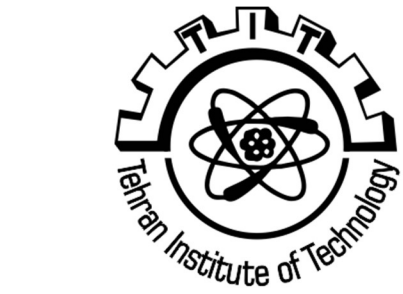


JAVASCRIPT

Tehran Institute of Technology

Zahra Mansoori
z.mansoori@gmail.com
http://tarsimm.com

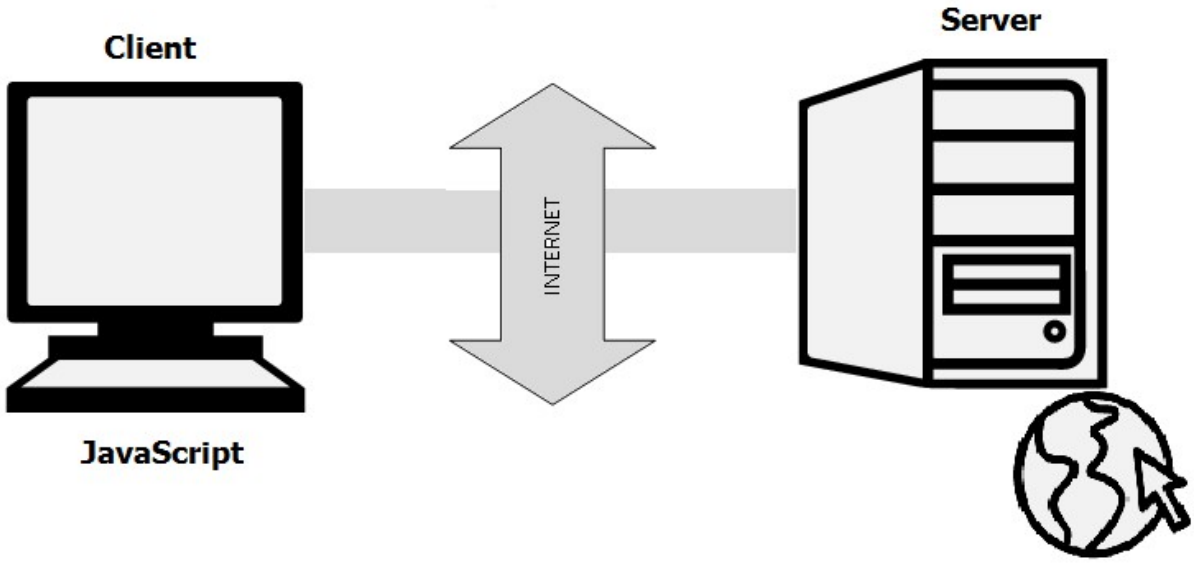
Contents

1. Introduction

JavaScript is a very powerful **client-side scripting language**. JavaScript is used mainly for enhancing the interaction of a user with the webpage. In other words, you can make your webpage more lively and interactive, with the help of JavaScript.

1.1. What is client-side scripting?

Client-end scripts are embedded in a website's HTML markup code, which is housed on the server in a language that is compatible with, or compiled to communicate with, the browser. The browser temporarily downloads that code, and then, apart from the server, processes it. If it needs to request additional information in response to user clicks, mouse-overs, etc. (called "events"), a request is sent back to the server.



Client-side scripting is always evolving—it is growing simpler, more nimble, and easier to use. As a result, sites are faster, more efficient, and less work is left up to the server.

1.2. <Script> Tag

JavaScript can be implemented using JavaScript statements that are placed within the <script>... </script> HTML tags in a web page.

You can place the <script> tags, containing your JavaScript, anywhere within you web page, but it is normally recommended that you should keep it within the <head> tags.

The <script> tag alerts the browser program to start interpreting all the text between these tags as a script. A simple syntax of your JavaScript will appear as follows.

```
<script ...>
        JavaScript code
</script>
```

The script tag takes two important attributes:

- **Language:** This attribute specifies what **scripting language** you are using. Typically, its value will be javascript. Although recent versions of HTML (and XHTML, its successor) have phased out the use of this attribute.
- **Type:** This attribute is what is now recommended to indicate the scripting language in use and its value should be set to "text/javascript".

So your JavaScript syntax will look as follows:

```
<script language="javascript" type="text/javascript">
        JavaScript code
</script>
```

1.3. First Javascript sample

Let us take a sample example to print out "Hello World". We added an optional HTML comment that surrounds our JavaScript code.

We call a function **document.write,** which writes a string into our HTML document.

Example. 1: Print "Hello World" in the screen

```
<html>
    <body>
        <script language="javascript" type="text/javascript">
            document.write ("Hello World!")
        </script>
    </body>
</html>
```

hello world

Hint:

1. JavaScript ignores **spaces**, **tabs**, and **newlines** that appear in JavaScript programs.
2. Simple statements in JavaScript are generally followed by a **semicolon character**, just as they are in C, C++, and Java. JavaScript, however, **allows you to omit this semicolon** if each of your statements are placed on a separate line.
3. JavaScript is a **case-sensitive language**. This means that the language keywords, variables, function names, and any other identifiers **must always be typed with a consistent capitalization of letters**.

1.4. Comments in JavaScript

JavaScript supports both C-style and C++-style comments. Thus:

- Any text between a `//` and the end of a line is treated as a comment and is ignored by JavaScript.
- Any text between the characters `/*` and `*/` is treated as a comment. This may span multiple lines.

Example.2: Comments in Javascript

```
<script language="javascript" type="text/javascript">
        <!--
        // This is a comment. It is similar to comments in C++
        /*
        * This is a multiline comment in JavaScript
        * It is very similar to comments in C Programming
        */
        //-->
</script>
```

1.5. Warning for Non-JavaScript Browsers

If you have to do something important using JavaScript, then you can display a warning message to the user using <noscript> tags.

We added an optional HTML comment that surrounds our JavaScript code. **This is to save our code from a browser that does not support JavaScript**. The comment ends with a "//-- >". Here "//" signifies a comment in JavaScript, so we add that to prevent a browser from reading the end of the HTML comment as a piece of JavaScript code.

You can add a noscript block immediately after the script block as follows:

Example.3: Using noscript tag

```
<body>
        <script language="javascript" type="text/javascript">
                <!--
                document.write ("Hello World!")
                //-->
        </script>
        <noscript>
                Sorry...JavaScript is needed to go ahead.
        </noscript>
</body>
```

1.6. JavaScript Debugging

A debugging tool is essential for JavaScript development. Firefox provides a debugger via the Firebug extension; Safari and Chrome provide built-in consoles.

console.log()	for sending general log messages
console.dir()	for logging a browseable object
console.warn()	for logging warnings
console.error()	for logging error messages

2. Placement

There is a flexibility given to include JavaScript code anywhere in an HTML document. However, the most preferred ways to include JavaScript in an HTML file are as follows:

- Script in `<head>...</head>` section
- Script in `<body>...</body>` section
- Script in `<body>...</body>` and `<head>...</head>` sections
- Script in an external file and then include in `<head>...</head>` section

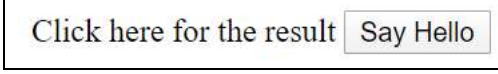
In the following section, we will see how we can place JavaScript in an HTML file in different ways.

2.1. JavaScript in <head>...</head> Section

If you want to have a script run on some event, such as when a user clicks somewhere, then you will place that script in the head as follows.

Example.4: JavaScript into <head> section

```
<html>
	<head>
		<script type="text/javascript">
			<!--
			function sayHello() {
				alert("Hello World")
			}
			//-->
		</script>
	</head>
	<body>
		Click here for the result
		<input type="button" onclick="sayHello()" value="Say Hello" />
	</body>
</html>
```

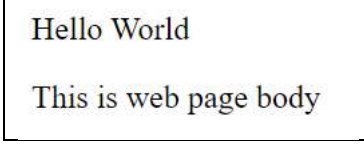


2.2. JavaScript in <body>...</body> Section

If you need a script to run as the page loads so that the script generates content in the page, then the script goes in the <body> portion of the document. In this case, you would not have any function defined using JavaScript. Look at the following code:

Example.5: JavaScript in <body> section

```
<html>
	<head>
	</head>
	<body>
		<script type="text/javascript">
			document.write("Hello World")
		</script>
		<p>This is web page body </p>
	</body>
</html>
```



2.3. JavaScript in <body> and <head> Sections

You can put your JavaScript code in <head> and <body> section altogether as follows.

Example.6: JavaScript in <body> and <head> Sections

```
<html>
	<head>
		<script type="text/javascript">
			function sayHello() {
			alert("Hello World")
			}
		</script>
	</head>
	<body>
		<script type="text/javascript">
			document.write("Hello World")
		</script>
		<input  type="button"  onclick="sayHello()"  value="Say
		Hello" />
	</body>
</html>
```

2.4. JavaScript in External File

As you begin to work more extensively with JavaScript, you will be likely to find that there are cases where you are reusing identical JavaScript code on multiple pages of a site.

You are not restricted to be maintaining identical code in multiple HTML files. The **script** tag provides a mechanism to allow you to store JavaScript in an external file and then include it into your HTML files.

Here is an example to show how you can include an external JavaScript file in your HTML code using **script** tag and its **src** attribute.

Example.7: JavaScript in External File

Example7.html

```
<html>
	<head>
		<script type="text/javascript" src="myscript.js"
		></script>
	</head>
	<body>
		<input  type="button"  onclick="sayHello()"  value="Say
		Hello" />
	</body>
</html>
```

myscript.js

```
function sayHello() {
	alert("Hello World")
}
```

3. Variables

3.1. JavaScript Datatypes

One of the most fundamental characteristics of a programming language is the set of data types it supports. These are the type of values that can be represented and manipulated in a programming language.

JavaScript allows you to work with three primitive data types:

1. **Numbers**, e.g., 123, 120.50 etc.
2. **Strings** of text, e.g. "This text string" etc.
3. **Boolean**, e.g. true or false

JavaScript also defines two trivial data types, **null** and **undefined**, each of which defines only a single value. In addition to these primitive data types, JavaScript supports a composite data type known as **object**. We will cover objects in detail in a separate chapter.

Hint: Java does not make a distinction between integer values and floatingpoint values. All numbers in JavaScript are represented as floating-point values. JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard.

3.2. JavaScript Variables

Like many other programming languages, JavaScript has variables. Variables can be thought of as named containers. You can place data into these containers and then refer to the data simply by naming the container.

Before you use a variable in a JavaScript program, you must declare it. Variables are declared with the **var** keyword as follows.

```
<script type="text/javascript">
	<!--
	var money;
	var name;
	//-->
</script>
```

You can also declare multiple variables with the same **var** keyword as follows:

```
<script type="text/javascript">
	<!--
	var money, name;
	//-->
</script>
```

Storing a value in a variable is called **variable initialization**. You can do variable initialization at the time of variable creation or at a later point in time when you need that variable.

For instance, you might create a variable named money and assign the value 2000.50 to it later. For another variable, you can assign a value at the time of initialization as follows.

```
<script type="text/javascript">
	var name = "Ali";
	var money;
	money = 2000.50;
</script>
```

Hint: Use the **var** keyword only for declaration or initialization, once for the life of any variable name in a document. You should not re-declare same variable twice.

JavaScript is **untyped** language. This means that a JavaScript variable can hold a value of any data type.

3.3. JavaScript Variable Scope

The scope of a variable is the region of your program in which it is defined. JavaScript variables have only two scopes.

- **Global Variables**: A global variable has global scope which means it can be defined anywhere in your JavaScript code.
- **Local Variables**: A local variable will be visible only within a function where it is defined. Function parameters are always local to that function.

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function parameter with the same name as a global variable, you effectively hide the global variable. Look into the following example.

Example. 8: Variable scope

```
<script type="text/javascript">
	var myVar = "global"; // Declare a global variable
	function checkscope( ) {
		var myVar = "local"; // Declare a local variable
		document.write(myVar);
	}
</script>
```

local

3.4. JavaScript Variable Names

While naming your variables in JavaScript, keep the following rules in mind.

- You should not use any of the JavaScript reserved keywords as a variable name. These keywords are mentioned in the next section. For example, **break** or **boolean** variable names are not valid.
- JavaScript variable names should not start with a numeral (0-9). They must begin with a letter or an underscore character. For example, *123test* is an invalid variable name but *_123test* is a valid one.
- JavaScript variable names are **case-sensitive**. For example, *Name* and *name* are two different variables.

3.5. JavaScript Reserved Words

A list of all the reserved words in JavaScript are given in the following table. They cannot be used as JavaScript **variables**, **functions**, **methods**, **loop labels**, or any object names.

abstract	else	Instanceof	switch
boolean	enum	int	synchronized
break	export	interface	this
byte	extends	long	throw
case	false	native	throws
catch	final	new	transient
char	finally	null	true
class	float	package	try
const	for	private	typeof
continue	function	protected	var
debugger	goto	public	void
default	if	return	volatile
delete	implements	short	while
do	import	static	with
double	in	super	

4. Operators

4.1. What is an Operator?

Let us take a simple expression **4 + 5** is equal to **9**. Here **4** and **5** are called **operands** and **'+'** is called the **operator**.

JavaScript supports the following types of operators:

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

Let us have a look at all the operators one by one.

4.2. Arithmetic Operators

JavaScript supports the following arithmetic operators:

Assume variable A holds 10 and variable B holds 20, then:

Operator and Description
+ (Addition) Adds two operands **Ex**: A + B will give 30
- (Subtraction) Subtracts the second operand from the first **Ex**: A - B will give -10
*** (Multiplication)** Multiply both operands **Ex**: A * B will give 200

/ (Division) Divide the numerator by the denominator **Ex**: B / A will give 2
% (Modulus) Outputs the remainder of an integer division **Ex**: B % A will give 0
-- (Decrement) Decreases an integer value by one **Ex**: A-- will give 9
++ (Increment) Increases an integer value by one **Ex**: A++ will give 11

Hint: Addition operator (**+**) works for Numeric as well as Strings. e.g. "a" + 10 will give "a10".

Example.9: arithmetic operators in JavaScript

```
<html>
	<body>
		<script type="text/javascript">
			var a = 33; var b = 10; var c = "Test";
			var linebreak = "<br />";

			document.write("a + b = ");
			result = a + b;
			document.write(result + linebreak);

			document.write("a - b = ");
			result = a - b;
			document.write(result + linebreak);

			document.write("a / b = ");
			result = a / b;
			document.write(result + linebreak);

			document.write("a % b = ");
			result = a % b;
			document.write(result + linebreak);

			document.write("a + b + c = ");
			result = a + b + c;
			document.write(result + linebreak);

			document.write("++a = ");
			result = ++a;
			document.write(result + linebreak);

			document.write("--b = ");
			result = --b;
			document.write(result + linebreak);
		</script>
	</body>
</html>
```

a + b = 43
a - b = 23
a / b = 3.3
a % b = 3
a + b + c = 43Test
++a = 34
--b = 9

4.3. Comparison Operators

JavaScript supports the following comparison operators:

Assume variable A holds 10 and variable B holds 20, then:

== (Equal) Checks if the value of two operands are equal or not, if yes, then the condition becomes true. **Ex**: (A == B) is not true.
!= (Not Equal) Checks if the value of two operands are equal or not, if the values are not equal, then the condition becomes true. **Ex**: (A != B) is true.
> (Greater than) Checks if the value of the left operand is greater than the value of the right operand, if yes, then the condition becomes true. **Ex**: (A > B) is not true.
< (Less than) Checks if the value of the left operand is less than the value of the right operand, if yes, then the condition becomes true. **Ex**: (A < B) is true.
>= (Greater than or Equal to) 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. **Ex**: (A >= B) is not true.
<= (Less than or Equal to) 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. **Ex**: (A <= B) is true.

4.4. Logical Operators

JavaScript supports the following logical operators:

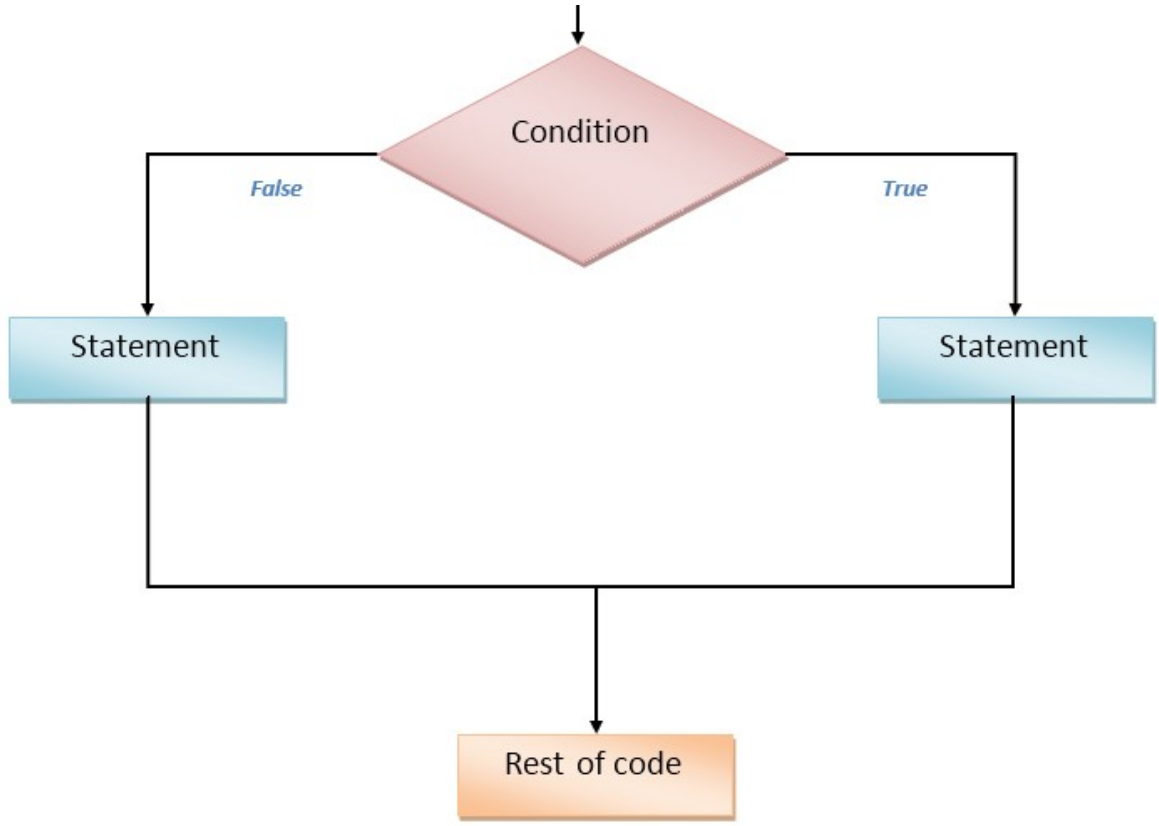
Assume variable A holds 10 and variable B holds 20, then:

&& (Logical AND) If both the operands are non-zero, then the condition becomes true. **Ex**: (A && B) is true.
|| (Logical OR) If any of the two operands are non-zero, then the condition becomes true. **Ex**: (A || B) is true.
! (Logical NOT) Reverses the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false. **Ex**: ! (A && B) is false.

4.5. Conditional Statements

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.



In JavaScript we have the following conditional statements:

- Use if to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false
- Use else if to specify a new condition to test, if the first condition is false
- Use switch to specify many alternative blocks of code to be executed

4.5.1. The if Statement

Use the if statement to specify a block of JavaScript code to be executed if a condition is true.

Syntax

```
if (condition) {
  //  block of code to be executed if the condition is true
}
```

4.5.2. The else Statement

Use the else statement to specify a block of code to be executed if the condition is false.

Syntax

```
if (hour < 18) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

4.5.3. The else if Statement

Use the else if statement to specify a new condition if the first condition is false.

Syntax

```
if (condition1) {
  //  block of code to be executed if condition1 is true
} else if (condition2) {
  //  block of code to be executed if the condition1 is false and condition2 is true
} else {
  //  block of code to be executed if the condition1 is false and condition2 is false
}
```

Example. 10: if else example

```
<html>
	<head>
		<script type="text/javascript">
			var one = prompt("Enter the first number");
			var two = prompt("Enter the second number");
			one = parseInt(one);
			two = parseInt(two);
			if (one == two)
				document.write(one + " is equal to " + two + ".");
			else if (one<two)
				document.write(one + " is less than " + two + ".");
			else
				document.write(one + " is greater than " + two + ".");
		</script>
	</head>
	<body>
	</body>
</html>
```

4.5.1. The switch Statement

Use the switch statement to select one of many code blocks to be executed.

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

Example. 11: switch example

```
<!DOCTYPE html>
<html>
	<body>

		<p id="demo"></p>

		<script>
			var day;
			switch (new Date().getDay()) {
			  case 0:
			    day = "Sunday";
			    break;
			  case 1:
			    day = "Monday";
			    break;
			  case 2:
			    day = "Tuesday";
			    break;
			  case 3:
			    day = "Wednesday";
			    break;
			  case 4:
			    day = "Thursday";
			    break;
			  case 5:
			    day = "Friday";
			    break;
			  case  6:
			    day = "Saturday";
			}
			document.getElementById("demo").innerHTML = "Today is " + day;
		</script>

	</body>
</html>
```

4.6. Bitwise Operators

JavaScript supports the following bitwise operators:

Assume:

- A holds 2 = (0x0010)
- B holds 3 = (0x0011)

then:

& (Bitwise AND)

It performs a Boolean AND operation on each bit of its integer arguments.

Ex: (A & B) is 2.

| (BitWise OR)

It performs a Boolean OR operation on each bit of its integer arguments.

Ex: (A | B) is 3.

^ (Bitwise XOR)

It performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both.

Ex: (A ^ B) is 1.

~ (Bitwise Not)

It is a unary operator and operates by reversing all the bits in the operand.

Ex: (~B) is -4.

<< (Left Shift)

It moves all the bits in its first operand to the left by the number of places specified in the second operand. New bits are filled with zeros.
Shifting a value left by one position is equivalent to multiplying it by 2, shifting two positions is equivalent to multiplying by 4, and so on.

Ex: (A << 1) is 4.

>> (Right Shift)

Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand.

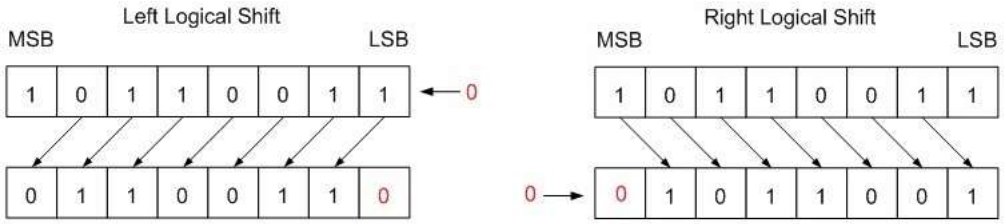
Ex: (A >> 1) is 1.

>>> (Right shift with Zero)

This operator is just like the >> operator, except that the bits shifted in on the left are always zero.

Ex: (A >>> 1) is 1.

Shift operations:



4.7. Assignment Operators

JavaScript supports the following assignment operators:

= (Simple Assignment) Assigns values from the right side operand to the left side operand **Ex**: C = A + B will assign the value of A + B into C
+= (Add and Assignment) It adds the right operand to the left operand and assigns the result to the left operand. **Ex**: C += A is equivalent to C = C + A
-= (Subtract and Assignment) It subtracts the right operand from the left operand and assigns the result to the left operand. **Ex**: C -= A is equivalent to C = C - A
***= (Multiply and Assignment)** It multiplies the right operand with the left operand and assigns the result to the left operand. **Ex**: C *= A is equivalent to C = C * A
/= (Divide and Assignment) It divides the left operand with the right operand and assigns the result to the left operand. Ex: C /= A is equivalent to C = C / A
%= (Modules and Assignment) It takes modulus using two operands and assigns the result to the left operand. **Ex**: C %= A is equivalent to C = C % A

Hint: Same logic applies to Bitwise operators, so they will become <<=, >>=, >>=, &=, |= and ^=.

4.8. Miscellaneous Operators

We will discuss two operators here that are quite useful in JavaScript: the **conditional operator (? :)** and the **typeof operator**.

4.7.1. 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.

? : (Conditional)
If Condition is true? Then value X : Otherwise value Y

Example.12: Conditional Operator

```
<html>
      <body>
            <script type="text/javascript">
                  <!--
                  var a = 10;
                  var b = 20;
                  var linebreak = "<br />";

                  document.write ("((a > b) ? 100 : 200) => ");
                  result = (a > b) ? 100 : 200;
                  document.write(result + linebreak);

                  document.write ("((a < b) ? 100 : 200) => ");
                  result = (a < b) ? 100 : 200;
                  document.write(result + linebreak);
                  //-->
            </script>
      </body>
</html>
```

((a > b) ? 100 : 200) => 200
((a < b) ? 100 : 200) => 100

4.8.2. typeof Operator

The **typeof** operator is a unary operator that is placed before its single operand, which can be of any type. Its value is a string indicating the data type of the operand.

The **typeof** operator evaluates to "**number**", "**string**", or "**boolean**" if its operand is a number, string, or boolean value and returns true or false based on the evaluation.

Here is a list of the return values for the **typeof** Operator.

Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"
Function	"function"
Undefined	"undefined"
Null	"object"

Example.13: typeof operator

```
<html>
	<body>
		<script type="text/javascript">
		<!--
			var a = 10;
			var b = "String";
			var linebreak = "<br />";

			result = (typeof b == "string" ? "B is String" :
			"B is Numeric");
			document.write("Result => ");
			document.write(result + linebreak);

			result = (typeof a == "string" ? "A is String" :
			"A is Numeric");
			document.write("Result => ");
			document.write(result + linebreak);
		//-->
		</script>
	</body>
</html>
```

Result => B is String
Result => A is Numeric

5. FOR-IN LOOP

The **"for...in"** loop is used to loop through an object's properties. As we have not discussed Objects yet, you may not feel comfortable with this loop. However, once you understand how objects behave in JavaScript, you will find this loop very useful.

Syntax of "for..in" loop

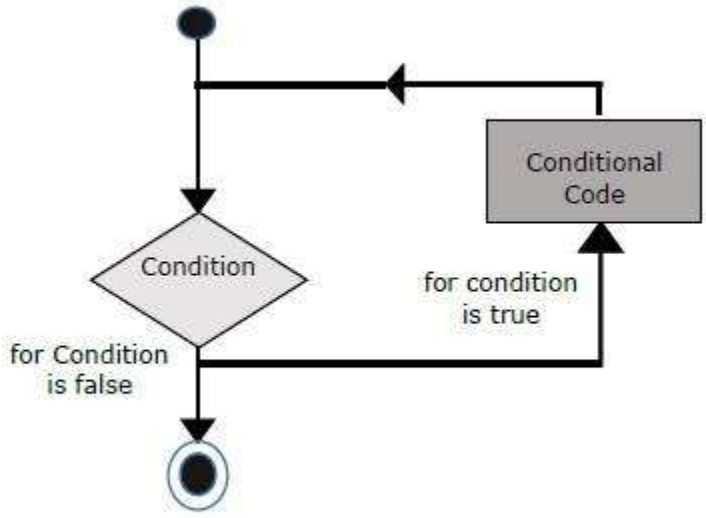
```
for (variable-name in object){
        statement or block to execute
}
```

In each iteration, one property from **object** is assigned to **variable-name** and this loop continues until all the properties of the object are exhausted.

Example. 14: Loop example

```
<body>
	<script type="text/javascript">
		var cars = ["BMW", "Volvo", "Saab", "Ford", "Fiat", "Audi"];
		var text = "";
		var i;
		for (i = 0; i < cars.length; i++) {
			text += cars[i] + "<br>";
		}
		document.write(text);
	</script>
</body>
```

BMW
Volvo
Saab
Ford
Fiat
Audi



Picture.1: Loop Functionality schematic

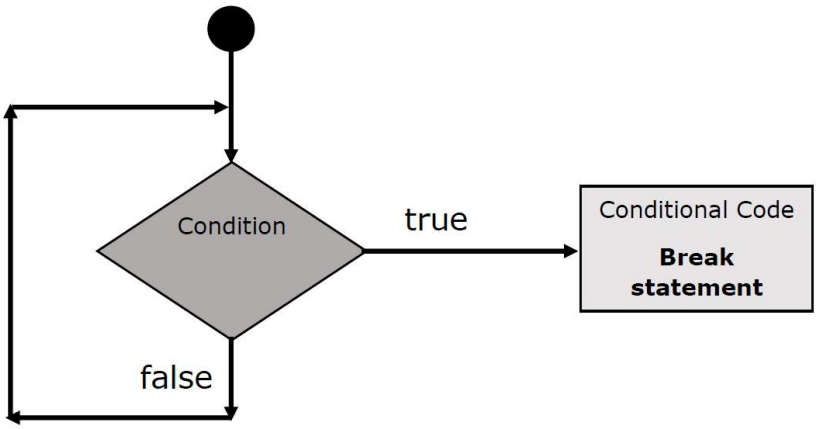
5.1. Loop control

JavaScript provides full control to handle loops and switch statements. There may be a situation when you need to come out of a loop without reaching at its bottom. There may also be a situation when you want to skip a part of your code block and start the next iteration of the look.

To handle all such situations, JavaScript provides **break** and **continue** statements. These statements are used to immediately come out of any loop or to start the next iteration of any loop respectively.

5.2. The break Statement

The **break** statement, which was briefly introduced with the switch statement, is used to exit a loop early, breaking out of the enclosing curly braces.



Picture.2: chart of a break statement

Example.15: break statement

```
<p id="demo"></P>
<script>
        var text = "";
        var i;
        for (i = 0; i < 10; i++) {
          if (i == 3) { break; }
          text += "The number is " + i + "<br>";
        }
        document.getElementById("demo").innerHTML = text;
</script>
```

The number is 0
The number is 1
The number is 2

5.3. The continue Statement

The **continue** statement tells the interpreter to immediately start the next iteration of the loop and skip the remaining code block. When a **continue** statement is encountered, the program flow moves to the loop check expression immediately and if the condition remains true, then it starts the next iteration, otherwise the control comes out of the loop.

Example.16: continue statement

```
<script>
        var text = "";
        var i;
        for (i = 0; i < 10; i++) {
          if (i === 3) { continue; }
          text += "The number is " + i + "<br>";
        }
        document.write(text);
</script>
```

The number is 0
The number is 1
The number is 2
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9

5.4. The While Loop

The while loop loops through a block of code as long as a specified condition is true.

Syntax of While

```
while (condition) {
  // code block to be executed
}
```

Example. 17: While Example

```
<!DOCTYPE html>
<html>
	<body>

		<h2>JavaScript While Loop</h2>

		<p id="demo"></p>

		<script>
			var text = "";
			var i = 0;
			while (i < 10) {
			  text += "<br>The number is " + i;
			  i++;
			}
			document.getElementById("demo").innerHTML = text;
		</script>

	</body>
</html>
```

JavaScript Do/While Loop

The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9

5.4. The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax of Do/While

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

Example. 18: Do/While Example

```
<!DOCTYPE html>
<html>
	<body>

		<h2>JavaScript Do/While Loop</h2>

		<p id="demo"></p>

		<script>
			var text = ""
			var i = 0;

			do {
			  text += "<br>The number is " + i;
			  i++;
			}
			while (i < 10);

			document.getElementById("demo").innerHTML = text;
		</script>

	</body>
</html>
```

JavaScript While Loop

The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9

6.Strings

The String object lets you work with a series of characters; it wraps Javascript's string primitive data type with a number of helper methods.

As JavaScript automatically converts between string primitives and String objects, you can call any of the helper methods of the String object on a string primitive.

Syntax

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

The string parameter is a series of characters that has been properly encoded.

Example. 19: String constructor

```
<html>
    <body>
        <script type="text/javascript">
            var str = new String( "This is string" );
        </script>
    </body>
</html>
```

6.1. Length

This property returns the number of characters in a string.

Syntax

```
string.length
```

Example. 20: String length

```
<html>
    <head>
        <title>JavaScript String length Property</title>
    </head>
    <body>
        <script type="text/javascript">
            var str = new String( "This is string" );
            document.write("str.length is:" + str.length);
        </script>
    </body>
</html>
```

str.length is:14

6.2. String Methods

Here is a list of the methods available in String object along with their description.

Method	*Description*
concat()	Combines the text of two strings and returns a new string.
indexOf()	Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found.
lastIndexOf()	Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found.
localeCompare()	Returns a number indicating whether a reference string comes before or after or is the same as the given string

	in sorted order.
match()	Used to match a regular expression against a string.
replace()	Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring.
search()	Executes the search for a match between a regular expression and a specified string.
map()	Creates a new array with the results of calling a provided function on every element in this array.
slice()	Extracts a section of a string and returns a new string.
split()	Splits a String object into an array of strings by separating the string into substrings.
substr()	Returns the characters in a string beginning at the specified location through the specified number of characters.
substring()	Returns the characters in a string between two indexes into the string.
toLocaleLowerCase()	The characters within a string are converted to lower case while respecting the current locale.
toLocaleUpperCase()	The characters within a string are converted to upper case while respecting the current locale.
toLowerCase()	Returns the calling string value converted to lower case.
toString()	Returns a string representing the specified object.
toUpperCase()	Returns the calling string value converted to uppercase.
valueOf()	Returns the primitive value of the specified object.

6.2.1. charAt()

charAt() is a method that returns the character from the specified index.

Characters in a string are indexed from left to right. The index of the first character is 0, and the index of the last character in a string, called stringName, is stringName.length – 1.

Syntax

```
string.charAt(index)
```

index: An integer between 0 and 1 less than the length of the string.

Example. 21: charAt Example

```
<html>
    <head>
        <title>JavaScript String charAt() Method</title>
    </head>
    <body>
        <script type="text/javascript">
            var str = new String( "This is string" );
            document.writeln("str.charAt(0) is:" + str.charAt(0));
            document.writeln("<br />str.charAt(1) is:" + str.charAt(1));
            document.writeln("<br />str.charAt(2) is:" + str.charAt(2));
            document.writeln("<br />str.charAt(3) is:" + str.charAt(3));
            document.writeln("<br />str.charAt(4) is:" + str.charAt(4));
            document.writeln("<br />str.charAt(5) is:" + str.charAt(5));
        </script>
    </body>
</html>
```

```
str.charAt(0) is:T
str.charAt(1) is:h
str.charAt(2) is:i
str.charAt(3) is:s
str.charAt(4) is:
str.charAt(5) is:i
```

6.2.2. contact()

This method adds two or more strings and returns a new single string.

Syntax

```
string.concat(string2, string3[, ..., stringN]);
```

string2...stringN: These are the strings to be concatenated

Example. 22: contact() example

```
<html>
    <head>
        <title>JavaScript String concat() Method</title>
    </head>
    <body>
        <script type="text/javascript">
            var str1 = new String( "This is string one" );
            var str2 = new String( "This is string two" );
            var str3 = str1.concat( str2 );
            document.write("Concatenated String :" + str3);
        </script>
    </body>
</html>
```

Concatenated String :This is string oneThis is string two

6.2.3. indexOf ()

This method returns the index within the calling String object of the first occurrence of the specified value, starting the search at from Index or -1 if the value is not found.

Syntax

```
string.indexOf(searchValue[, fromIndex])
```

- *searchValue: A string representing the value to search for*
- *fromIndex: The location within the calling string to start the search from. It can be any integer between 0 and the length of the string. The default value is 0*

Example. 23: indexOf() example

```
<html>
    <head>
        <title>JavaScript String indexOf() Method</title>
    </head>
    <body>
        <script type="text/javascript">
            var str1 = new String( "This is string one" );
            var index = str1.indexOf( "string" );
            document.write("indexOf found String :" + index );
            document.write("<br />");
            var index = str1.indexOf( "one" );
            document.write("indexOf found String :" + index );
        </script>
    </body>
</html>
```

indexOf found String :8
indexOf found String :15

6.2.4. lastindexOf ()

This method returns the index within the calling String object of the last occurrence of the specified value, starting the search at fromIndex or -1 if the value is not found.

Syntax

```
string.lastIndexOf(searchValue[, fromIndex])
```

- *searchValue: A string representing the value to search for*
- *fromIndex: The location within the calling string to start the search from. It can be any integer between 0 and the length of the string. The default value is 0*

Example. 24: lastindexOf() example

```
<html>
    <head>
        <title>JavaScript String lastIndexOf() Method</title>
    </head>
    <body>
        <script type="text/javascript">
            var str1 = new String( "This is string one and again string" );
            var index = str1.lastIndexOf( "string" );
            document.write("lastIndexOf found String :" + index );
            document.write("<br />");
            var index = str1.lastIndexOf( "one" );
            document.write("lastIndexOf found String :" + index );
        </script>
    </body>
</html>
```

lastIndexOf found String :29
lastIndexOf found String :15

6.2.5. replace ()

This method finds a match between a regular expression and a string, and replaces the matched substring with a new substring.

Syntax

```
string.replace(searchvalue, newvalue)
```

- *searchValue: The value, or regular expression, that will be replaced by the new value*
- *newvalue: The value to replace the search value with*

Example. 25: replace() example

```
<!DOCTYPE html>
<html>
	<body>
		<p>Click the button to replace "Microsoft" with "W3Schools" in the
		paragraph below:</p>
		<p id="demo">Visit Microsoft!</p>
		<button onclick="myFunction()">Try it</button>
		<script>
			function myFunction() {
			  var str = document.getElementById("demo").innerHTML;
			  var res = str.replace("Microsoft", "W3Schools");
			  document.getElementById("demo").innerHTML = res;
			}
		</script>
	</body>
</html>
```

6.2.6. slice()

This method extracts a section of a string and returns a new string.

Syntax

```
string.slice( beginslice [, endSlice] );
```

- *beginSlice : The zero-based index at which to begin extraction.*
- *endSlice : The zero-based index at which to end extraction. If omitted, slice extracts to the end of the string.*

Hint: If start is negative, slice uses it as a character index from the end of the string.

Example. 26: slice() example

```
<html>
	<head>
		<title>JavaScript String slice() Method</title>
	</head>
	<body>
		<script type="text/javascript">
			var str = "Apples are round, and apples are juicy.";
			var sliced = str.slice(3, -2);
			document.write( sliced );
		</script>
	</body>
</html>
```

les are round, and apples are juic

6.2.7. split()

This method splits a String object into an array of strings by separating the string into substrings.

Syntax

```
string.split([separator][, limit]);
```

- *separator : Specifies the character to use for separating the string. If separator is omitted, the array returned contains one element consisting of the entire string*
- *limit : Integer specifying a limit on the number of splits to be found*

Example. 27: split() example

```
<html>
	<head>
		<title>JavaScript String split() Method</title>
	</head>
	<body>
		<script type="text/javascript">
			var str = "Apples are round, and apples are juicy.";
			var splitted = str.split(" ", 3);
			document.write( splitted );
		</script>
	</body>
</html>
```

Apples,are,round,

6.2.8. substr()

This method returns the characters in a string beginning at the specified locationthrough the specified number of characters.

Syntax

```
string.substr(start[, length]);
```

- *start: Location at which to start extracting characters (an integer between 0 and one less than the length of the string).*
- *length : The number of characters to extract.*

Hint: If start is negative, substr uses it as a character index from the end of the string.

Example. 28: substr() example

```
<html>
    <head>
        <title>JavaScript String substr() Method</title>
    </head>
    <body>
        <script type="text/javascript">
            var str = "Apples are round, and apples are juicy.";
            document.write("(1,2): " + str.substr(1,2));
            document.write("<br />(-2,2): " + str.substr(-2,2));
            document.write("<br />(1): " + str.substr(1));
            document.write("<br />(-20, 2): " + str.substr(-20,2));
            document.write("<br />(20, 2): " + str.substr(20,2));
        </script>
    </body>
</html>
```

(1,2): pp
(-2,2): y.
(1): pples are round, and apples are juicy.
(-20, 2): nd
(20, 2): d

6.2.9. substring()

This method returns a subset of a String object.

Syntax

```
string.substring(indexA, [indexB])
```

- *indexA : An integer between 0 and one less than the length of the string*
- *indexB : (optional) An integer between 0 and the length of the string*

Example. 29: substring() example

```
<html>
	<head>
		<title>JavaScript String substring() Method</title>
	</head>
	<body>
		<script type="text/javascript">
			var str = "Apples are round, and apples are juicy.";
			document.write("(1,2): " + str.substring(1,2));
			document.write("<br />(0,10): " + str.substring(0, 10));
			document.write("<br />(5): " + str.substring(5));
		</script>
	</body>
</html>
```

(1,2): p
(0,10): Apples are
(5): s are round, and apples are juicy.

6.2.10. toLowerCase()

This method returns the calling string value converted to lowercase.

Syntax

```
string.toLowerCase( )
```

Example. 30: toLowerCase() example

```
<html>
	<head>
		<title>JavaScript String toLowerCase() Method</title>
	</head>
	<body>
		<script type="text/javascript">
			var str = "Apples are round, and Apples are Juicy.";
			document.write(str.toLowerCase( ));
		</script>
	</body>
</html>
```

apples are round, and apples are juicy.

6.2.11. toUpperCase()

This method returns the calling string value converted to uppercase.

Syntax

```
string.toUpperCase( )
```

Example. 31: toUpperCase() example

```
<html>
	<body>
		<script type="text/javascript">
			var str = "Apples are round, and Apples are Juicy.";
			document.write(str.toUpperCase( ));
		</script>
	</body>
</html>
```

APPLES ARE ROUND, AND APPLES ARE JUICY.

6.2.12. toString()

This method returns a string representing the specified object.

Syntax

```
string.toString( )
```

Example. 32: toString() example

```
<html>
    <head>
        <title>JavaScript String toString() Method</title>
    </head>
    <body>
        <script type="text/javascript">
            var str = "Apples are round, and Apples are Juicy.";
            document.write(str.toString( ));
        </script>
    </body>
</html>
```

Apples are round, and Apples are Juicy.

7. Arrays

The **Array** object lets you store multiple values in a single variable. It stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Syntax

```
var fruits = new Array( "apple", "orange", "mango" );
```

The **Array** parameter is a list of strings or integers. When you specify a single numeric parameter with the Array constructor, you specify the initial length of the array. The maximum length allowed for an array is 4,294,967,295.

You can create array by simply assigning values as follows:

```
var fruits = [ "apple", "orange", "mango" ];
```

You will use ordinal numbers to access and to set values inside an array as follows.

```
fruits[0] is the first element
fruits[1] is the second element
fruits[2] is the third element
```

7.1. Array Properties

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

Property	*Description*
constructor	Returns a reference to the array function that created the object.

index	The property represents the zero-based index of the match in the string
input	This property is only present in arrays created by regular expression matches.
length	Reflects the number of elements in an array.
prototype	The prototype property allows you to add properties and methods to an object.

7.1.1. constructor

Javascript array **constructor** property returns a reference to the array function that created the instance's prototype.

Example.33: Array constructor

```
<script type="text/javascript">
        var arr = new Array( 10, 20, 30 );
</script>
```

7.1.2. length

Javascript array **length** property returns an unsigned, 32-bit integer that specifies the number of elements in an array.

Syntax

```
array.length
```

Example.34: Array length

```
<script type="text/javascript">
        var arr = new Array( 10, 20, 30 );
        document.write("arr.length is:" + arr.length);
</script>
```

arr.length is:3

7.2. Array Methods

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

Method	*Description*
concat()	Returns a new array comprised of this array joined with other array(s) and/or value(s).
every()	Returns true if every element in this array satisfies the provided testing function.
filter()	Creates a new array with all of the elements of this array for which the provided filtering function returns true.
forEach()	Calls a function for each element in the array.
indexOf()	Returns the first (least) index of an element within the array equal to the specified value or - 1 if none is found.
join()	Joins all elements of an array into a string.
lastIndexOf()	Returns the last (greatest) index of an element within the array equal to the specified value or - 1 if none is found.
map()	Creates a new array with the results of calling a provided function on every element in this array.
pop()	Removes the last element from an array and returns that element.
push()	Adds one or more elements to the end of an array and returns the new length of the array.
reduce()	Apply a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value.
reduceRight()	Apply a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value.

reverse()	Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first.
shift()	Removes the first element from an array and returns that element.
slice()	Extracts a section of an array and returns a new array.
some()	Returns true if at least one element in this array satisfies the provided testing function.
toSource()	Represents the source code of an object
sort()	Sorts the elements of an array.
splice()	Adds and/or removes elements from an array.
toString()	Returns a string representing the array and its elements.
unshift()	Adds one or more elements to the front of an array and returns the new length of the array.

7.2.1. concat ()

Javascript array **concat()** method returns a new array comprised of this array joined with two or more arrays.

Syntax

```
array.concat(value1, value2, ..., valueN);
```

Example. 35: Array concat

```
<script type="text/javascript">
        var alpha = ["a", "b", "c"];
        var numeric = [1, 2, 3];
        var alphaNumeric = alpha.concat(numeric);
        document.write("alphaNumeric : " + alphaNumeric );
</script>
```

alphaNumeric : a,b,c,1,2,3

7.2.2. forEach ()

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

Example. 36.: forEach method

```
<script type="text/javascript">
        var fruits = ["apple", "orange", "cherry"];
        fruits.forEach(myFunction);

        function myFunction(item, index) {
          document.write(index + ":" + item + "<br>");
        }
</script>
```

```
0:apple
1:orange
2:cherry
```

Example.37: forEach method

```
<script type="text/javascript">
        var sum = 0;
        var numbers = [65, 44, 12, 4];
        numbers.forEach(myFunction);

        function myFunction(item) {
          sum += item;
        }
        document.write(sum);
</script>
```

```
125
```

7.2.3. indexOf ()

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

Example. 38: indexOf() method

```
<script type="text/javascript">
        var fruits = ["Banana", "Orange", "Apple", "Mango"];
        var a = fruits.indexOf("Apple");
        document.write("Index of Apple phrase is" , a);
</script>
```

Index of welcome phrase is 13

7.2.4. join()

Javascript array **join()** method joins all the elements of an array into a string.

Example. 39: join() method

```
<script type="text/javascript">
        var arr = new Array("First","Second","Third");
        var str = arr.join();
        document.write("str : " + str );

        var str = arr.join(", ");
        document.write("<br />str : " + str );

        var str = arr.join(" + ");
        document.write("<br />str : " + str );
</script>
```

str : First,Second,Third
str : First, Second, Third
str : First + Second + Third

7.2.5. lastIndexOf ()

Javascript array **lastIndexOf()** method returns the last index at which a given element can be found in the array, or -1 if it is not present. The array is searched backwards, starting at **fromIndex**.

Example. 40: lastIndexOf() method

```
<script type="text/javascript">
        var arr = ["C","C++","Python","C++","Java"];
        var result= arr.lastIndexOf("C++");
        document.writeln("Last Index of C++ is: " + result);
</script>
```

Last occurrence of planet is at index of 36

7.2.7. map()

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

Example. 41: map() method

```
<script type="text/javascript">
        var numbers = [4, 9, 16, 25];
        var x = numbers.map(Math.sqrt)
        document.write(" the result of mapping on the array is:  " , x);
</script>
```

the result of mapping on the array is: 2,3,4,5

7.2.8. pop()

Javascript array **pop()** method removes the last element from an array and returns that element.

Example. 42: pop() method

```
<script type="text/javascript">
        var fruits = ["Banana", "Orange", "Apple", "Mango"];
        fruits.pop();
        document.write(fruits.join(", "));
</script>
```

Banana, Orange, Apple

7.2.9. push()

Javascript array **push()** method appends the given element(s) in the last of the array and returns the length of the new array.

Example. 43: push() method

```
<script type="text/javascript">
        var fruits = ["Banana", "Orange", "Apple", "Mango"];
        fruits.push("Kiwi");
        document.write(fruits.join(", "));
</script>
```

Banana, Orange, Apple, Mango, Kiwi

7.2.10. reduce()

Javascript array **reduce()** method applies a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value.

Example.44: reduce method()

```
<script type="text/javascript">
        var numbers = [175, 50, 25];

        document.write("Reduce applied to array and the result is:
        ",numbers.reduce(myFunc));

        function myFunc(total, num) {
          return total - num;
        }
</script>
```

Reduce applied to array and the result is: 100

7.2.11. reduceRight()

Javascript array **reduceRight()** method applies a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value.

Example. 45: reduceRight() method

```
<script type="text/javascript">
        var numbers = [175, 50, 25];

        document.write("Reduce applied to array and the result is:
        ",numbers.reduceRight(myFunc));

        function myFunc(total, num) {
          return total - num;
        }
</script>
```

Reduce applied to array and the result is: -200

7.2.12. reverse()

Javascript array **reverse()** method reverses the element of an array. The first array element becomes the last and the last becomes the first.

Example. 46: reverse() method

```
<script type="text/javascript">
        var arr = [0, 1, 2, 3].reverse();
        document.write("Reversed array is : " + arr );
</script>
```

Reversed array is : 3,2,1,0

7.2.13. shift()

Javascript array **shift()** method removes the first element from an array and returns that element.

Example. 47: shift() method

```
<script type="text/javascript">
        var arr = [105, 1, 2, 3];
        var element = arr.shift();
        document.write("Removed element is: " + element );
        document.write("<br> Result array will be: " + arr.join(","));
</script>
```

Removed element is: 105
Result array will be: 1,2,3

7.2.14. Slice()

Javascript array **slice()** method extracts a section of an array and returns a new array.

Syntax

```
array.slice( begin [,end] );
```

Parameter Details

- begin : Zero-based index at which to begin extraction. As a negative index, start indicates an offset from the end of the sequence.
- end : Zero-based index at which to end extraction.

Example. 48: slice() method

```
<script type="text/javascript">
        var arr = ["orange", "mango", "banana", "sugar", "tea"];
        document.write("arr.slice( 1, 2) : " + arr.slice( 1, 2) );
        document.write("<br />arr.slice( 1, 3) : " + arr.slice( 1, 3) );
</script>
```

```
arr.slice( 1, 2) : mango
arr.slice( 1, 3) : mango,banana
```

7.2.15. some()

Javascript array **some()** method tests whether some element in the array passes the test implemented by the provided function.

Example. 49: some() method

```
<script type="text/javascript">
        var ages = [3, 10, 18, 20];

        function checkAdult(param) {
                return param >= 18;
        }

        document.write(ages.some(checkAdult));

</script>
```

```
true
```

7.2.16. sort()

Javascript array **sort()** method sorts the elements of an array.

Example. 50: sort() method

```
<script type="text/javascript">
        var arr = new Array("orange", "mango", "banana", "sugar");
        var sorted = arr.sort();
        document.write("Returned string is : " + sorted );
</script>
```

Returned string is : banana,mango,orange,sugar

8. Functions

A function is a group of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmers in writing modular codes. Functions allow a programmer to divide a big program into a number of small and manageable functions.

Like any other advanced programming language, JavaScript also supports all the features necessary to write modular code using functions. You must have seen functions like **alert()** and **write()** in the earlier chapters. We were using these functions again and again, but they had been written in core JavaScript only once.

JavaScript allows us to write our own functions as well. This section explains how to write your own functions in JavaScript.

8.1. Function Definition

Before we use a function, we need to define it. The most common way to define a function in JavaScript is by using the **function** keyword, followed by a unique function name, a list of parameters (that might be empty), and a statement block surrounded by curly braces.

Syntax of Function

```
<script type="text/javascript">
        function functionname(parameter-list) {
                statements
        }
</script>
```

Example. 51: Function

```
<script type="text/javascript">
        function sayHello() {
                alert("Hello there");
        }
</script>
```

8.2. Calling a Function

To invoke a function somewhere later in the script, you would simply need to write the name of that function as shown in the following code.

Example. 52: Calling Function

```
<html>
        <head>
                <script type="text/javascript">
                        function sayHello() {
                                document.write ("Hello there!");
                        }
                </script>
        </head>
        <body>
                <p>Click the following button to call the function</p>
                <form>
                        <input type="button" onclick="sayHello()" value="Say Hello">
                </form>
        </body>
</html>
```

Click the following button to call the function

Say Hello

Hello there!

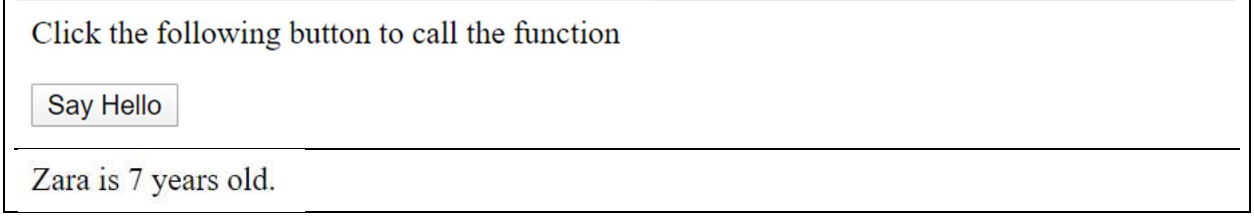
8.3. Function Parameters

Till now, we have seen functions without parameters. But there is a facility to pass different parameters while calling a function. These passed parameters can be captured inside the

function and any manipulation can be done over those parameters. A function can take multiple parameters separated by comma.

Example. 53: Function Parameters

```
<html>
	<head>
		<script type="text/javascript">
			function sayHello(name, age) {
				document.write (name + " is " + age + " years old.");
			}
		</script>
	</head>
	<body>
		<p>Click the following button to call the function</p>
		<form>
			<input type="button" onclick="sayHello('Zara', 7)" value="Say
			Hello">
		</form>
	</body>
</html>
```



8.4. The return Statement

A JavaScript function can have an optional **return** statement. This is required if you want to return a value from a function. This statement should be the last statement in a function.

For example, you can pass two numbers in a function and then you can expect the function to return their multiplication in your calling program.

Example. 54: Function Return Statement

```
<html>
    <head>
        <script type="text/javascript">
            function concatenate(first, last) {
                var full;
                full = first + last;
                return full;
            }
            function secondFunction() {
                var result;
                result = concatenate('Zara', 'Ali');
                document.write (result );
            }
        </script>
    </head>
    <body>
        <p>Click the following button to call the function</p>
        <form>
            <input type="button" onclick="secondFunction()" value="Call
            Function">
        </form>
    </body>
</html>
```

Click the following button to call the function

Call Function

ZaraAli

8.5. Nested Functions

Prior to JavaScript 1.2, function definition was allowed only in top-level global code, but JavaScript 1.2 allows function definitions to be nested within other functions as well. Still there is a restriction that function definitions may not appear within loops or conditionals. These restrictions on function definitions apply only to function declarations with the function statement.

As we will discuss later in the next chapter, function literals (another feature introduced in JavaScript 1.2) may appear within any JavaScript expression, which means that they can appear within **if** and other statements.

Example. 55: Nested Functions

```
<html>
	<head>
		<script type="text/javascript">
			function hypotenuse(a, b) {
				function square(x) { return x*x; }
				return Math.sqrt(square(a) + square(b));
				}
			function secondFunction(){
				var result;
				result = hypotenuse(1,2);
				document.write ( result );
				}
		</script>
	</head>
	<body>
		<p>Click the following button to call the function</p>
		<form>
			<input type="button" onclick="secondFunction()" value="Call
			Function">
		</form>
	</body>
</html>
```

Click the following button to call the function

Call Function

2.23606797749979

8.6. Function () Constructor

function statement is not the only way to define a new function; you can define your function dynamically using **Function()** constructor along with the **new** operator.

Hint: Constructor is a terminology from Object Oriented Programming. You may not feel comfortable for the first time, which is OK.

Syntax of Constructor

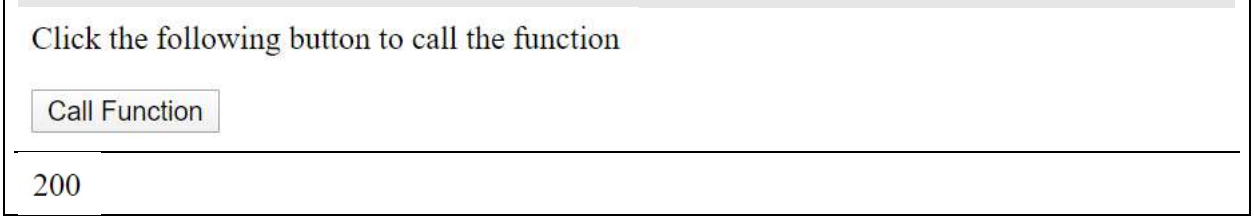
```
<script type="text/javascript">
        var variablename = new Function(Arg1, Arg2..., "Function Body");
</script>
```

The **Function**() constructor expects any number of string arguments. The last argument is the body of the function – it can contain arbitrary JavaScript statements, separated from each other by semicolons.

Notice that the **Function**() constructor is not passed any argument that specifies a name for the function it creates. The unnamed functions created with the **Function**() constructor are called anonymous functions.

Example. 56: Function Constructor

```
<html>
        <head>
                <script type="text/javascript">
                        var func = new Function("x", "y", "return x*y;");
                        function secondFunction(){
                                var result;
                                result = func(10,20);
                                document.write ( result );
                        }
                </script>
        </head>
        <body>
                <p>Click the following button to call the function</p>
                <form>
                        <input type="button" onclick="secondFunction()" value="Call
                        Function">
                </form>
        </body>
</html>
```



8.7. Function Literals

JavaScript 1.2 introduces the concept of function literals, which is another new way of defining functions. A function literal is an expression that defines an unnamed function.

Syntax No.1 of Function Literal

```
<script type="text/javascript">
        var variablename = function(Argument List){
                Function Body
        };
</script>
```

Syntactically, you can specify a function name while creating a literal function as follows.

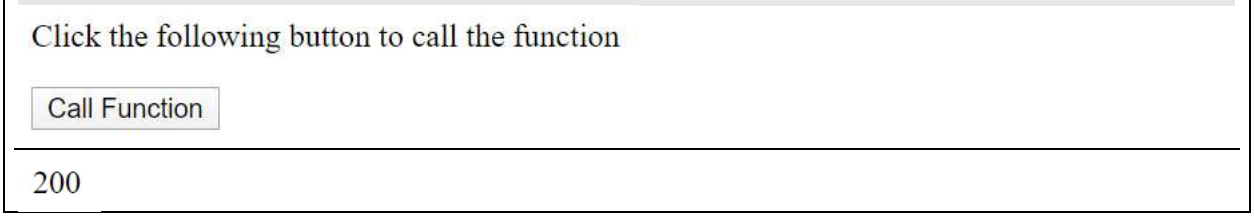
Syntax No.2 of Function Literal

```
<script type="text/javascript">
        var variablename = function FunctionName(Argument List){
                Function Body
        };
</script>
```

However, this name does not have any significance, so it is not worthwhile.

Example. 57: Function Literal

```
<html>
	<head>
		<script type="text/javascript">
			var func = function(x,y){ return x*y };
			function secondFunction(){
				var result;
				result = func(10,20);
				document.write ( result );
			}
		</script>
	</head>
	<body>
		<p>Click the following button to call the function</p>
		<form>
			<input type="button" onclick="secondFunction()" value="Call
			Function">
		</form>
	</body>
</html>
```



200

9. Events

JavaScript's interaction with HTML is handled through events that occur when the user or the browser manipulates a page.

When the page loads, it is called an event. When the user clicks a button, that click too is an event. Other examples include events like pressing any key, closing a window, resizing a window, etc.

Developers can use these events to execute JavaScript coded responses, which cause buttons to close windows, messages to be displayed to users, data to be validated, and virtually any other type of response imaginable.

Events are a part of the Document Object Model (DOM) Level 3 and every HTML element contains a set of events which can trigger JavaScript Code.

9.1. onclick Event Type

This is the most frequently used event type, which occurs when a user clicks the left button of his mouse. You can put your validation, warning etc., against this event type.

Example. 58: Onclick Event

```
<html>
	<head>
		<script type="text/javascript">
			<!--
			function sayHello() {
				document.write ("Hello World")
			}
			//-->
		</script>
	</head>
	<body>
		<p> Click the following button and see result</p>
		<input type="button" onClick="sayHello()" value="Say Hello" />
	</body>
</html>
```

Example. 58 Result: Onclick Event

Click the following button and see result

Say Hello

Hello World

9.2. onmouseover and onmouseout

These two event types will help you create nice effects with images or even with text as well. The onmouseover event triggers when you bring your mouse over any element and the onmouseout triggers when you move your mouse out from that element.

Example. 60: OnmouseOver and OnmouseOut Events

```
<html>
	<head>
		<script type="text/javascript">
			<!--
			function over() {
				console.log ("Mouse Over");
			}
			function out() {
				console.log ("Mouse Out");
			}
			//-->
		</script>
	</head>
	<body>
		<p>Bring your mouse inside the division to see the result:</p>
		<div onmouseover="over()" onmouseout="out()">
			<h2> This is inside the division </h2>
		</div>
	</body>
</html>
```

Example. 61: Event handler for random background generator

```
<script>
        const btn = document.querySelector('button');

        function random(number) {
          return Math.floor(Math.random() * (number+1));
        }

        btn.onclick = function() {
          const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255)
        + ')';
          document.body.style.backgroundColor = rndCol;
        }
</script>
```

Example. 62: Event handler

```
<script>
        const btn = document.querySelector('button');

        function random(number) {
          return Math.floor(Math.random() * (number+1));
        }

        function bgChange() {
          const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255)
        + ')';
          document.body.style.backgroundColor = rndCol;
        }

        btn.onclick = bgChange;
</sccript>
```

Example. 63: Event Handler

```
<script>
        const buttons = document.querySelectorAll('button');

        function bgChange() {
          const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255)
        + ')';
          document.body.style.backgroundColor = rndCol;
        }

        for (let i = 0; i < buttons.length; i++) {
          buttons[i].onclick = bgChange;
        }
</script>
```

9.3. addEventListener()

The newest type of event mechanism is defined in the Document Object Model (DOM) Level 2 Events Specification, which provides browsers with a new function — addEventListener(). This functions in a similar way to the event handler properties, but the syntax is obviously different. We could rewrite our random color example to look like this:

Example. 64: addEventListener

```
<script>
	const btn = document.querySelector('button');

	function random(number) {
	  return Math.floor(Math.random() * (number+1));
	}

	function bgChange() {
	  const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255)
	+ ')';
	  document.body.style.backgroundColor = rndCol;
	}

	btn.addEventListener('click', bgChange);
</script>
```

Or:

Example. 65: addEventListener

```
<script>
	btn.addEventListener('click', function() {
	  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) +
	')';
	  document.body.style.backgroundColor = rndCol;
	});
</script>
```

9.4. removeEventListener()

Removes an event handler that has been attached with the addEventListener() method.

Note: To remove event handlers, the function specified with the addEventListener() method must be an external function, like in the example above (myFunction).

Example. 66: addEventListener

```
<script>
	btn.removeEventListener('click', bgChange);
</script>
```

9.5. preventDefualt method

cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur.

Example. 67: preventDefault Method

```
<!DOCTYPE html>
<html>
	<body>

		<a id="myAnchor" href="https://w3schools.com/">Go to
		W3Schools.com</a>

		<p>The preventDefault() method will prevent the link above from
		following the URL.</p>

		<script>
			document.getElementById("myAnchor").addEventListener("cl
			ick", function(event){
			  event.preventDefault()
			});
		</script>

	</body>
</html>
```

9.6. Drag and Drop

Example.68: Simple drag and drop example

```
<!DOCTYPE HTML>
<html>
	<head>
		<script>
			function allowDrop(ev) {
			 ev.preventDefault();
			}

			function drag(ev) {
			 ev.dataTransfer.setData("text", ev.target.id);
			}

			function drop(ev) {
			 ev.preventDefault();
			 var data = ev.dataTransfer.getData("text");
			 ev.target.appendChild(document.getElementById(data));
			}
		</script>
	</head>
	<body>

		<div id="div1" ondrop="drop(event)" ondragover="allowDrop(event)"></div>

		<img id="drag1" src="img_logo.gif" draggable="true"
		ondragstart="drag(event)" width="336" height="69">

	</body>
</html>
```

9.7. HTML5 Standard Events

The standard HTML 5 events are listed here for your reference.
Here script indicates a Javascript function to be executed against that event.

Attribute	*Value*	*Description*
Offline	script	Triggers when the document goes offline
Onabort	script	Triggers on an abort event
onafterprint	script	Triggers after the document is printed

onbeforeonload	script	Triggers before the document loads
onbeforeprint	script	Triggers before the document is printed
onblur	script	Triggers when the window loses focus
oncanplay	script	Triggers when media can start play, but might has to stop for buffering
oncanplaythrough	script	Triggers when media can be played to the end, without stopping for buffering
onchange	script	Triggers when an element changes
onclick	script	Triggers on a mouse click
oncontextmenu	script	Triggers when a context menu is triggered
ondblclick	script	Triggers on a mouse double-click
ondrag	script	Triggers when an element is dragged
ondragend	script	Triggers at the end of a drag operation
ondragenter	script	Triggers when an element has been dragged to a valid drop target
ondragleave	script	Triggers when an element leaves a valid drop target
ondragover	script	Triggers when an element is being dragged over a valid drop target
ondragstart	script	Triggers at the start of a drag operation
ondrop	script	Triggers when dragged element is being Dropped
ondurationchange	script	Triggers when the length of the media is changed
onemptied	script	Triggers when a media resource element suddenly becomes empty.
onended	script	Triggers when media has reach the end

onerror	script	Triggers when an error occur
onfocus	script	Triggers when the window gets focus
onformchange	script	Triggers when a form changes
onforminput	script	Triggers when a form gets user input
onhaschange	script	Triggers when the document has change
oninput	script	Triggers when an element gets user input
oninvalid	script	Triggers when an element is invalid
onkeydown	script	Triggers when a key is pressed
onkeypress	script	Triggers when a key is pressed and released
onkeyup	script	Triggers when a key is released
onload	script	Triggers when the document loads
onloadeddata	script	Triggers when media data is loaded
onloadedmetadata	script	Triggers when the duration and other media data of a media element is loaded
onloadstart	script	Triggers when the browser starts to load the media data
onmessage	script	Triggers when the message is triggered
onmousedown	script	Triggers when a mouse button is pressed
onmousemove	script	Triggers when the mouse pointer moves
onmouseout	script	Triggers when the mouse pointer moves out of an element
onmouseover	script	Triggers when the mouse pointer moves over an element
onmouseup	script	Triggers when a mouse button is released
onmousewheel	script	Triggers when the mouse wheel is being rotated

onoffline	script	Triggers when the document goes offline
ononline	script	Triggers when the document comes online
onpagehide	script	Triggers when the window is hidden
onpageshow	script	Triggers when the window becomes visible
onpause	script	Triggers when media data is paused
onplay	script	Triggers when media data is going to start playing
onplaying	script	Triggers when media data has start playing
onpopstate	script	Triggers when the window's history changes
onprogress	script	Triggers when the browser is fetching the media data
onratechange	script	Triggers when the media data's playing rate has changed
onreadystatechange	script	Triggers when the ready-state changes
onredo	script	Triggers when the document performs a redo
onresize	script	Triggers when the window is resized
onscroll	script	Triggers when an element's scrollbar is being scrolled
onseeked	script	Triggers when a media element's seeking attribute is no longer true, and the seeking has ended
onseeking	script	Triggers when a media element's seeking attribute is true, and the seeking has begun
onselect	script	Triggers when an element is selected
onstalled	script	Triggers when there is an error in fetching media data
onstorage	script	Triggers when a document loads

onsubmit	script	Triggers when a form is submitted
onsuspend	script	Triggers when the browser has been fetchingmedia data, but stopped before the entire media file was fetched
ontimeupdate	script	Triggers when media changes its playing position
onundo	script	Triggers when a document performs an undo
onunload	script	Triggers when the user leaves the document
onvolumechange	script	Triggers when media changes the volume, also when volume is set to "mute"
onwaiting	script	Triggers when media has stopped playing, but is expected to resume

10. DOM

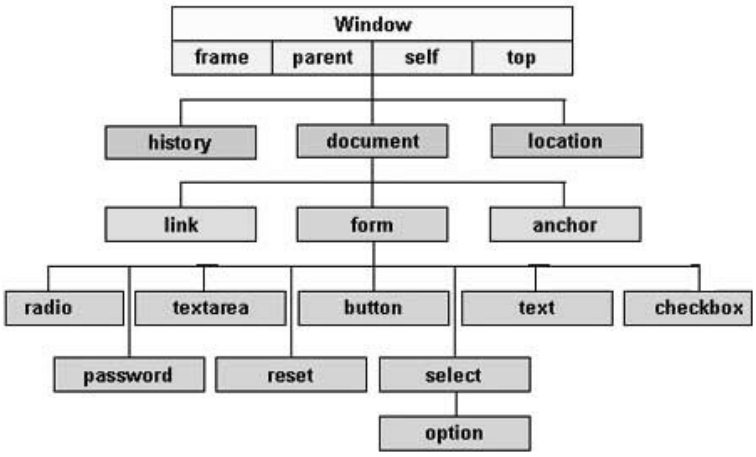
Every web page resides inside a browser window, which can be considered as an object.

A Document object represents the HTML document that is displayed in that window. The Document object has various properties that refer to other objects, which allow access to and modification of document content.

The way a document content is accessed and modified is called the **Document Object Model**, or **DOM**. The Objects are organized in a hierarchy. This hierarchical structure applies to the organization of objects in a Web document.

- **Window object:** Top of the hierarchy. It is the outmost element of the object hierarchy.
- **Document object:** Each HTML document that gets loaded into a window becomes a document object. The document contains the contents of the page.
- **Form object:** Everything enclosed in the <form>...</form> tags sets the form object.
- **Form control elements:** The form object contains all the elements defined for that object such as text fields, buttons, radio buttons, and checkboxes.

Here is a simple hierarchy of a few important objects:



There are several DOMs in existence. The following sections explain each of these DOMs in detail and describe how you can use them to access and modify document content.

- **The Legacy DOM**: This is the model which was introduced in early versions of JavaScript language. It is well supported by all browsers, but allows access only to certain key portions of documents, such as forms, form elements, and images.
- **The W3C DOM**: This document object model allows access and modification of all document content and is standardized by the World Wide Web Consortium (W3C). This model is supported by almost all the modern browsers.
- **The IE4 DOM**: This document object model was introduced in Version 4 of Microsoft's Internet Explorer browser. IE 5 and later versions include support for most basic W3C DOM features.

10.1. The Legacy DOM

This is the model, which was introduced in early versions of JavaScript language. It is well supported by all browsers, but allows access only to certain key portions of documents, such as forms, form elements, and images.

This model provides several read-only properties, such as title, URL, and lastModified provide information about the document as a whole. Apart from that, there are various methods provided by this model, which can be used to set and get document property values.

10.1.1. Document Properties in Legacy DOM

Here is a list of the document properties, which can be accessed using Legacy DOM.

No.	Property and Description
1	**alinkColor** Deprecated - A string that specifies the color of activated links. **Ex:** document.alinkColor

2	**anchors[ ]** An array of Anchor objects, one for each anchor that appears in the document **Ex:** document.anchors[0], document.anchors[1] and so on
3	**applets[ ]** An array of Applet objects, one for each applet that appears in the document **Ex:** document.applets[0], document.applets[1] and so on
4	**bgColor** Deprecated - A string that specifies the background color of the document. **Ex:** document.bgColor
5	**Cookie** A string valued property with special behavior that allows the cookies associated with this document to be queried and set. **Ex:** document.cookie
6	**Domain** A string that specifies the Internet domain the document is from. Used for security purpose. **Ex:** document.domain
7	**embeds[ ]** An array of objects that represent data embedded in the document with the <embed> tag. A synonym for plugins []. Some plugins and ActiveX controls can be controlled with JavaScript code. **Ex:** document.embeds[0], document.embeds[1] and so on
8	**fgColor** A string that specifies the default text color for the document **Ex:** document.fgColor
9	**forms[ ]** An array of Form objects, one for each HTML form that appears in the document. **Ex:** document.forms[0], document.forms[1] and so on
10	**images[ ]** An array of Image objects, one for each image that is embedded in the document with the HTML <img> tag.

	Ex: document.images[0], document.images[1] and so on
11	**lastModified** A read-only string that specifies the date of the most recent change to the document **Ex:** document.lastModified
12	**linkColor** Deprecated - A string that specifies the color of unvisited links **Ex:** document.linkColor
13	**links[ ]** It is a document link array. **Ex:** document.links[0], document.links[1] and so on
14	**Location** The URL of the document. Deprecated in favor of the URL property. **Ex:** document.location
15	**plugins[ ]** A synonym for the embeds[ ] **Ex:** document.plugins[0], document.plugins[1] and so on
16	**Referrer** A read-only string that contains the URL of the document, if any, from which the current document was linked. **Ex:** document.referrer
17	**Title** The text contents of the <title> tag. **Ex:** document.title
18	**URL** A read-only string that specifies the URL of the document. **Ex:** document.URL
19	**vlinkColor** Deprecated - A string that specifies the color of visited links. **Ex:** document.vlinkColor

10.1.2. Document Methods in Legacy DOM

Here is a list of methods supported by Legacy DOM.

No.	Property and Description
1	**clear( )** Deprecated - Erases the contents of the document and returns nothing. **Ex:** document.clear( )
2	**close( )** Closes a document stream opened with the open( ) method and returns nothing. **Ex:** document.close( )
3	**open( )** Deletes existing document content and opens a stream to which new document contents may be written. Returns nothing. **Ex:** document.open( )
4	**write(value, ...)** Inserts the specified string or strings into the document currently being parsed or appends to document opened with open( ). Returns nothing. **Ex:** document.write(value, ...)
5	**writeln(value, ...)** Identical to write( ), except that it appends a newline character to the output. Returns nothing. **Ex:** document.writeln(value, ...)

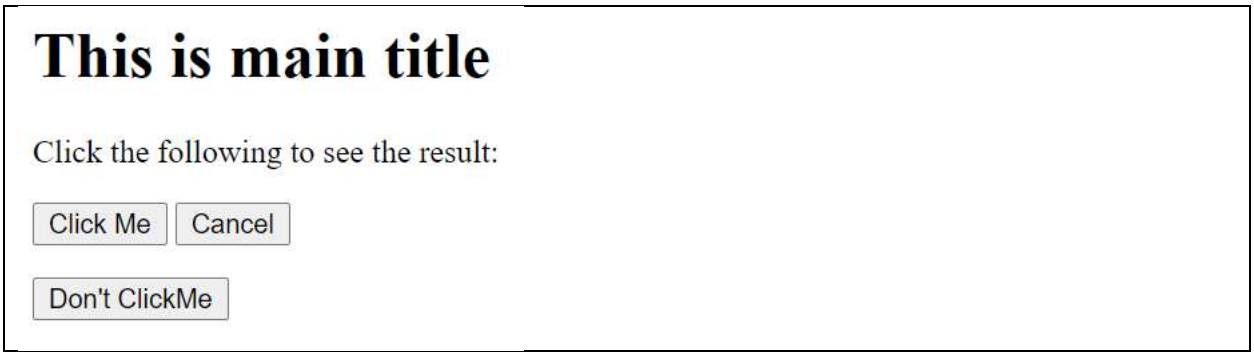
Example. 69: access document properties using Legacy DOM method

We can locate any HTML element within any HTML document using HTML DOM. For instance, if a web document contains a **form** element, then using JavaScript, we can refer to it as **document.forms[0].** If your Web document includes two **form** elements, the first form is referred to as document.forms[0] and the second as document.forms[1].

Using the hierarchy and properties given above, we can access the first form element using **document.forms[0].elements[0]** and so on.

Here is an example to access document properties using Legacy DOM method.

```
<html>
	<head>
		<title> Document Title </title>
		<script type="text/javascript">
		<!--
			function myFunc()
			{
				var ret = document.title;
				alert("Document Title : " + ret );
				var ret = document.URL;
				alert("Document URL : " + ret );
				var ret = document.forms[0];
				alert("Document First Form : " + ret );
				var ret = document.forms[0].elements[1];
				alert("Second element : " + ret );
			}
		//-->
		</script>
	</head>
<body>
		<h1 id="title">This is main title</h1>
		<p>Click the following to see the result:</p>
		<form name="FirstForm">
			<input type="button" value="Click Me" onclick="myFunc();"
			/>
			<input type="button" value="Cancel">
		</form>
		<form name="SecondForm">
			<input type="button" value="Don't ClickMe"/>
		</form>
	</body>
</html>
```



10.2. The W3C DOM

This document object model allows access and modification of all document content and is standardized by the World Wide Web Consortium (W3C). This model is supported by almost all the modern browsers.

The W3C DOM standardizes most of the features of the legacy DOM and adds new ones as well. In addition to supporting forms[ ], images[ ], and other array properties of the Document object, it defines methods that allow scripts to access and manipulate any document element and not just special-purpose elements like forms and images.

10.2.1. Document Properties in W3C DOM

This model supports all the properties available in Legacy DOM. Additionally, here is a list of document properties, which can be accessed using W3C DOM.

No.	Property and Description
1	**Body** A reference to the Element object that represents the <body> tag of this document. **Ex:** document.body
2	**defaultView** It is a read-only property and represents the window in which the document is displayed. **Ex:** document.defaultView
3	**documentElement** A read-only reference to the <html> tag of the document. **Ex:** document.documentElement8/31/2008
4	**Implementation** It is a read-only property and represents the DOM Implementation object that represents the implementation that created this document. **Ex:** document.implementation

10.2.2. Document Methods in W3C DOM

This model supports all the methods available in Legacy DOM. Additionally, here is a list of methods supported by W3C DOM.

No.	Property and Description
1	**createAttribute(name)** Returns a newly-created Attr node with the specified name. **Ex:** document.createAttribute(name)
2	**createComment(text)** Creates and returns a new Comment node containing the specified text. **Ex:** document.createComment(text)
3	**createDocumentFragment( )** Creates and returns an empty DocumentFragment node. **Ex:** document.createDocumentFragment( )
4	**createElement(tagName)** Creates and returns a new Element node with the specified tag name. **Ex:** document.createElement(tagName)
5	**createTextNode(text)** Creates and returns a new Text node that contains the specified text. **Ex:** document.createTextNode(text)
6	**getElementById(id)** Returns the Element of this document that has the specified value for its id attribute, or null if no such Element exists in the document. **Ex:** document.getElementById(id)
7	**getElementsByName(name)** Returns an array of nodes of all elements in the document that have a specified value for their name attribute. If no such elements are found, returns a zero-length array. **Ex:** document.getElementsByName(name)
8	**getElementsByTagName(tagname)** Returns an array of all Element nodes in this document that have the specified tag name. The Element nodes appear in the returned array in the same order they appear in the document source.

	Ex: document.getElementsByTagName(tagname)
9	**importNode(importedNode, deep)** Creates and returns a copy of a node from some other document that is suitable for insertion into this document. If the deep argument is true, it recursively copies the children of the node too. Supported in DOM Version 2 **Ex:** document.importNode(importedNode, deep)
10	**getElementsByClassName (classname)** Get all elements with the specified class name. **Ex:** var x = document.getElementsByClassName("example");
11	**querySelector()** returns the first Element within the document that matches the specified selector, or group of selectors. If no matches are found, null is returned. **Ex**: element = document.querySelector(selectors);
12	**querySelectorAll()** returns all elements in the document that matches a specified CSS selector(s), as a static NodeList object **Ex**: elementList = document.querySelectorAll(selectors);
13	**setAttribute()** sets a specific HTML Attribute **Ex:** videoBox.setAttribute('class', 'hidden');

Example. 70: access document properties using W3C DOM method

```
<html>
	<head>
		<title> Document Title </title>
		<script type="text/javascript">
		<!--
			function myFunc()
			{
				var ret = document.getElementsByTagName("title");
				alert("Document Title : " + ret[0].text );
				var ret = document.getElementById("heading");
				alert("Document URL : " + ret.innerHTML );
			}
		//-->
		</script>
	</head>
	<body>
		<h1 id="heading">This is main title</h1>
		<p>Click the following to see the result:</p>
		<form id="form1" name="FirstForm">
			<input type="button" value="Click Me" onclick="myFunc();"
			/>
			<input type="button" value="Cancel">
		</form>
		<form d="form2" name="SecondForm">
			<input type="button" value="Don't ClickMe"/>
		</form>
	</body>
</html>
```

Example. 71: createAttribute Example

```
<div id="div1">
</div>
<button onClick="clickMe()">Click me</button>
<script>
	function clickMe(){
		var node = document.getElementById("div1");
		var a = document.createAttribute("my_attrib");
		a.value = "newVal";
		node.setAttributeNode(a);
		console.log(node.getAttribute("my_attrib")); // "newVal"
	}
</script>
```

Example. 72: createElement Example

```
<html>
	<head>
		<style type="text/css">
			.myBTN {
			  background-color: #AC93FF;
			  color: #050E43;
			  width: 300px;
			  height: 100px;
			}
		</style>
	<head>
	<body>

	<p>Click the button to make a BUTTON element.</p>

	<button onclick="myFunction()">Try it</button>

	<script>
		function myFunction() {
		 var btn = document.createElement("BUTTON");
			btn.innerHTML = "Click me";
			btn.setAttribute("class", "myBTN");
		 document.body.appendChild(btn);
		}
	</script>

	</body>
</html>
```

10.3. The IE 4 DOM

This document object model was introduced in Version 4 of Microsoft's Internet Explorer browser. IE 5 and later versions include support for most basic W3C DOM features.

9.3.1. Document Properties in IE 4 DOM

The following non-standard (and non-portable) properties are defined by Internet Explorer 4 and later versions.

No.	Property and Description
1	**activeElement** A read-only property that refers to the input element that is currently active (i.e., has the input focus). **Ex:** document.activeElement
2	**all[ ]** An array of all Element objects within the document. This array may be indexed numerically to access elements in source order, or it may be indexed by element id or name. **Ex:** document.all[ ]
3	**Charset** The character set of the document. **Ex:** document.charset
4	**children[ ]** An array that contains the HTML elements that are the direct children of the document. Note that this is different from the all [ ] array that contains all the elements in the document, regardless of their position in the containment hierarchy. **Ex:** document.children[ ]
5	**defaultCharset** The default character set of the document. **Ex:** document.defaultCharset
6	**Expand** This property, if set to false, prevents client-side objects from being expanded. **Ex:** document.expando
7	**parentWindow** The window that contains the document. **Ex:** document.parentWindow
8	**readyState** Specifies the loading status of a document. It has one of the following four string values: **Uninitialized** The document has not started loading. **Example**: document.uninitialized

	Loading The document is loading. **Ex:** document.loading **Interactive** The document has loaded sufficiently for the user to interact with it. **Ex:** document.interactive **Complete** The document is completely loaded. **Ex:** document.complete **Ex:** document.readyState

10.3.2. Document Methods in IE4 DOM

This model supports all the methods available in Legacy DOM. Additionally, here is a list of methods supported by IE4 DOM.

No.	Property and Description
1	**elementFromPoint(x,y)** Returns the Element located at a specified point. **Ex:** document.elementFromPoint(x,y)

Example

The IE 4 DOM does not support the **getElementById()** method. Instead, it allows you to look up arbitrary document elements by **id** attribute within the **all []** array of the document object.

Here is how to find all <li> tags within the first <ul> tag. Note that you must specify the desired HTML tag name in uppercase with the **all.tags()** method.

Example. 73: access document properties using IE4 DOM

```
<html>
	<head>
		<title> Document Title </title>
		<script type="text/javascript">
			<!--
			function myFunc()
			{
				var ret = document.all["heading"];
				alert("Document Heading : " + ret.innerHTML );
				var ret = document.all.tags("P");;
				alert("First Paragraph : " + ret[0].innerHTML);
			}
			//-->
		</script>
	</head>
	<body>
		<h1 id="heading">This is main title</h1>
		<p>Click the following to see the result:</p>
		<form id="form1" name="FirstForm">
			<input type="button" value="Click Me" onclick="myFunc();"
			/>
			<input type="button" value="Cancel">
		</form>
		<form d="form2" name="SecondForm">
			<input type="button" value="Don't ClickMe"/>
		</form>
	</body>
</html>
```

10.4. DOM Compatibility

If you want to write a script with the flexibility to use either W3C DOM or IE 4 DOM depending on their availability, then you can use a capability-testing approach that first checks for the existence of a method or property to determine whether the browser has the capability you desire. For example:

```
if (document.getElementById) {
// If the W3C method exists, use it
}
else if (document.all) {
// If the all[] array exists, use it
}
else {
// Otherwise use the legacy DOM
}
```

11. How to traverse DOM

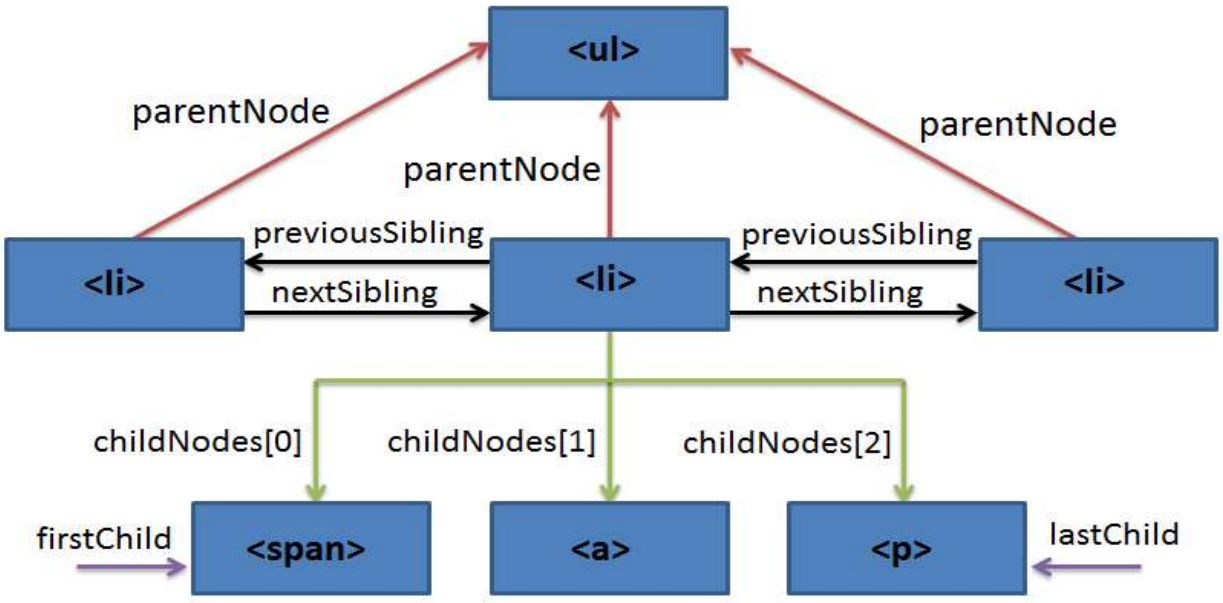
Elements in the DOM are organized into a tree-like data structure that can be traversed to navigate, locate, or modify elements and/or content within an XML/HTML document.

The DOM tree can be imagined as a collection of "**nodes**" related to each other through **parent-child** and **sibling-sibling** relationships. Each node represents an object in an XML document including elements, textual content, and comments. Each XML document contains a single root element (<html> in HTML, for example) from which all other nodes ultimately descend.

DOM tree traversal may be accomplished through the use of six basic properties. All properties, **except childNodes**, contain a reference to another node object. The **childNodes** property contains a reference to an array of nodes.

previousSibling	nextSibling	childNodes
firstChild	lastChild	parentNode

The following schematic succinctly illustrates the relationship between nodes:



Hint: The above schematic does not take into account text nodes, which would be considered child nodes of their containing element node.

Sample HTML Code

```
<html>
    <head>
        <title>This is a Document!</title>
    </head>
    <body>
        <h1>This is a header!</h1>
        <p id="excitingText">
            This is a paragraph! <em>Excitement</em>!
        </p>
        <p>
            This is also a paragraph, but it's not nearly as exciting as the
            last one.
        </p>
    </body>
</html>
```

As you can see, the entire document is contained within an html element. That element directly contains two others: head and body. Those show up in our model as its children, and they each point to html as their parent. And so it goes, down through the document hierarchy, with each element pointing to its direct descendants as children, and to its direct ancestor as parent:

title is a child of head.

body has three children — two p elements and an h1 element.

The p element with the id="excitingText has a child of its own — an em element.

The plain text content of the elements (ie “This is a Document!”) is also represented in the DOM, as **text nodes**. These have no children of their own, but do point to their containing elements as parents.

11.1. Nodes

Each node in the DOM tree is an object representing a single element on the page. Nodes understand their relationship to other nodes in their immediate vicinity, and contain a good deal of information about themselves. In much the same way as a child might clamber from one branch to the next closest in a backyard oak, I can gather all the information from a node that I need to get to its parent or to its children.

As you might expect, given JavaScript's object-nature, the information I'm looking for in this case is exposed via the node's properties. Specifically, the parentNode and childNodes properties. As each element on the page has at most one parent, the parentNode property is straightforward: it simply gives you access to the node's parent. Nodes can have any number of children, however, so the childNodes property is actually an array. Each element of the array points to one child, in the same order they appear in the document. Our example document's body element would therefore have a childNodes array containing the h1, the first p, then the second p, in that order.

11.2. Branch to branch

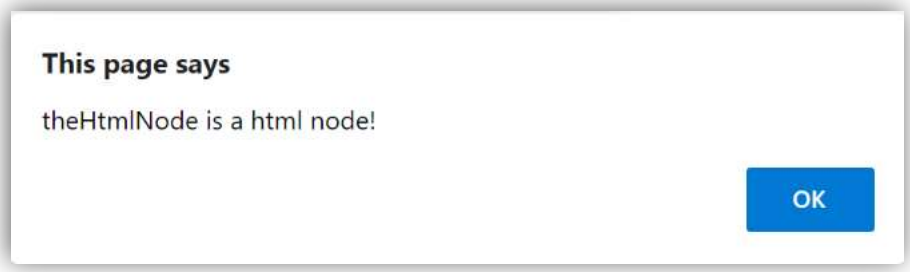
The best place to begin is at the document's root, accessible via an object creatively named document. As document is right at the root, it doesn't have a parentNode, but it does have a single child: the html element node, which we can access via document's childNodes array:

```
var theHtmlNode = document.childNodes[0];
```

This line of code creates a new variable named theHtmlNode, and assigns it the value of the document object's first child (remember that JavaScript arrays start numbering with 0, not 1). You can confirm that you've gotten your hands on the html node by examining theHtmlNode's nodeName property, which gives vital information about the exact kind of node you're dealing with:

Example. 74: childNodes example

```
<script type="text/javascript">
        var theHtmlNode = document.childNodes[0];
        alert( "theHtmlNode is a " + theHtmlNode.nodeName + " node!" );
</script>
```



11.3 Start to traverse

To begin, we will create a new file called nodes.html comprised of the following code:

Example. 75: traverse Dom

```
<!DOCTYPE html>
<html>
	<head>
		<title>Learning About Nodes</title>
	</head>
	<body>
		<h1>Shark World</h1>
		<p>The world's leading source on <strong>shark</strong> related
		information.</p>
		<h2>Types of Sharks</h2>
		<ul id="myUl">
			<li>Hammerhead</li>
			<li>Tiger</li>
			<li>Great White</li>
		</ul>
	</body>

	<script>
		const h1 = document.getElementsByTagName('h1')[0];
		const p = document.getElementsByTagName('p')[0];
		const ul = document.getElementsByTagName('ul')[0];
		document.write("<hr>h1 is: ", h1, "<br>p is: ", p, "<br> ul is: ", ul);
	</script>
</html>
```

Since there is only one of each `h1`, `p`, and `ul`, we can access the first index on each respective `getElementsByTagName` property.

11.4. Root Nodes

The `document` object is the root of every node in the DOM. This object is actually a property of the `window` object, which is the global, top-level object representing a tab in the browser. The `window` object has access to such information as the toolbar, height and width of the window, prompts, and alerts. The `document` consists of what is inside of the inner `window`.

Below is a chart consisting of the root elements that every document will contain. Even if a blank HTML file is loaded into a browser, these three nodes will be added and parsed into the DOM.

Property	Node	Node Type
document	#document	DOCUMENT_NODE
document.documentElement	html	ELEMENT_NODE
document.head	head	ELEMENT_NODE
document.body	body	ELEMENT_NODE

Since the html, head, and body elements are so common, they have their own properties on the document.

11.5. Parent Nodes

The nodes in the DOM are referred to as **parents**, **children**, and **siblings**, depending on their relation to other nodes. The **parent** of any node is the node that is **one level above it**, or closer to the document in the DOM hierarchy. There are two properties to get the parent — parentNode and parentElement.

Property	Gets
parentNode	Parent Node
parentElement	Parent Element Node

We can test what the parent of our p element is with the parentNode property. This p variable comes from our custom document.getElementsByTagName('p')[0] declaration.

Example. 76: parentNode example

```
<script>
        const p = document.getElementsByTagName('p')[0];
        document.write("<hr>Parent of P is: ", p.parentNode);
</script>
```

Parent of P is: [object HTMLBodyElement]

The parent of p is body, but how can we get the grandparent, which is two levels above? We can do so by chaining properties together.

Example. 77: Grandparent access

```
<script>
        const p = document.getElementsByTagName('p')[0];
        document.write("<hr>Parent of parent of P is: ", p.parentNode.
      parentNode);
</script>
```

Parent of parent of P is: [object HTMLHtmlElement]

Using parentNode twice, we retrieved the grandparent of p.

There are properties to retrieve the parent of a node, but only one small difference between them, as demonstrated in this snippet below.

Example. 78: parentElement Example

```
<script>
        const html = document.documentElement;

      console.log("Html parentNode", html.parentNode);
      console.log("HTML parentElement", html.parentElement);
</script>
```

```
call succeeded!!
success
Html parentNode ▶#document
HTML parentElement null
>
```

The parent of almost any node is an element node, as text and comments cannot be parents to other nodes. However, the parent of html is a document node, so parentElement returns null. Generally, parentNode is more commonly used when traversing the DOM.

11.6. Children Nodes

The **children** of a node are the nodes that are one level below it. Any nodes beyond one level of nesting are usually referred to as descendants.

Property	*Gets*
childNodes	Child Nodes
firstChild	First Child Node
lastChild	Last Child Node
children	Element Child Nodes
firstElementChild	First Child Element Node
lastElementChild	Last Child Element Node

The childNodes property will return a live list of every child of a node. You might expect the ul element to get three li elements. Let us test what it retrieves.

Example. 79: childNodes example

```
<script>
    const ul = document.getElementById("myUl");

    console.log("ul childNodes:", ul.childNodes);
</script>
```

```
call succeeded!!
success
ul childNodes: ▶ NodeList(7) [text, li, text, li, text, li, text]
>
```

In addition to the three li elements, it also gets four text nodes. This is because we wrote our own HTML (it was not generated by JavaScript) and the indentation between elements is counted in the DOM as text nodes.

If we attempted to change the background color of the first child node using the firstChild property, it would fail because the first node is text.

Example. 80: firstChild Example

```
<script>
    const ul = document.getElementById("myUl");

    ul.firstChild.style.background = 'yellow';
</script>
```

```
call succeeded!!
success
Uncaught TypeError: Cannot set property 'background' of undefined
    at 2XLfKOr:30
>
```

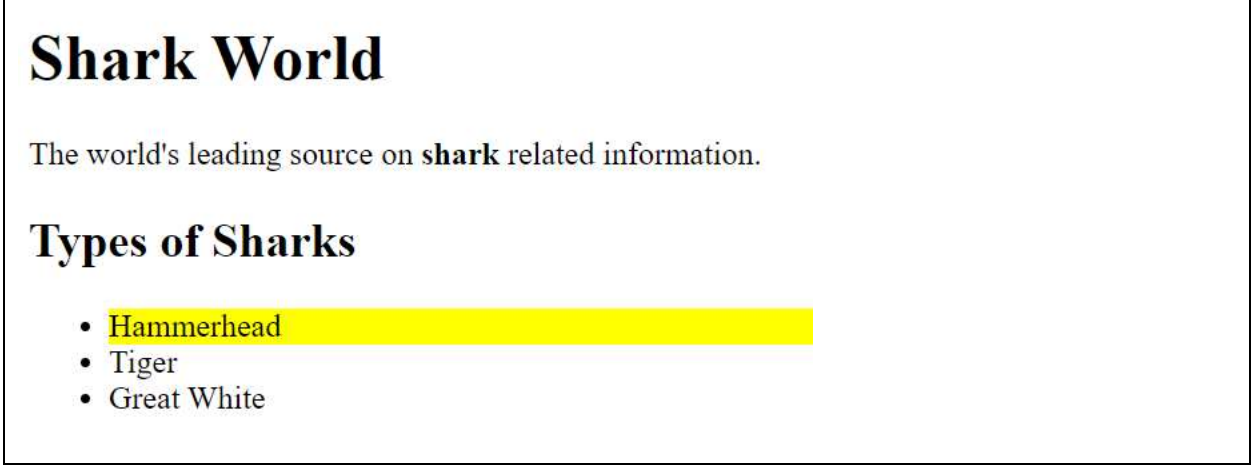
The children, firstElementChild and lastElementChild properties exist in these types of situations to retrieve only the element nodes. ul.children will only return the three li elements.

Using firstElementChild, we can change the background color of the first li in the ul.

Example. 81: firstElementChild Example

```
<script>
    const ul = document.getElementById("myUl");

    ul.firstElementChild.style.background = 'yellow';
</script>
```



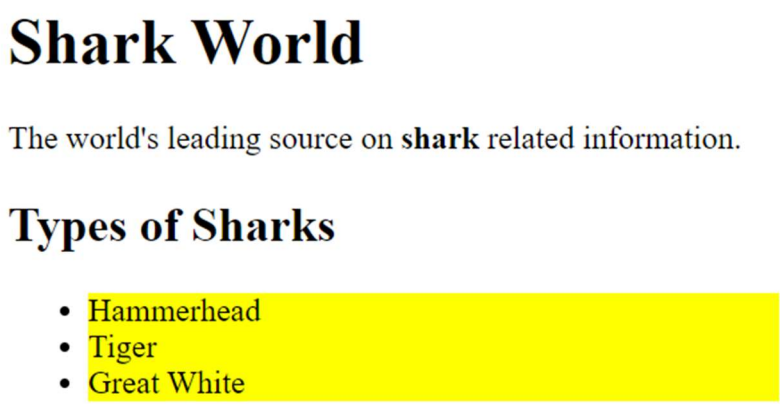
When doing basic DOM manipulation such as in this example, the element-specific properties are extremely helpful. In JavaScript-generated web apps, the properties that select all nodes are more likely to be used, as white-space newlines and indentation will not exist in this case.

A for...of loop can be used to iterate through all children elements.

Example. 82: traverse all children elements

```
<script>
        const ul = document.getElementById("myUl");

        for (let element of ul.children) {
                element.style.background = 'yellow';
              }
</script>
```



The below code will find the last element child (li) of the fourth child element (ul) of body and apply a style.

Example. 83: lastElementChild Example

```
<script>
        const ul = document.getElementById("myUl");

        for (let element of ul.children) {
                element.style.background = 'yellow';
              }
        document.body.children[3].lastElementChild.style.background = 'fuchsia';
</script>
```

11.7. Sibling Nodes

The **siblings** of a node are any node on the same tree level in the DOM. Siblings do not have to be the same type of node - text, element, and comment nodes can all be siblings.

Property	*Gets*
previousSibling	Previous Sibling Node
nextSibling	Next Sibling Node
previousElementSibling	Previous Sibling Element Node
nextElementSibling	Next Sibling Element Node

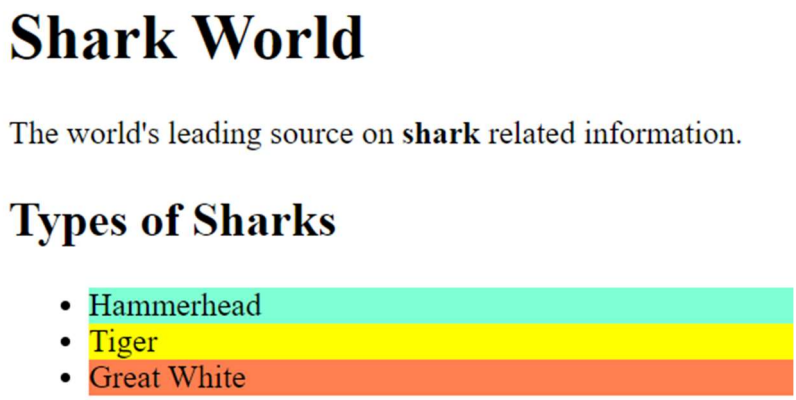
Sibling properties work the same way as the children nodes, in that there is a set of properties to traverse all nodes, and a set of properties for only element nodes. `previousSibling` and `nextSibling` will get the next node that immediately precedes or follows the specified node, and `previousElementSibling` and `nextElementSibling` will only get element nodes.

In our previous example, let's select the middle element of `ul`.

Example. 84: sibling traverese

```
<script>
        const ul = document.getElementById("myUl");

        for (let element of ul.children) {
        element.style.background = 'yellow';
        }
        const tiger = ul.children[1];
        tiger.nextElementSibling.style.background = 'coral';
        tiger.previousElementSibling.style.background = 'aquamarine';
</script>
```

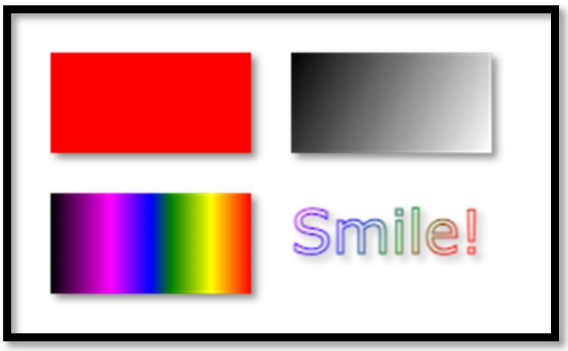


Sibling properties can be chained together, just like parent and node properties.

12. Javascript Convas

The HTML <canvas> element is used to draw graphics on a web page.

The graphic to the left is created with <canvas>. It shows four elements:



a red rectangle

a gradient rectangle

a multicolor rectangle

and a multicolor text

12.1. What is HTML Canvas?

The HTML <canvas> element is used to draw graphics, on the fly, via JavaScript. The <canvas> element is only a container for graphics. You must use JavaScript to actually draw the graphics. Canvas has several methods for drawing paths, boxes, circles, text, and adding images.

12.2. Canvas Examples

A canvas is a rectangular area on an HTML page. By default, a canvas has no border and no content.

The markup looks like this:

```
<canvas id="myCanvas" width="200" height="100"></canvas>
```

Hint: Note: Always specify an id attribute (to be referred to in a script), and a width and height attribute to define the size of the canvas. To add a border, use the style attribute.

Example. 85: Basic Convas

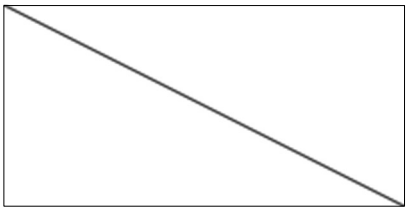
```
<canvas id="myCanvas" width="200" height="100" style="border:1px solid
#000000;">
</canvas>
```

12.3. Add a JavaScript

After creating the rectangular canvas area, you must add a JavaScript to do the drawing.

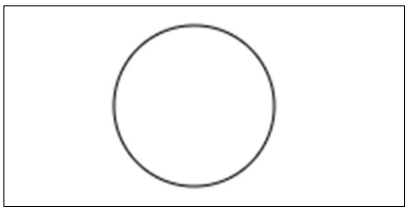
Example. 86: Draw a Line

```
<script>
	var c = document.getElementById("myCanvas");
	var ctx = c.getContext("2d");
	ctx.moveTo(0, 0);
	ctx.lineTo(200, 100);
	ctx.stroke();
</script>
```



Example. 87: Draw a Circle

```
<script>
        var c = document.getElementById("myCanvas");
        var ctx = c.getContext("2d");
        ctx.beginPath();
        ctx.arc(95, 50, 40, 0, 2 * Math.PI);
        ctx.stroke();
</script>
```



Example. 88: Draw a Text

```
<script>
        var c = document.getElementById("myCanvas");
        var ctx = c.getContext("2d");
        ctx.font = "30px Arial";
        ctx.fillText("Hello World", 10, 50);
</script>
```

Example. 89: Draw a Stroke Text

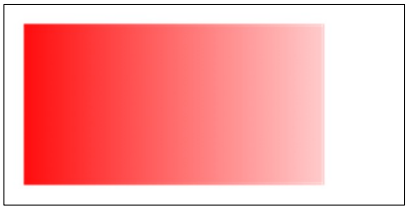
```
<script>
        var c = document.getElementById("myCanvas");
        var ctx = c.getContext("2d");
        ctx.font = "30px Arial";
        ctx.strokeText("Hello World", 10, 50);
</script>
```

Example. 90: Draw a Gradient

```
<script>
        var c = document.getElementById("myCanvas");
        var ctx = c.getContext("2d");

        // Create gradient
        var grd = ctx.createLinearGradient(0, 0, 200, 0);
        grd.addColorStop(0, "red");
        grd.addColorStop(1, "white");

        // Fill with gradient
        ctx.fillStyle = grd;
        ctx.fillRect(10, 10, 150, 80);
</script>
```



Example. 91: Draw Circular Gradient

```
<script>
        var c = document.getElementById("myCanvas");
        var ctx = c.getContext("2d");

        // Create gradient
        var grd = ctx.createRadialGradient(75, 50, 5, 90, 60, 100);
        grd.addColorStop(0, "red");
        grd.addColorStop(1, "white");

        // Fill with gradient
        ctx.fillStyle = grd;
        ctx.fillRect(10, 10, 150, 80);
</script>
```