, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
  res.write("The date and time are currently: " + dt.myDateTime());  
  res.end();  
}).listen(8080);
```

Notice that we use `./` to locate the module, that means that the module is located in the same folder as the Node.js file.

Save the code above in a file called "demo\_module.js", and initiate the file:

#### Initiate demo\_module.js:

```
C:\Users\Your Name>node demo_module.js
```

If you have followed the same steps on your computer, you will see the same result as the example: <http://localhost:8080>

### Node.js File System Module

#### Node.js as a File Server

The Node.js file system module allows you to work with the file system on your computer.

To include the File System module, use the `require()` method:

```
var fs = require('fs');
```

Common use for the File System module:

- Read files
- Create files
- Update files
- Delete files
- Rename files

#### Read Files

The `fs.readFile()` method is used to read files on your computer.

Assume we have the following HTML file (located in the same folder as Node.js):

#### demofile1.html

```
<html>  
<body>
```

```
<h1>My Header</h1>
<p>My paragraph.</p>
</body>
</html>
```

**Create a Node.js file that reads the HTML file, and return the content:**

#### Example

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  fs.readFile('demofile1.html', function(err, data) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    return res.end();
  });
}).listen(8080);
```

Save the code above in a file called "demo\_readfile.js", and initiate the file:

#### Initiate demo\_readfile.js:

```
C:\Users\Your Name>node demo_readfile.js
```

If you have followed the same steps on your computer, you will see the same result as the example: <http://localhost:8080>

## Create Files

The File System module has methods for creating new files:

- `fs.appendFile()`
- `fs.open()`
- `fs.writeFile()`

The `fs.appendFile()` method appends specified content to a file. If the file does not exist, the file will be created:

#### Example

Create a new file using the `appendFile()` method:

```
var fs = require('fs');

fs.appendFile('mynewfile1.txt', 'Hello content!', function (err) {
```

```
if (err) throw err;  
console.log('Saved!');  
});
```

The **fs.open()** method takes a "flag" as the second argument, if the flag is "w" for "writing", the specified file is opened for writing. If the file does not exist, an empty file is created:

### Example

Create a new, empty file using the open() method:

```
var fs = require('fs');  
  
fs.open('mynewfile2.txt', 'w', function (err, file) {  
  if (err) throw err;  
  console.log('Saved!');  
});
```

The **fs.writeFile()** method replaces the specified file and content if it exists. If the file does not exist, a new file, containing the specified content, will be created:

### Example

Create a new file using the writeFile() method:

```
var fs = require('fs');  
  
fs.writeFile('mynewfile3.txt', 'Hello content!', function (err) {  
  if (err) throw err;  
  console.log('Saved!');  
});
```

### Update Files

The File System module has methods for updating files:

- **fs.appendFile()**
- **fs.writeFile()**

The **fs.appendFile()** method appends the specified content at the end of the specified file:

### Example

Append "This is my text." to the end of the file "mynewfile1.txt":

```
var fs = require('fs');

fs.appendFile('mynewfile1.txt', ' This is my text.', function (err) {
  if (err) throw err;
  console.log('Updated!');
});
```

The **fs.writeFile()** method replaces the specified file and content:

### Example

**Replace the content of the file "mynewfile3.txt":**

```
var fs = require('fs');

fs.writeFile('mynewfile3.txt', 'This is my text', function (err) {
  if (err) throw err;
  console.log('Replaced!');
});
```

### Delete Files

To delete a file with the File System module, use the **fs.unlink()** method.

The **fs.unlink()** method deletes the specified file:

### Example

**Delete "mynewfile2.txt":**

```
var fs = require('fs');

fs.unlink('mynewfile2.txt', function (err) {
  if (err) throw err;
  console.log('File deleted!');
});
```

### Rename Files

To rename a file with the File System module, use the **fs.rename()** method.

The **fs.rename()** method renames the specified file:

### Example

**Rename "mynewfile1.txt" to "myrenamedfile.txt":**

```
var fs = require('fs');
fs.rename('mynewfile1.txt', 'myrenamedfile.txt', function (err) {
  if (err) throw err;
  console.log('File Renamed!');
});
```

## Upload Files

You can also use Node.js to upload files to your computer.

## Node.js URL Module

### The Built-in URL Module

The URL module splits up a web address into readable parts.

To include the URL module, use the `require()` method:

```
var url = require('url');
```

Parse an address with the `url.parse()` method, and it will return a URL object with each part of the address as properties:

### Example

#### Split a web address into readable parts:

```
var url = require('url');
var adr = 'http://localhost:8080/default.htm?year=2017&month=february';
var q = url.parse(adr, true);
```

```
console.log(q.host); //returns 'localhost:8080'
console.log(q.pathname); //returns '/default.htm'
console.log(q.search); //returns '?year=2017&month=february'
```

```
var qdata = q.query; //returns an object: { year: 2017, month: 'february' }
console.log(qdata.month); //returns 'february'
```

## Node.js File Server

Now we know how to parse the query string, and in the previous chapter we learned how to make Node.js behave as a file server. Let us combine the two, and serve the file requested by the client.

Create two html files and save them in the same folder as your node.js files.

**summer.html**

```
<!DOCTYPE html>
<html>
<body>
<h1>Summer</h1>
<p>I love the sun!</p>
</body>
</html>
```

**winter.html**

```
<!DOCTYPE html>
<html>
<body>
<h1>Winter</h1>
<p>I love the snow!</p>
</body>
</html>
```

Create a Node.js file that opens the requested file and returns the content to the client. If anything goes wrong, throw a 404 error:

**demo\_fileserver.js:**

```
var http = require('http');
var url = require('url');
var fs = require('fs');

http.createServer(function (req, res) {
  var q = url.parse(req.url, true);
  var filename = "." + q.pathname;
  fs.readFile(filename, function(err, data) {
    if (err) {
      res.writeHead(404, {'Content-Type': 'text/html'});
      return res.end("404 Not Found");
    }
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    return res.end();
  });
}).listen(8080);
```

Remember to initiate the file:

**Initiate demo\_fileserver.js:**

```
C:\Users\Your Name>node demo_fileserver.js
```

If you have followed the same steps on your computer, you should see two different results when opening these two addresses:

<http://localhost:8080/summer.html>

Will produce this result:

Summer

I love the sun!

<http://localhost:8080/winter.html>

Will produce this result:

Winter

I love the snow!

## **Node.js Events**

Node.js is perfect for event-driven applications.

### **Events in Node.js**

Every action on a computer is an event. Like when a connection is made or a file is opened.

Objects in Node.js can fire events, like the readStream object fires events when opening and closing a file:

### **Example**

```
var fs = require('fs');
var rs = fs.createReadStream('./demoFile.txt');
rs.on('open', function () {
  console.log("The file is open");
});
```

### **Events Module**

Node.js has a built-in module, called "Events", where you can create-, fire-, and listen for- your own events.

To include the built-in Events module use the `require()` method. In addition, all event properties and methods are an instance of an EventEmitter object. To be able to access these properties and methods, create an EventEmitter object:

```
var events = require('events');  
var EventEmitter = new events.EventEmitter();
```

### The EventEmitter Object

You can assign event handlers to your own events with the EventEmitter object.

In the example below we have created a function that will be executed when a "scream" event is fired.

To fire an event, use the `emit()` method.

#### Example

```
var events = require('events');  
var EventEmitter = new events.EventEmitter();
```

//Create an event handler:

```
var myEventHandler = function () {  
  console.log('I hear a scream!');  
}
```

//Assign the event handler to an event:

```
eventEmitter.on('scream', myEventHandler);
```

//Fire the 'scream' event:

```
eventEmitter.emit('scream');
```