About the Tutorial

ECMAScript (ES) is a scripting language specification standardized by ECMAScript International. It is used by applications to enable client-side scripting. Languages like JavaScript, Jscript and ActionScript are governed by this specification.

This tutorial introduces you to ES6 implementation in JavaScript.

Audience

This tutorial has been prepared for JavaScript developers who are keen on knowing the difference between ECMAScript 5 and ECMAScript 6. It is useful for those who want to learn the latest developments in the language and implement the same in JavaScript.

Prerequisites

You need to have a basic understanding of JavaScript to make the most of this tutorial.

Disclaimer & Copyright

© Copyright 2016 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com.
Table of Contents

About the Tutorial ............................................................................................................. i
Audience ............................................................................................................................... i
Prerequisites ......................................................................................................................... i
Disclaimer & Copyright ..................................................................................................... i
Table of Contents .............................................................................................................. ii

1. ES6 – OVERVIEW .......................................................................................................... 1
   JavaScript ......................................................................................................................... 1

2. ES6 – ENVIRONMENT ................................................................................................... 2
   Local Environment Setup ............................................................................................... 2
   Installation on Windows ................................................................................................. 2
   Installation on Mac OS X ............................................................................................... 3
   Installation on Linux ....................................................................................................... 4
   Integrated Development Environment (IDE) Support ................................................ 4
   Visual Studio Code ......................................................................................................... 4
   Brackets .......................................................................................................................... 7

3. ES6 – SYNTAX ................................................................................................................ 10
   Whitespace and Line Breaks ....................................................................................... 11
   Comments in JavaScript .............................................................................................. 11
   Your First JavaScript Code .......................................................................................... 12
   Executing the Code ...................................................................................................... 12
   Node.js and JS/ES6 ....................................................................................................... 13
   The Strict Mode ........................................................................................................... 13
   ES6 and Hoisting .......................................................................................................... 14
4. ES6 – VARIABLES

<table>
<thead>
<tr>
<th>Type Syntax</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript and Dynamic Typing</td>
<td>16</td>
</tr>
<tr>
<td>JavaScriptVariable Scope</td>
<td>16</td>
</tr>
<tr>
<td>The Let and Block Scope</td>
<td>17</td>
</tr>
<tr>
<td>The const</td>
<td>18</td>
</tr>
<tr>
<td>ES6 and Variable Hoisting</td>
<td>18</td>
</tr>
</tbody>
</table>

5. ES6 – OPERATORS

<table>
<thead>
<tr>
<th>Arithmetic Operators</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational Operators</td>
<td>22</td>
</tr>
<tr>
<td>Logical Operators</td>
<td>23</td>
</tr>
<tr>
<td>Bitwise Operators</td>
<td>24</td>
</tr>
<tr>
<td>Assignment Operators</td>
<td>26</td>
</tr>
<tr>
<td>Miscellaneous Operators</td>
<td>27</td>
</tr>
<tr>
<td>Type Operators</td>
<td>29</td>
</tr>
</tbody>
</table>

6. ES6 – DECISION MAKING

<table>
<thead>
<tr>
<th>The if Statement</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>The if...else Statement</td>
<td>32</td>
</tr>
<tr>
<td>The else...if Ladder</td>
<td>33</td>
</tr>
<tr>
<td>The switch...case Statement</td>
<td>34</td>
</tr>
</tbody>
</table>

7. ES6 – LOOPS

<table>
<thead>
<tr>
<th>Definite Loop</th>
<th>38</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indefinite Loop</td>
<td>41</td>
</tr>
<tr>
<td>The Loop Control Statements</td>
<td>44</td>
</tr>
<tr>
<td>Using Labels to Control the Flow</td>
<td>45</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>ES6 – FUNCTIONS</td>
</tr>
<tr>
<td></td>
<td>Classification of Functions</td>
</tr>
<tr>
<td></td>
<td>Rest Parameters</td>
</tr>
<tr>
<td></td>
<td>Anonymous Function</td>
</tr>
<tr>
<td></td>
<td>The Function Constructor</td>
</tr>
<tr>
<td></td>
<td>Recursion and JavaScript Functions</td>
</tr>
<tr>
<td></td>
<td>Lambda Functions</td>
</tr>
<tr>
<td></td>
<td>Function Expression and Function Declaration</td>
</tr>
<tr>
<td></td>
<td>Function Hoisting</td>
</tr>
<tr>
<td></td>
<td>Immediately Invoked Function Expression</td>
</tr>
<tr>
<td></td>
<td>Generator Functions</td>
</tr>
<tr>
<td>9</td>
<td>ES6 – EVENTS</td>
</tr>
<tr>
<td></td>
<td>Event Handlers</td>
</tr>
<tr>
<td></td>
<td>onclick Event Type</td>
</tr>
<tr>
<td></td>
<td>onsubmitEvent Type</td>
</tr>
<tr>
<td></td>
<td>onmouseover and onmouseout</td>
</tr>
<tr>
<td></td>
<td>HTML 5 Standard Events</td>
</tr>
<tr>
<td>10</td>
<td>ES6 – COOKIES</td>
</tr>
<tr>
<td></td>
<td>How It Works?</td>
</tr>
<tr>
<td></td>
<td>Storing Cookies</td>
</tr>
<tr>
<td></td>
<td>Reading Cookies</td>
</tr>
<tr>
<td></td>
<td>Setting Cookies Expiry Date</td>
</tr>
<tr>
<td></td>
<td>Deleting a Cookie</td>
</tr>
<tr>
<td>11</td>
<td>ES6 – PAGE REDIRECT</td>
</tr>
<tr>
<td></td>
<td>JavaScript Page Redirection</td>
</tr>
<tr>
<td></td>
<td>Redirection and Search Engine Optimization</td>
</tr>
</tbody>
</table>
12. ES6 – DIALOG BOXES
- Alert Dialog Box ................................................................. 79
- Confirmation Dialog Box .................................................. 80
- Prompt Dialog Box ............................................................ 81

13. ES6 – VOID KEYWORD
- Void and Immediately Invoked Function Expressions ............. 83
- Void and JavaScript URIs .................................................... 83

14. ES6 – PAGE PRINTING ......................................................... 85

15. ES6 – OBJECTS
- Object Initializers ............................................................... 86
- The Object() Constructor .................................................. 87
- Constructor Function ......................................................... 89
- The Object.create Method ................................................. 91
- The Object.assign() Function .......................................... 91
- Deleting Properties .......................................................... 93
- Comparing Objects .......................................................... 93
- Object De-structuring ....................................................... 94

16. ES6 – NUMBER ................................................................. 95
- Number Properties .......................................................... 95
- EPSILON ............................................................................. 96
- MAX_SAFE_INTEGER .......................................................... 96
- MAX_VALUE ........................................................................ 96
- MIN_SAFE_INTEGER ............................................................. 97
- MIN_VALUE ........................................................................ 97
- Nan ................................................................................... 98
Length ................................................................. 115
Prototype ............................................................... 115
String Methods ......................................................... 116
charAt ................................................................. 117
charCodeAt() ......................................................... 118
concat() .............................................................. 119
indexOf() ............................................................. 119
lastIndexOf() .......................................................... 120
localeCompare() ..................................................... 121
replace() ............................................................. 122
search() .............................................................. 123
slice() ................................................................. 124
split() ................................................................. 125
substr() .............................................................. 125
substring() ........................................................... 126
toLocaleLowerCase() ................................................. 127
toLowerCase() ........................................................ 127
toString() ............................................................ 128
toUpperCase() ........................................................ 128
valueOf() .............................................................. 129

19. ES6 – NEW STRING METHODS .................................................. 130
startsWith ............................................................ 130
endsWith .............................................................. 131
includes() ............................................................ 131
repeat() ............................................................... 132
Template Literals ...................................................... 133
Multiline Strings and Template Literals ................................................................. 134
String.raw() ........................................................................................................ 134
String.fromCharCode() ...................................................................................... 135

20. ES6 – ARRAYS .................................................................................................. 136
Features of an Array ............................................................................................. 136
Declaring and Initializing Arrays ......................................................................... 136
Accessing Array Elements ................................................................................... 137
Array Object .......................................................................................................... 138
Array Methods ....................................................................................................... 139
concat() ............................................................................................................... 140
every() ................................................................................................................. 141
filter() .................................................................................................................. 141
forEach() ............................................................................................................. 142
indexOf() ............................................................................................................. 143
join() .................................................................................................................... 144
lastIndexOf() .................................................................................................... 145
map() ................................................................................................................... 146
pop() .................................................................................................................... 146
push() .................................................................................................................. 147
reduce() .............................................................................................................. 148
reduceRight() ................................................................................................... 148
reverse() ............................................................................................................. 149
shift() ................................................................................................................... 150
slice() .................................................................................................................. 150
some() .................................................................................................................. 151
sort() ..................................................................................................................... 152
ES6

splice() ................................................................................................................................................. 152

toString() .................................................................................................................................................. 153

unshift() .................................................................................................................................................... 154

ES6 − Array Methods .................................................................................................................................. 154

Array Traversal using for...in loop ........................................................................................................ 157

Arrays in JavaScript .................................................................................................................................... 157

Array De-structuring .................................................................................................................................. 160

21. ES6 − DATE ........................................................................................................................................ 161

Date Properties .......................................................................................................................................... 161

Constructor ............................................................................................................................................... 161

prototype .................................................................................................................................................... 162

Date Methods ............................................................................................................................................. 163

Date() ......................................................................................................................................................... 165

getDate() ................................................................................................................................................... 165

gOfDay() .................................................................................................................................................... 166

getFullYear() ........................................................................................................................................... 166

getHours() ................................................................................................................................................ 167

getMilliseconds() ................................................................................................................................... 167

getMinutes() ............................................................................................................................................ 168

getMonth() ............................................................................................................................................... 168

getSeconds() ............................................................................................................................................ 169

getTime() ............................................................................................................................................... 169

getTimezoneOffset() .................................................................................................................................. 170

getUTCDate() ........................................................................................................................................... 170

getUTCDay() ........................................................................................................................................... 171

getUTCFullYear() .................................................................................................................................... 171
getUTCHours() ................................................................................................................................. 172
getUTCMilliseconds() ......................................................................................................................... 172
getUTCMonth() ..................................................................................................................................... 173
getUTCMilliseconds() ......................................................................................................................... 173
getUTCHours() ................................................................................................................................. 174
setDate() ............................................................................................................................................... 174
setFullYear() ..................................................................................................................................... 175
setHours() ............................................................................................................................................ 175
setMilliseconds() ................................................................................................................................ 176
setMinutes() ...................................................................................................................................... 177
setMonth() ......................................................................................................................................... 177
set_seconds() ..................................................................................................................................... 178
setTime() ............................................................................................................................................. 178
setUTCDate() ...................................................................................................................................... 179
setUTCFullYear() .................................................................................................................................. 180
setUTCHours() .................................................................................................................................... 180
setUTCMilliseconds() .......................................................................................................................... 181
setUTCMonth() ..................................................................................................................................... 182
setUTCMilliseconds() .......................................................................................................................... 182
setUTCMonth() ..................................................................................................................................... 183
setUTCSeconds() .................................................................................................................................. 184
toDateString() ..................................................................................................................................... 184
toLocaleDateString() ............................................................................................................................ 184
toDateString() ..................................................................................................................................... 185
toLocaleString() ................................................................................................................................... 185
toLocaleTimeString() ............................................................................................................................. 186
toString() .............................................................................................................................................. 187
toTimeString() ...................................................................................................................................... 187
toUTCString() ....................................................................................................................................... 188
22. ES6 – MATH

Math Properties

Math- E

Math- LN2

Math- LN10

Math- LOG2E

Math - LOG10E

Math- PI

Math- SQRT1_2

Math - SQRT2

Exponential Functions

Pow()

sqrt()

cbrt()

exp()

expm1(X)

Math.hypot(x1, x2,...)

Logarithmic Functions

Math.log(x)

Math.log10(x)

Math.log2(x)

Math.log1p(x)

Miscellaneous Algebraic Functions

Abs()

sign()
round() .......................................................... 202
trunc() ................................................................ 202
floor() .................................................................. 203
ceil() ................................................................... 203
min() ................................................................... 204
max() ................................................................... 205
Trigonometric Functions ........................................... 205
Math.sin(x) .......................................................... 206
Math.cos(x) .......................................................... 206
Math.tan(x) .......................................................... 207
Math.asin(x) .......................................................... 207
Math.acos(x) .......................................................... 208
Math.atan(x) .......................................................... 209
Math.atan2() .......................................................... 209
Math.random() ......................................................... 210

23. ES6 – REGEXP .................................................. 211
   Constructing Regular Expressions ......................... 211
   Meta-characters .................................................... 214
   RegExp Properties .................................................. 215
   RegExp Constructor .................................................. 215
   global ................................................................ 216
   ignoreCase .......................................................... 217
   lastIndex ............................................................. 218
   multiline ............................................................. 218
   source .................................................................. 219
   RegExp Methods .................................................... 220
exec() ............................................................................................................................. 220

test() .................................................................................................................................. 221

match() ............................................................................................................................... 222

replace() ............................................................................................................................ 222

search() ................................................................................................................................ 223

split() ................................................................................................................................... 224

toString() ............................................................................................................................. 225

24. ES6 – HTML DOM ........................................................................................................ 226

The Legacy DOM ................................................................................................................ 227

Document Properties in Legacy DOM ................................................................................. 227

Document Methods in Legacy DOM ..................................................................................... 229

25. ES6 – COLLECTIONS ..................................................................................................... 232

Maps ...................................................................................................................................... 232

Understanding basic Map operations ................................................................................. 233

Map Methods ....................................................................................................................... 235

clear() .................................................................................................................................. 235

delete(key) ........................................................................................................................... 236

entries() ............................................................................................................................... 237

forEach .................................................................................................................................. 237

keys() .................................................................................................................................. 238

values() ................................................................................................................................ 239

The for...of Loop ................................................................................................................ 239

Weak Maps ........................................................................................................................... 240

Sets ....................................................................................................................................... 240

Set Properties ....................................................................................................................... 241

Set Methods .......................................................................................................................... 241
add() ................................................................................................................................. 242
clear() .................................................................................................................................. 243
delete() .................................................................................................................................. 243
entries() .............................................................................................................................. 244
forEach ..................................................................................................................................... 245
has() ....................................................................................................................................... 245
values() and keys() .............................................................................................................. 246
Weak Set ................................................................................................................................. 248
Iterator ..................................................................................................................................... 248

26. ES6 – CLASSES .............................................................................................................. 251
   Object-Oriented Programming Concepts ........................................................................... 251
   Creating Objects .................................................................................................................. 253
   Accessing Functions .......................................................................................................... 253
   The Static Keyword ............................................................................................................ 254
   The instanceof operator ..................................................................................................... 255
   Class Inheritance .............................................................................................................. 255
   Class Inheritance and Method Overriding ....................................................................... 257
   The Super Keyword .......................................................................................................... 257

27. ES6 – PROMISES ............................................................................................................ 259
   Understanding Callback ...................................................................................................... 259
   Understanding AsyncCallback ............................................................................................ 260

28. ES6 – MODULES ............................................................................................................ 266
   Exporting a Module ............................................................................................................ 266
   Importing a Module ............................................................................................................ 266
29. ES6 − ERROR HANDLING ........................................................................................................................................... 269
   Syntax Errors ......................................................................................................................................................... 269
   Runtime Errors ....................................................................................................................................................... 269
   Logical Errors ......................................................................................................................................................... 269
   Throwing Exceptions ............................................................................................................................................... 270
   Exception Handling ............................................................................................................................................... 270
   The onerror() Method ........................................................................................................................................... 272
   Custom Errors ......................................................................................................................................................... 273

30. ES6 − VALIDATIONS .................................................................................................................................................... 275
   Basic Form Validation ............................................................................................................................................... 277
   Data Format Validation ......................................................................................................................................... 278

31. ES6 − ANIMATION ....................................................................................................................................................... 279
   Manual Animation ................................................................................................................................................... 280
   Automated Animation ............................................................................................................................................ 281
   Rollover with a Mouse Event ................................................................................................................................. 282

32. ES6 − MULTIMEDIA ..................................................................................................................................................... 284
   Checking for Plugins ............................................................................................................................................. 285
   Controlling Multimedia ........................................................................................................................................ 286

33. ES6 − DEBUGGING ..................................................................................................................................................... 288
   Error Messages in IE ............................................................................................................................................... 288
   Error Messages in Firefox or Mozilla .................................................................................................................... 289
   Error Notifications ............................................................................................................................................... 289
   Debugging a Script .................................................................................................................................................. 289
   Useful Tips for Developers ................................................................................................................................... 290
   Debugging with Node.js ......................................................................................................................................... 291
ES6 – Overview

ECMAScript (ES) is a scripting language specification standardized by ECMAScript International. It is used by applications to enable client-side scripting. The specification is influenced by programming languages like Self, Perl, Python, Java etc. Languages like JavaScript, Jscript and ActionScript are governed by this specification.

This tutorial introduces you to ES6 implementation in JavaScript.

JavaScript

JavaScript was developed by Brendan Eich, a developer at Netscape Communications Corporation, in 1995. JavaScript started life with the name Mocha, and was briefly named LiveScript before being officially renamed to JavaScript. It is a scripting language that is executed by the browser, i.e. on the client’s end. It is used in conjunction with HTML to develop responsive webpages.

ECMA Script6’s implementation discussed here covers the following new features:

- Support for constants
- Block Scope
- Arrow Functions
- Extended Parameter Handling
- Template Literals
- Extended Literals
- Enhanced Object Properties
- De-structuring Assignment
- Modules
- Classes
- Iterators
- Generators
- Collections
- New built-in methods for various classes
- Promises
In this chapter, we will discuss the setting up of the environment for ES6.

Local Environment Setup

JavaScript can run on any browser, any host, and any OS. You will need the following to write and test a JavaScript program standard:

Text Editor

The text editor helps you to write your source code. Examples of few editors include Windows Notepad, Notepad++ , Emacs, vim or vi etc. Editors used may vary with the operating systems. The source files are typically named with the extension .js.

Installing Node.js

Node.js is an open source, cross-platform runtime environment for server-side JavaScript. Node.js is required to run JavaScript without a browser support. It uses Google V8 JavaScript engine to execute the code. You may download Node.js source code or a pre-built installer for your platform. Node is available at https://nodejs.org/en/download

Installation on Windows

Download and run the .msi installer for Node.
To verify if the installation was successful, enter the command `node -v` in the terminal window.

```
C:\Users>node -v
v4.2.3
C:\Users>
```

**Installation on Mac OS X**

To install node.js on OS X you can download a pre-compiled binary package which makes a nice and easy installation. Head over to http://nodejs.org/ and click the install button to download the latest package.

Install the package from the `.dmg` by following along the install wizard which will install both `node` and `npm`. npm is the Node Package Manager which facilitates installs of additional packages for Node.js.
Installation on Linux

You need to install a number of dependencies before you can install Node.js and npm.

- **Ruby** and **GCC**. You’ll need Ruby 1.8.6 or newer and GCC 4.2 or newer.
- **Homebrew**. Homebrew is a package manager originally for the Mac, but it’s been ported to Linux as Linuxbrew. You can learn more about Homebrew at the [http://brew.sh](http://brew.sh) and Linuxbrew at the [http://brew.sh/linuxbrew](http://brew.sh/linuxbrew).

Integrated Development Environment (IDE) Support

JavaScript can be built on a plethora of development environments like Visual Studio, Sublime Text 2, WebStorm/PHPStorm, Eclipse, Brackets, etc. The Visual Studio Code and Brackets IDE is discussed in this section. The development environment used here is Visual Studio Code (Windows platform).

**Visual Studio Code**

This is open source IDE from Visual Studio. It is available for Mac OS X, Linux, and Windows platforms. VScode is available at [https://code.visualstudio.com](https://code.visualstudio.com)

**Installation on Windows**

Double-click on VSCodeSetup.exe to launch the setup process. This will only take a minute.
Following is the screenshot of the IDE.

You may directly traverse to the file's path by a right-click on the file -> open in command prompt. Similarly, the **Reveal in Explorer** option shows the file in the File Explorer.
Installation on Mac OS X

Visual Studio Code’s Mac OS X specific installation guide can be found at https://code.visualstudio.com/Docs/editor/setup

Installation on Linux

Linux specific installation guide for Visual Studio Code can be found at https://code.visualstudio.com/Docs/editor/setup

Brackets

Brackets is a free open-source editor for web development, created by Adobe Systems. It is available for Linux, Windows and Mac OS X. Brackets is available at http://brackets.io
You can run DOS prompt/Shell within Brackets itself by adding one more extension Brackets Shell.
Upon installation, you will find an icon of shell on the right hand side of the editor. Once you click on the icon, you will see the shell window as shown in the following screenshot.

You are all set!!!
Syntax defines the set of rules for writing programs. Every language specification defines its own syntax.

A JavaScript program can be composed of:

- **Variables**: Represents a named memory block that can store values for the program.
- **Literals**: Represents constant/fixed values.
- **Operators**: Symbols that define how the operands will be processed.
- **Keywords**: Words that have a special meaning in the context of a language.

The following table lists some keywords in JavaScript. Some commonly used keywords are listed in the following table.

<table>
<thead>
<tr>
<th>break</th>
<th>as</th>
<th>any</th>
<th>Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
<td>if</td>
<td>throw</td>
<td>Else</td>
</tr>
<tr>
<td>var</td>
<td>number</td>
<td>string</td>
<td>Get</td>
</tr>
<tr>
<td>module</td>
<td>type</td>
<td>instanceof</td>
<td>Typeof</td>
</tr>
<tr>
<td>finally</td>
<td>for</td>
<td>enum</td>
<td>Export</td>
</tr>
<tr>
<td>while</td>
<td>void</td>
<td>this</td>
<td>New</td>
</tr>
<tr>
<td>null</td>
<td>super</td>
<td>Catch</td>
<td>let</td>
</tr>
<tr>
<td>static</td>
<td>return</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

- **Modules**: Represents code blocks that can be reused across different programs/scripts.
- **Comments**: Used to improve code readability. These are ignored by the JavaScript engine.
- **Identifiers**: These are the names given to elements in a program like variables, functions, etc. The rules for identifiers are:
  - Identifiers can include both, characters and digits. However, the identifier cannot begin with a digit.
  - Identifiers cannot include special symbols except for underscore (_) or a dollar sign ($).
  - Identifiers cannot be keywords. They must be unique.
  - Identifiers are case sensitive. Identifiers cannot contain spaces.
The following table illustrates some valid and invalid identifiers.

<table>
<thead>
<tr>
<th>Examples of valid identifiers</th>
<th>Examples of invalid identifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstName</td>
<td>Var#</td>
</tr>
<tr>
<td>first_name</td>
<td>first name</td>
</tr>
<tr>
<td>num1</td>
<td>first-name</td>
</tr>
<tr>
<td>$result</td>
<td>1number</td>
</tr>
</tbody>
</table>

### Whitespace and Line Breaks

ES6 ignores spaces, tabs, and newlines that appear in programs. You can use spaces, tabs, and newlines freely in your program and you are free to format and indent your programs in a neat and consistent way that makes the code easy to read and understand.

### JavaScript is Case-sensitive

JavaScript is case-sensitive. This means that JavaScript differentiates between the uppercase and the lowercase characters.

### Semicolons are Optional

Each line of instruction is called a **statement**. Semicolons are optional in JavaScript.

**Example**

```javascript
console.log("hello world")
console.log("We are learning ES6")
```

A single line can contain multiple statements. However, these statements must be separated by a semicolon.

### Comments in JavaScript

Comments are a way to improve the readability of a program. Comments can be used to include additional information about a program like the author of the code, hints about a function/construct, etc. Comments are ignored by the compiler.

JavaScript supports the following types of comments:

- **Single-line comments (//)**: Any text between a // and the end of a line is treated as a comment.

- **Multi-line comments (/*  */)**: These comments may span multiple lines.
Example

```javascript
//this is single line comment

/* This is a
Multi-line comment */
```

Your First JavaScript Code

Let us start with the traditional “Hello World” example”.

```javascript
var message="Hello World"
console.log(message)
```

The program can be analyzed as:

- Line 1 declares a variable by the name message. Variables are a mechanism to store values in a program.

- Line 2 prints the variable’s value to the prompt. Here, the console refers to the terminal window. The function log () is used to display the text on the screen.

Executing the Code

We shall use Node.js to execute our code.

**Step 1:** Save the file as Test.js

**Step 2:** Right-click the Test.js file under the working files option in the project-explorer window of the Visual Studio Code.

**Step 3:** Select Open in Command Prompt option.

**Step 4:** Type the following command in Node’s terminal window.

```bash
node Test.js
```

The following output is displayed on successful execution of the file.

```
Hello World
```
Node.js and JS/ES6

ECMAScript 2015(ES6) features are classified into three groups:

- **For Shipping**: These are features that V8 considers stable.

- **Staged Features**: These are almost completed features but not considered stable by the V8 team.

- **In Progress**: These features should be used only for testing purposes.

The first category of features is fully supported and turned on by default by node. Staged features require a runtime --harmony flag to execute.

A list of component specific CLI flags for Node.js can be found here: https://nodejs.org/api/cli.html

The Strict Mode

The fifth edition of the ECMAScript specification introduced the Strict Mode. The Strict Mode imposes a layer of constraint on JavaScript. It makes several changes to normal JavaScript semantics.

The code can be transitioned to work in the Strict Mode by including the following:

```javascript
// Whole-script strict mode syntax
"use strict";

v = "Hi! I'm a strict mode script!"; // ERROR: Variable v is not declared
```

In the above snippet, the entire code runs as a constrained variant of JavaScript.

JavaScript also allows to restrict, the Strict Mode within a block’s scope as that of a function. This is illustrated as follows:

```javascript
v=15
function f1()
{
"use strict";
var v = "Hi! I'm a strict mode script!";
}
```

In, the above snippet, any code outside the function will run in the non-script mode. All statements within the function will be executed in the Strict Mode.
ES6 and Hoisting

The JavaScript engine, by default, moves declarations to the top. This feature is termed as **hoisting**. This feature applies to variables and functions. Hoisting allows JavaScript to use a component before it has been declared. However, the concept of hoisting does not apply to scripts that are run in the Strict Mode.

Variable Hoisting and Function Hoisting are explained in the subsequent chapters.
A variable, by definition, is “a named space in the memory” that stores values. In other words, it acts as a container for values in a program. Variable names are called identifiers. Following are the naming rules for an identifier:

- Identifiers cannot be keywords.
- Identifiers can contain alphabets and numbers.
- Identifiers cannot contain spaces and special characters, except the underscore (_) and the dollar ($) sign.
- Variable names cannot begin with a number.

**Type Syntax**

A variable must be declared before it is used. ES5 syntax used the `var` keyword to achieve the same. The ES5 syntax for declaring a variable is as follows:

```javascript
//Declaration using var keyword
var variable_name
```

ES6 introduces the following variable declaration syntax:

- Using the `let`
- Using the `const`

**Variable initialization** refers to the process of storing a value in the variable. A variable may be initialized either at the time of its declaration or at a later point in time.

The traditional ES5 type syntax for declaring and initializing a variable is as follows:

```javascript
//Declaration using var keyword
var variable_name=value
```

**Example: Using Variables**

```javascript
var name="Tom"
console.log("The value in the variable is: "+name);
```

The above example declares a variable and prints its value.

The following output is displayed on successful execution.

```
The value in the variable is Tom
```
JavaScript and Dynamic Typing

JavaScript is an un-typed language. This means that a JavaScript variable can hold a value of any data type. Unlike many other languages, you don't have to tell JavaScript during variable declaration what type of value the variable will hold. The value type of a variable can change during the execution of a program and JavaScript takes care of it automatically. This feature is termed as *dynamic typing*.

JavaScript Variable Scope

The scope of a variable is the region of your program in which it is defined. Traditionally, JavaScript defines only two scopes—global and local.

- **Global Scope**: A variable with global scope can be accessed from within any part of the JavaScript code.
- **Local Scope**: A variable with a local scope can be accessed from within a function where it is declared.

Example: Global vs. Local Variable

The following example declares two variables by the name `num` - one outside the function (global scope) and the other within the function (local scope).

```javascript
var num=10
function test()
{
    var num=100
    console.log("value of num in test() "+num)
}
console.log("value of num outside test() "+num)
test()
```

The variable when referred to within the function displays the value of the locally scoped variable. However, the variable `num` when accessed outside the function returns the globally scoped instance.

The following output is displayed on successful execution.

```
value of num outside test() 10
value of num outside test() 100
```

ES6 defines a new variable scope - The Block scope.
The Let and Block Scope

The block scope restricts a variable’s access to the block in which it is declared. The `var` keyword assigns a function scope to the variable. Unlike the `var` keyword, the `let` keyword allows the script to restrict access to the variable to the nearest enclosing block.

```javascript
"use strict"
function test()
{
    var num=100
    console.log("value of num in test() "+num)
    {
        console.log("Inner Block begins")
        let num=200
        console.log("value of num : "+num)
    }
}
test()
```

The script declares a variable `num` within the local scope of a function and re-declares it within a block using the `let` keyword. The value of the locally scoped variable is printed when the variable is accessed outside the inner block, while the block scoped variable is referred to within the inner block.

**Note:** The strict mode is a way to opt in to a restricted variant of JavaScript.

The following output is displayed on successful execution.

```
value of num in test() 100
Inner Block begins
value of num : 200
```

**Example: let v/s var**

```javascript
var no =10;
var no =20;
console.log(no);
```

The following output is displayed on successful execution of the above code.

```
20
```
Let us re-write the same code using the `let` keyword.

```javascript
let no = 10;
let no = 20;
console.log(no);
```

The above code will throw an error: Identifier 'no' has already been declared. Any variable declared using the `let` keyword is assigned the block scope.

### The `const`

The `const` declaration creates a read-only reference to a value. It does not mean the value it holds is immutable, just that the variable identifier cannot be reassigned. Constants are block-scoped, much like variables defined using the `let` statement. The value of a constant cannot change through re-assignment, and it can't be re-declared.

The following rules hold true for a variable declared using the `const` keyword:

- Constants cannot be reassigned a value.
- A constant cannot be re-declared.
- A constant requires an initializer. This means constants must be initialized during its declaration.
- The value assigned to a `const` variable is mutable.

**Example**

```javascript
const x = 10
x = 12 // will result in an error!!
```

The above code will return an error since constants cannot be reassigned a value. Constants variable are immutable.

### ES6 and Variable Hoisting

The scope of a variable declared with `var` is its current execution context, which is either the enclosing function or, for variables declared outside any function, global. Variable hoisting allows the use of a variable in a JavaScript program, even before it is declared.

The following example better explains this concept.

**Example: Variable Hoisting**

```javascript
var main = function()
{
    for(var x=0;x<5;x++)
```
```javascript
{    
    console.log(x);
}

console.log("x can be accessed outside the block scope x value is ":"+x);
console.log('x is hoisted to the function scope');

main();
```

The following output is displayed on successful execution of the above code:

```
0 1 2 3 4
x can be accessed outside the block scope x value is :5
x is hoisted to the function scope
```

The JavaScript engine internally represents the script as:

```javascript
var main = function()
{
    
    var x; // x is hoisted to function scope
    for( x=0;x<5;x++)
    {
        console.log(x);
    }
    
    console.log("x can be accessed outside the block scope x value is ":"+x);
    console.log('x is hoisted to the function scope');
}

main();
```

**Note:** The concept of hoisting applies to variable declaration but not variable initialization. It is recommended to always declare variables at the top of their scope (the top of global code and the top of function code), to enable the code resolve the variable's scope.
An **expression** is a special kind of statement that evaluates to a value. Every expression is composed of:

- **Operands**: Represents the data.
- **Operator**: Defines how the operands will be processed to produce a value.

Consider the following expression - 2 + 3. Here in the expression, 2 and 3 are operands and the symbol + (plus) is the operator. JavaScript supports the following types of operators:

- Arithmetic operators
- Logical operators
- Relational operators
- Bitwise operators
- Assignment operators
- Ternary/conditional operators
- String operators
- Type operators
- The void operator

### Arithmetic Operators

Assume the values in variables \( a \) and \( b \) are 10 and 5 respectively.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>Addition: Returns the sum of the operands</td>
<td>( a + b ) is 15</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction: Returns the difference of the values</td>
<td>( a - b ) is 5</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiplication: Returns the product of the values</td>
<td>( a \times b ) is 50</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division: Performs a division operation and returns the quotient</td>
<td>( a / b ) is 2</td>
</tr>
<tr>
<td><code>%</code></td>
<td></td>
<td>( a % b ) is 2</td>
</tr>
</tbody>
</table>
**Example: Arithmetic Operators**

```javascript
var num1=10
var num2=2
var res=0
res= num1+num2
console.log("Sum:        "+ res);
res=num1-num2;
console.log("Difference: "+res)
res=num1*num2
console.log("Product:    "+res)
res=num1/num2
console.log("Quotient:   "+res)
res=num1%num2
console.log("Remainder:   "+res)
num1++
console.log("Value of num1 after increment "+num1) 
num2--
console.log("Value of num2 after decrement "+num2)
```

The following output is displayed on successful execution of the above program.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>Increments the value of the variable by one</td>
<td>a++ is 11</td>
</tr>
<tr>
<td>--</td>
<td>Decrements the value of the variable by one</td>
<td>A-- is 9</td>
</tr>
</tbody>
</table>

**Modulus:** Performs a division and returns the remainder
Relational Operators

Relational operators test or define the kind of relationship between two entities. Relational operators return a boolean value, i.e., true/false.

Assume the value of A is 10 and B is 20.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>(A &gt; B) is False</td>
</tr>
<tr>
<td>&lt;</td>
<td>Lesser than</td>
<td>(A &lt; B) is True</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>(A &gt;= B) is False</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Lesser than or equal to</td>
<td>(A &lt;= B) is True</td>
</tr>
<tr>
<td>==</td>
<td>Equality</td>
<td>(A == B) is True</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
<td>(A != B) is True</td>
</tr>
</tbody>
</table>

Example

```javascript
var num1 = 5;
var num2 = 9;
console.log("Value of num1: " + num1);
console.log("Value of num2: " + num2);
var res = num1 > num2;
console.log("num1 greater than num2: " + res);
res = num1 < num2;
console.log("num1 lesser than num2: " + res);
res = num1 >= num2;
console.log("num1 greater than or equal to num2: " + res);
res = num1 <= num2;
console.log("num1 lesser than or equal to num2: " + res);
res = num1 == num2;
console.log("num1 is equal to num2: " + res);
res = num1 != num2;
console.log("num1 not equal to num2: " + res);
```
The following output is displayed on successful execution of the above code.

<table>
<thead>
<tr>
<th>Value of num1: 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of num2 :9</td>
</tr>
<tr>
<td>num1 greater than num2: false</td>
</tr>
<tr>
<td>num1 lesser than num2: true</td>
</tr>
<tr>
<td>num1 greater than or equal to num2: false</td>
</tr>
<tr>
<td>num1 lesser than or equal to num2: true</td>
</tr>
<tr>
<td>14 num1 is equal to num2: false</td>
</tr>
<tr>
<td>16 num1 not equal to num2: true</td>
</tr>
</tbody>
</table>

**Logical Operators**

Logical operators are used to combine two or more conditions. Logical operators, too, return a Boolean value. Assume the value of variable A is 10 and B is 20.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>And: The operator returns true only if all the expressions specified return true</td>
<td>((A &gt; 10 &amp;&amp; B &gt; 10)) is False</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>NOT: The operator returns the inverse of the expression’s result. For E.g.: (!(7&gt;5)) returns false</td>
<td>(! (A &gt;10 )) is True</td>
</tr>
</tbody>
</table>

**Example**

```javascript
var avg = 20;
var percentage = 90;
console.log("Value of avg: "+ avg + ",value of percentage: "+ percentage);
var res = ((avg > 50) && (percentage > 80));
console.log("(avg>50)&&(percentage>80): ", res);
var res = ((avg > 50) || (percentage > 80));
console.log("(avg>50)||(percentage>80): ", res);
var res = !((avg > 50) && (percentage > 80));
console.log("!(avg>50)&&(percentage>80)): ", res);
```
The following output is displayed on successful execution of the above code.

<table>
<thead>
<tr>
<th>Value of avg: 20 ,value of percentage: 90</th>
</tr>
</thead>
<tbody>
<tr>
<td>(avg&gt;50)&amp;&amp;(percentage&gt;80): false</td>
</tr>
<tr>
<td>(avg&gt;50)</td>
</tr>
<tr>
<td>!((avg&gt;50)&amp;&amp;(percentage&gt;80)): true</td>
</tr>
</tbody>
</table>

**Short-circuit Operators**

The `&&` and `||` operators are used to combine expressions.

The `&&` operator returns true only when both the conditions return true. Let us consider an expression:

```javascript
var a=10
var result=( a<10 && a>5)
```

In the above example, `a<10` and `a>5` are two expressions combined by an `&&` operator. Here, the first expression returns false. However, the `&&` operator requires both the expressions to return true. So, the operator skips the second expression.

The `||` operator returns true, if one of the expressions return true. For example:

```javascript
var a=10
var result=( a>5 || a<10)
```

In the above snippet, two expressions `a>5` and `a<10` are combined by a `||` operator. Here, the first expression returns true. Since, the first expression returns true, the `||` operator skips the subsequent expression and returns true.

Due to this behavior of the `&&` and `||` operator, they are called as short-circuit operators.

**Bitwise Operators**

JavaScript supports the following bitwise operators. The following table summarizes JavaScript's bitwise operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwise AND</td>
<td>a &amp; b</td>
<td>Returns a one in each bit position for which the corresponding bits of both operands are ones</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td>a</td>
<td>Returns a one in each bit position for which the corresponding bits of either or both operands are ones</td>
</tr>
<tr>
<td>Bitwise XOR</td>
<td>a ^ b</td>
<td>Returns a one in each bit position for which the corresponding bits of either but not both operands are ones</td>
</tr>
</tbody>
</table>
### Bitwise NOT

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~ a</td>
<td></td>
<td>Inverts the bits of its operand</td>
</tr>
</tbody>
</table>

### Left shift

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;&lt; b</td>
<td></td>
<td>Shifts a in binary representation b (&lt; 32) bits to the left, shifting in zeroes from the right</td>
</tr>
</tbody>
</table>

### Sign-propagating right shift

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &gt;&gt; b</td>
<td></td>
<td>Shifts a in binary representation b (&lt; 32) bits to the right, discarding bits shifted off</td>
</tr>
</tbody>
</table>

### Zero-fill right shift

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &gt;&gt;&gt; b</td>
<td></td>
<td>Shifts a in binary representation b (&lt; 32) bits to the right, discarding bits shifted off, and shifting in zeroes from the left</td>
</tr>
</tbody>
</table>

### Example

```javascript
var a = 2; // Bit presentation 10
var b = 3; // Bit presentation 11
var result;
result = (a & b);
console.log("(a & b) => ", result);
result = (a | b);
console.log("(a | b) => ", result);
result = (a ^ b);
console.log("(a ^ b) => ", result);
result = (~b);
console.log("(~b) => ", result);
result = (a << b);
console.log("(a << b) => ", result);
result = (a >> b);
console.log("(a >> b) => ", result);
```

### Output

```
(a & b) => 2
(a | b) => 3
(a ^ b) => 1
(~b) => -4
(a << b) => 16
(a >> b) => 0
```
Assignment Operators

The following table summarizes Assignment operators.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Operator and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>= (Simple Assignment)</td>
</tr>
<tr>
<td></td>
<td>Assigns values from the right side operand to the left side operand</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong>: C = A + B will assign the value of A + B into C</td>
</tr>
<tr>
<td>2</td>
<td>+= (Add and Assignment)</td>
</tr>
<tr>
<td></td>
<td>It adds the right operand to the left operand and assigns the result to the left operand.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong>: C += A is equivalent to C = C + A</td>
</tr>
<tr>
<td>3</td>
<td>-= (Subtract and Assignment)</td>
</tr>
<tr>
<td></td>
<td>It subtracts the right operand from the left operand and assigns the result to the left operand.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong>: C -= A is equivalent to C = C - A</td>
</tr>
<tr>
<td>4</td>
<td>*= (Multiply and Assignment)</td>
</tr>
<tr>
<td></td>
<td>It multiplies the right operand with the left operand and assigns the result to the left operand.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong>: C *= A is equivalent to C = C * A</td>
</tr>
<tr>
<td>5</td>
<td>/= (Divide and Assignment)</td>
</tr>
<tr>
<td></td>
<td>It divides the left operand with the right operand and assigns the result to the left operand.</td>
</tr>
</tbody>
</table>

**Note**: The same logic applies to Bitwise operators, so they will become <<=, >>=, >>>, &=, |= and ^=

**Example**

```
var a = 12;
var b = 10;
a = b;
```
```javascript
console.log("a=b: " + a);
a += b;
console.log("a+=b: " + a);
a -= b;
console.log("a-=b: " + a);
a *= b;
console.log("a*=b: " + a);
a /= b;
console.log("a/=b: " + a);
a %= b;
console.log("a%=b: " + a);
```

The following output is displayed on successful execution of the above program.

```
a=b: 10
a+=b: 20
a-=b: 10
a*=b: 100
a/=b: 10
a%=b: 0
```

### Miscellaneous Operators

Following are some of the miscellaneous operators.

**The negation operator (-)**

Changes the sign of a value. The following program is an example of the same.

```javascript
var x=4
var y=-x;
console.log("value of x: ",x); //outputs 4
console.log("value of y: ",y); //outputs -4
```

The following output is displayed on successful execution of the above program.

```
value of x: 4
value of y: -4
```
String Operators: Concatenation operator (+)
The + operator when applied to strings appends the second string to the first. The following program helps to understand this concept.

```javascript
var msg="hello"+"world"
console.log(msg)
```
The following output is displayed on successful execution of the above program.

```
helloworld
```
The concatenation operation doesn’t add a space between the strings. Multiple strings can be concatenated in a single statement.

Conditional Operator (?)
This operator is used to represent a conditional expression. The conditional operator is also sometimes referred to as the ternary operator. Following is the syntax.

```
Test ? expr1 : expr2
```
Where,

- **Test**: Refers to the conditional expression
- **expr1**: Value returned if the condition is true
- **expr2**: Value returned if the condition is false

Example

```javascript
var num=-2
var result= num > 0 ?"positive":"non-positive"
console.log(result)
```
Line 2 checks whether the value in the variable num is greater than zero. If num is set to a value greater than zero, it returns the string "positive" else a "non-positive" string is returned.

The following output is displayed on successful execution of the above program.

```
non-positive
```
Type Operators

typeof operator

It is a unary operator. This operator returns the data type of the operand. The following table lists the data types and the values returned by the `typeof` operator in JavaScript.

<table>
<thead>
<tr>
<th>Type</th>
<th>String Returned by typeof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>&quot;number&quot;</td>
</tr>
<tr>
<td>String</td>
<td>&quot;string&quot;</td>
</tr>
<tr>
<td>Boolean</td>
<td>&quot;boolean&quot;</td>
</tr>
<tr>
<td>Object</td>
<td>&quot;object&quot;</td>
</tr>
</tbody>
</table>

The following example code displays the number as the output.

```javascript
var num=12
console.log(typeof num); //output: number
```

The following output is displayed on successful execution of the above code.

```
number
```
A conditional/decision-making construct evaluates a condition before the instruction/s are executed.

Conditional constructs in JavaScript are classified in the following table.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>if statement</td>
<td>An ‘if’ statement consists of a Boolean expression followed by one or more statements</td>
</tr>
<tr>
<td>if...else statement</td>
<td>An ‘if’ statement can be followed by an optional ‘else’ statement, which executes when the Boolean expression is false</td>
</tr>
<tr>
<td>The else.. if ladder / nested if statements</td>
<td>You can use one ‘if’ or ‘else if’ statement inside another ‘if’ or ‘else if’ statement(s)</td>
</tr>
<tr>
<td>switch statement</td>
<td>A ‘switch’ statement allows a variable to be tested for equality against a list of values</td>
</tr>
</tbody>
</table>

**The if Statement**

The ‘if...else’ construct evaluates a condition before a block of code is executed.
Following is the syntax.

```javascript
if(boolean_expression)
{
    // statement(s) will execute if the Boolean expression is true
}
```

If the Boolean expression evaluates to true, then the block of code inside the if statement will be executed. If the Boolean expression evaluates to false, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

**Flowchart**

![Flowchart of if statement]

**Example**

```javascript
var num=5
if (num>0)
{
    console.log("number is positive")
}
```

The following output is displayed on successful execution of the above code.

```
number is positive
```
The above example will print “number is positive” as the condition specified by the if block is true.

**The if...else Statement**

An if can be followed by an optional else block. The else block will execute if the Boolean expression tested by if evaluates to false.

Following is the syntax.

```javascript
if(boolean_expression)
{
    // statement(s) will execute if the Boolean expression is true
}
else
{
    // statement(s) will execute if the Boolean expression is false
}
```

**Flowchart**

The if block guards the conditional expression. The block associated with the if statement is executed if the Boolean expression evaluates to true. The if block may be followed by an optional else statement. The instruction block associated with the else block is executed if the expression evaluates to false.
Example: Simple if...else

```javascript
var num = 12;
if (num % 2 == 0)
{
    console.log("Even");
}
else
{
    console.log("Odd");
}
```

The above example prints whether the value in a variable is even or odd. The if block checks the divisibility of the value by 2 to determine the same.

The following output is displayed on successful execution of the above code.

Even

The else...if Ladder

The else...if ladder is useful to test multiple conditions. Following is the syntax of the same.

```javascript
if (boolean_expression1)
{
    // statements if the expression1 evaluates to true
}
else if (boolean_expression2)
{
    // statements if the expression2 evaluates to true
}
else
{
    // statements if both expression1 and expression2 result to false
}
```

When using if...else statements, there are a few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.
Example: else...if ladder

```javascript
var num = 2
if (num > 0) {
  console.log(num + " is positive")
}
else if (num < 0) {
  console.log(num + " is negative")
}
else {
  console.log(num + " is neither positive nor negative")
}
```

The code displays whether the value is positive, negative, or zero.

The following output is displayed on successful execution of the above code.

```
2 is positive
```

The switch...case Statement

The switch statement evaluates an expression, matches the expression's value to a case clause and executes the statements associated with that case.

Following is the syntax.

```
switch(variable_expression) {
  case constant_expr1:
    {
      //statements;
      break;
    }
  case constant_expr2:
    {
      //statements;
    }
```


```javascript
    break;
    }
default:
    {
        //statements;
        break;
    }
}
```

The value of the `variable_expression` is tested against all cases in the switch. If the variable matches one of the cases, the corresponding code block is executed. If no case expression matches the value of the variable_expression, the code within the default block is associated.

The following rules apply to a switch statement:

- There can be any number of case statements within a switch.

- The case statements can include only constants. It cannot be a variable or an expression.

- The data type of the variable_expression and the constant expression must match.

- Unless you put a break after each block of code, the execution flows into the next block.

- The case expression must be unique.

- The default block is optional.
Example: switch...case

```javascript
var grade="A";
switch(grade)
{
    case "A":
        {
            console.log("Excellent");
            break;
        }
    case "B":
        {
            console.log("Good");
            break;
        }
```
case "C":
{
    console.log("Fair");
    break;
}

case "D":
{
    console.log("Poor");
    break;
}

default:
{
    console.log("Invalid choice");
    break;
}

The following output is displayed on successful execution on the above code.

Excellent

The example verifies the value of the variable grade against the set of constants (A, B, C, D, and E) and executes the corresponding blocks. If the value in the variable doesn’t match any of the constants mentioned above, the default block will be executed.
At times, certain instructions require repeated execution. Loops are an ideal way to do the same. A loop represents a set of instructions that must be repeated. In a loop’s context, a repetition is termed as an **iteration**.

The following figure illustrates the classification of loops:

![Loops Diagram](image)

### Definite Loop
A loop whose number of iterations are definite/fixed is termed as a **definite loop**. The ‘for loop’ is an implementation of a definite loop.

```javascript
for (initial_count_value; termination-condition; step)  
{
   //statements
}
```

### The ‘for’ loop
The for loop executes the code block for a specified number of times. It can be used to iterate over a fixed set of values, such as an array. Following is the syntax of the for loop.

```javascript
var num=5
var factorial=1;
for( let i = num ; i >= 1; i-- )
{
   factorial *= i;
}
console.log(factorial);
```
The for loop has three parts: the initializer (i=num), the condition (i>=1) and the final expression (i--).

The program calculates the factorial of the number 5 and displays the same. The for loop generates the sequence of numbers from 5 to 1, calculating the product of the numbers in every iteration.

Multiple assignments and final expressions can be combined in a for loop, by using the comma operator (,). For example, the following for loop prints the first eight Fibonacci numbers:

**Example**

```
"use strict"
for(let temp, i=0, j=1; j<30; temp = i, i = j, j = i + temp)
    console.log(j);
```

The following output is displayed on successful execution of the above code.

```
1
1
2
3
5
8
13
21
```

**The for...in loop**

The for...in loop is used to loop through an object's properties.

Following is the syntax of 'for...in' loop.

```
for (variblename in object)
{
    statement or block to execute
}
```

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

```javascript
var obj = {a:1, b:2, c:3};

for (var prop in obj) {
    console.log(obj[prop]);
}
```

The above example illustrates iterating an object using the for... in loop. The following output is displayed on successful execution of the code.

1
2
3

The for...of loop

The for...of loop is used to iterate iterables instead of object literals.

Following is the syntax of ‘for...of’ loop.

```javascript
for (variblename of object){
    statement or block to execute
}
```

Example

```javascript
for (let val of[12, 13, 123]){

    console.log(val)
}
```

The following output is displayed on successful execution of the above code.

12
13
123
**Indefinite Loop**

An indefinite loop is used when the number of iterations in a loop is indeterminate or unknown.

Indefinite loops can be implemented using:

- while loop
- do...while loop

**The while loop**

The while loop executes the instructions each time the condition specified evaluates to true. In other words, the loop evaluates the condition before the block of code is executed.

**Flow chart**

```
while(condition)
{
    Conditional code;
}
```

Following is the syntax for the while loop.

```
while (expression)
{
    Statement(s) to be executed if expression is true
}
```
Example

```javascript
var num=5;
var factorial=1;
while(num >=1)
{
    factorial=factorial * num;
    num--;
}
console.log("The factorial is "+factorial);
```

The above code uses a while loop to calculate the factorial of the value in the variable num.

The following output is displayed on successful execution of the code.

The factorial is 120

The do…while loop

The do…while loop is similar to the while loop except that the do…while loop doesn’t evaluate the condition for the first time the loop executes. However, the condition is evaluated for the subsequent iterations. In other words, the code block will be executed at least once in a do…while loop.

Flowchart
Following is the syntax for do-while loop in JavaScript.

```javascript
do{
    Statement(s) to be executed;
} while (expression);
```

**Note:** Don’t miss the semicolon used at the end of the do...while loop.

**Example**

```javascript
var n=10;
do{
    console.log(n);
    n--;
} while(n>=0);
```

The example prints numbers from 0 to 10 in the reverse order.

The following output is displayed on successful execution of the above code.

```
10
9
8
7
6
5
4
3
2
1
0
```

**Example: while versus do...while**

**do...while loop:**

```javascript
var n=10;
do{
    console.log(n);
    n--;
} while(n>=0);
```
while loop:

```javascript
var n=10;
while(n>=0)
{
    console.log(n);
    n--;
}
```

In the above example, the while loop is entered only if the expression passed to while evaluates to true. In this example, the value of `n` is not greater than zero, hence the expression returns false and the loop is skipped.

On the other hand, the do...while loop executes the statement once. This is because the initial iteration does not consider the boolean expression. However, for the subsequent iteration, the while checks the condition and takes the control out of the loop.

The Loop Control Statements

The break statement

The break statement is used to take the control out of a construct. Using break in a loop causes the program to exit the loop. Following is an example of the break statement.

Example

```javascript
var i=1
while(i<=10)
{
    if (i % 5 == 0)
    {
        console.log("The first multiple of 5 between 1 and 10 is : "+i)
        break     //exit the loop if the first multiple is found
    }
    i++
}
```

The above code prints the first multiple of 5 for the range of numbers within 1 to 10.
If a number is found to be divisible by 5, the if construct forces the control to exit the loop using the break statement. The following output is displayed on successful execution of the above code.

The first multiple of 5 between 1 and 10 is: 5

**The continue statement**

The continue statement skips the subsequent statements in the current iteration and takes the control back to the beginning of the loop. Unlike the break statement, the continue doesn’t exit the loop. It terminates the current iteration and starts the subsequent iteration. Following is an example of the continue statement.

```javascript
var num=0
var count=0;
for(num=0;num<=20;num++)
{
    if (num % 2==0)
    {
        continue
    }
    count++
}
console.log(" The count of odd values between 0 and 20 is: "+count)
```

The above example displays the number of even values between 0 and 20. The loop exits the current iteration if the number is even. This is achieved using the continue statement.

The following output is displayed on successful execution of the above code.

The count of odd values between 0 and 20 is: 10

**Using Labels to Control the Flow**

A label is simply an identifier followed by a colon (:) that is applied to a statement or a block of code. A label can be used with break and continue to control the flow more precisely.

Line breaks are not allowed between the `continue` or `break` statement and its label name. Also, there should not be any other statement in between a label name and an associated loop.
Example: Label with Break

```javascript
outerloop: // This is the label name
for (var i = 0; i < 5; i++)
{
    console.log("Outerloop: " + i);
    innerloop:
    for (var j = 0; j < 5; j++)
    {
        if (j > 3) break; // Quit the innermost loop
        if (i == 2) break innerloop; // Do the same thing
        if (i == 4) break outerloop; // Quit the outer loop
        console.log("Innerloop: " + j);
    }
}
```

The following output is displayed on successful execution of the above code.

```
Outerloop: 0
Innerloop: 0
Innerloop: 1
Innerloop: 2
Innerloop: 3
Outerloop: 1
Innerloop: 0
Innerloop: 1
Innerloop: 2
Innerloop: 3
Outerloop: 2
Outerloop: 3
Innerloop: 0
Innerloop: 1
Innerloop: 2
Innerloop: 3
Outerloop: 4
```
Example: Label with continue

```javascript
outerloop: // This is the label name
for (var i = 0; i < 3; i++)
{
    console.log("Outerloop: " + i);
    for (var j = 0; j < 5; j++)
    {
        if (j == 3){
            continue outerloop;
        }
        console.log("Innerloop: " + j);
    }
}
```

The following output is displayed on successful execution of the above code.

```
Outerloop: 0
Innerloop: 0
Innerloop: 1
Innerloop: 2
Outerloop: 1
Innerloop: 0
Innerloop: 1
Innerloop: 2
Outerloop: 2
Innerloop: 0
Innerloop: 1
Innerloop: 2
```
Functions are the building blocks of readable, maintainable, and reusable code. Functions are defined using the function keyword. Following is the syntax for defining a standard function.

```javascript
function function_name()
{
    // function body
}
```

To force execution of the function, it must be called. This is called as function invocation. Following is the syntax to invoke a function.

```javascript
function_name()
```

**Example: Simple function definition**

```javascript
//define a function
function test()
{
    console.log("function called")
}
//call the function
test()
```

The example defines a function test(). A pair of delimiters ( `{ } `) define the function body. It is also called as the function scope. A function must be invoked to force its execution.

The following output is displayed on successful execution of the above code.

```
function called
```

**Classification of Functions**

Functions may be classified as **Returning** and **Parameterized** functions.

**Returning functions**

Functions may also return the value along with control, back to the caller. Such functions are called as returning functions.

Following is the syntax for the returning function.
function function_name()
{
    //statements
    return value;
}

- A returning function must end with a return statement.
- A function can return at the most one value. In other words, there can be only one return statement per function.
- The return statement should be the last statement in the function.

The following code snippet is an example of a returning function:

```javascript
function retStr()
{
    return "hello world!!!"
}

var val=retStr()
console.log(val)
```

The above Example defines a function that returns the string “hello world!!!” to the caller. The following output is displayed on successful execution of the above code.

```
hello world!!!
```

**Parameterized functions**

Parameters are a mechanism to pass values to functions. Parameters form a part of the function’s signature. The parameter values are passed to the function during its invocation. Unless explicitly specified, the number of values passed to a function must match the number of parameters defined.

Following is the syntax defining a parameterized function.

```javascript
function func_name( param1,param2 ,...,paramN)
{
    .......
    .......
}
```
Example: Parametrized Function

The Example defines a function `add` that accepts two parameters `n1` and `n2` and prints their sum. The parameter values are passed to the function when it is invoked.

```javascript
function add( n1, n2) {
    var sum=n1 + n2
    console.log("The sum of the values entered "+sum)
}
add(12,13)
```

The following output is displayed on successful execution of the above code.

```
The sum of the values entered 25
```

Default function parameters

In ES6, a function allows the parameters to be initialized with default values, if no values are passed to it or it is undefined. The same is illustrated in the following code.

```javascript
function add(a, b = 1) {
    return a+b;
}
console.log(add(4))
```

The above function, sets the value of b to 1 by default. The function will always consider the parameter b to bear the value 1 unless a value has been explicitly passed. The following output is displayed on successful execution of the above code.

```
5
```

The parameter's default value will be overwritten if the function passes a value explicitly.

```javascript
function add(a, b = 1) {
    return a+b;
}
console.log(add(4,2))
```

The above code sets the value of the parameter b explicitly to 2, thereby overwriting its default value. The following output is displayed on successful execution of the above code.

```
6
```
Rest Parameters

Rest parameters are similar to variable arguments in Java. Rest parameters doesn’t restrict the number of values that you can pass to a function. However, the values passed must all be of the same type. In other words, rest parameters act as placeholders for multiple arguments of the same type.

To declare a rest parameter, the parameter name is prefixed with three periods, known as the spread operator. The following example illustrates the same.

```javascript
function fun1(...params) {
    console.log(params.length);
}

fun1();
fun1(5);
fun1(5, 6, 7);
```

The following output is displayed on successful execution of the above code.

```
0
1
3
```

**Note**: Rest parameters should be the last in a function’s parameter list.

Anonymous Function

Functions that are not bound to an identifier (function name) are called as anonymous functions. These functions are dynamically declared at runtime. Anonymous functions can accept inputs and return outputs, just as standard functions do. An anonymous function is usually not accessible after its initial creation.

Variables can be assigned an anonymous function. Such an expression is called a function expression.

Following is the syntax for anonymous function.

```javascript
var res=function( [arguments] ) { ... }
```

**Example: Anonymous Function**

```javascript
var f=function(){ return "hello"}
console.log(f())
```
The following output is displayed on successful execution of the above code.

```
hello
```

**Example: Anonymous Parameterized Function**

```javascript
var func = function(x,y){ return x*y }
function product()
{
  var result;
  result = func(10,20);
  console.log("The product : "+result)
}
product()
```

The following output is displayed on successful execution of the above code.

```
The product : 200
```

**The Function Constructor**

The 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.

Following is the syntax to create a function using `Function()` constructor along with the new operator.

```javascript
var variablename = new Function(Arg1, Arg2..., "Function Body");
```

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.

The `Function()` constructor is not passed any argument that specifies a name for the function it creates.

**Example: Function Constructor**

```javascript
var func = new Function("x", "y", "return x*y;");
function product()
{
  var result;
  result = func(10,20);
  console.log("The product : "+result)
```
In the above example, the `Function()` constructor is used to define an anonymous function. The function accepts two parameters and returns their product.

The following output is displayed on successful execution of the above code.

```
The product : 200
```

**Recursion and JavaScript Functions**

Recursion is a technique for iterating over an operation by having a function call itself repeatedly until it arrives at a result. Recursion is best applied when you need to call the same function repeatedly with different parameters from within a loop.

**Example: Recursion**

```javascript
function factorial(num)
{
    if(num<=0)
    {
        return 1;
    }
    else
    {
        return (num * factorial(num-1) )
    }
}
console.log(factorial(6))
```

In the above example the function calls itself. The following output is displayed on successful execution of the above code.

```
720
```

**Example: Anonymous Recursive Function**

```javascript
(function()
{
    var msg="Hello World"
    console.log(msg)
}
```
The function calls itself using a pair of parentheses (()). The following output is displayed on successful execution of the above code.

Hello World

**Lambda Functions**

Lambda refers to anonymous functions in programming. Lambda functions are a concise mechanism to represent anonymous functions. These functions are also called as **Arrow functions**.

**Lambda Function - Anatomy**

There are 3 parts to a Lambda function:

- **Parameters**: A function may optionally have parameters.
- The **fat arrow notation/lambda notation** (=>): It is also called as the goes to operator.
- **Statements**: Represents the function’s instruction set

**Tip**: By convention, the use of a single letter parameter is encouraged for a compact and precise function declaration.

**Lambda Expression**

It is an anonymous function expression that points to a single line of code. Following is the syntax for the same.

```
( [param1, param2,…param n ])=>statement;
```

**Example: Lambda Expression**

```
var foo=(x)=>10+x
console.log(foo(10))
```

The Example declares a lambda expression function. The function returns the sum of 10 and the argument passed.

The following output is displayed on successful execution of the above code.

```
20
```

**Lambda Statement**
It is an anonymous function declaration that points to a block of code. This syntax is used when the function body spans multiple lines. Following is the syntax of the same.

```
( [param1, para2,...param n] )=>
{
    //code block
}
```

**Example: Lambda Statement**

```
var msg=()=>
{
    console.log("function invoked")
}
msg()
```

The function's reference is returned and stored in the variable msg. The following output is displayed on successful execution of the above code.

```
function invoked
```

**Syntactic Variations**

Optional parentheses for a single parameter

```
var msg=x=>
{
    console.log(x)
}
msg(10)
```

Optional braces for a single statement. Empty parentheses for no parameter.

```
var disp=()=>console.log("Hello World")
disp();
```

**Function Expression and Function Declaration**

Function expression and function declaration are not synonymous. Unlike a function expression, a function declaration is bound by the function name.

The fundamental difference between the two is that, function declarations are parsed before their execution. On the other hand, function expressions are parsed only when the script engine encounters it during an execution.
When the JavaScript parser sees a function in the main code flow, it assumes function declaration. When a function comes as a part of a statement, it is a function expression.

**Function Hoisting**

Like variables, functions can also be hoisted. Unlike variables, function declarations when hoisted, hoists the function definition rather than just hoisting the function’s name.

The following code snippet, illustrates function hoisting in JavaScript.

```javascript
hoist_function();

function hoist_function() {
    console.log("foo");
}
```

The following output is displayed on successful execution of the above code.

```
foo
```

However, function expressions cannot be hoisted. The following code snippet illustrates the same.

```javascript
hoist_function(); // TypeError: hoist_function() is not a function

var hoist_function()= function() {
    console.log("bar");
};
```

**Immediately Invoked Function Expression**

Immediately Invoked Function Expressions (IIFEs) can be used to avoid variable hoisting from within blocks. It allows public access to methods while retaining privacy for variables defined within the function. This pattern is called as a self-executing anonymous function. The following two examples better explain this concept.

**Example 1: IIFE**

```javascript
var main = function() {
    var loop =function() {
        for(var x=0;x<5;x++)
        {
```
```javascript
console.log(x);
}
})();
console.log("x can not be accessed outside the block scope x value is :"+x);
main();
```

**Example 2:** IIFE

```javascript
var main = function()
{
    (function()
    {
        for(var x=0;x<5;x++)
        {
            console.log(x);
        }
    })();
    console.log("x can not be accessed outside the block scope x value is :"+x);
}
main();
```

Both the Examples will render the following output.

```
0
1
2
3
4
5
Uncaught ReferenceError: x is not defined
```

**Generator Functions**

When a normal function is invoked, the control rests with the function called until it returns. With generators in ES6, the caller function can now control the execution of a called function. A generator is like a regular function except that:
- The function can yield control back to the caller at any point.

- When you call a generator, it doesn't run right away. Instead, you get back an iterator. The function runs as you call the iterator's next method.

Generators are denoted by suffixing the function keyword with an asterisk; otherwise, their syntax is identical to regular functions.

The following example illustrates the same.

```javascript
"use strict"
function* rainbow() { // the asterisk marks this as a generator
  yield 'red';
  yield 'orange';
  yield 'yellow';
  yield 'green';
  yield 'blue';
  yield 'indigo';
  yield 'violet';
}
for(let color of rainbow()) {
  console.log(color);
}
```

Generators enable two-way communication between the caller and the called function. This is accomplished by using the `yield` keyword.

Consider the following example:

```javascript
function* ask() {
  const name = yield "What is your name?";
  const sport = yield "What is your favorite sport?"
  return `${name}'s favorite sport is ${sport}`;
}
const it = ask();
console.log(it.next());
console.log(it.next('Ethan'));
console.log(it.next('Cricket'));
```

Sequence of the generator function is as follows:

- Generator started in paused stated; iterator is returned.
The `it.next()` yields "What is your name". The generator is paused. This is done by the `yield` keyword.

The call `it.next("Ethan")` assigns the value Ethan to the variable `name` and yields "What is your favorite sport?" Again the generator is paused.

The call `it.next("Cricket")` assigns the value Cricket to the variable `sport` and executes the subsequent return statement.

Hence, the output of the above code will be:

```javascript
{ value: 'What is your name?', done: false }
{ value: 'What is your favorite sport?', done: false }
{ value: 'Ethan\'s favorite sport is Cricket', done: true }
```

**Note:** Generator functions cannot be represented using arrow functions.
JavaScript is meant to add interactivity to your pages. JavaScript does this using a mechanism using events. **Events** are a part of the Document Object Model (DOM) Level 3 and every HTML element contains a set of events that can trigger JavaScript Code.

An event is an action or occurrence recognized by the software. It can be triggered by a user or the system. Some common examples of events include a user clicking on a button, loading the web page, clicking on a hyperlink and so on. Following are some of the common HTML events.

### Event Handlers

On the occurrence of an event, the application executes a set of related tasks. The block of code that achieves this purpose is called the **eventhandler**. Every HTML element has a set of events associated with it. We can define how the events will be processed in JavaScript by using event handlers.

#### 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**

```html
<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>
```

The following output is displayed on successful execution of the above code.
onsubmitEvent Type

onsubmit is an event that occurs when you try to submit a form. You can put your form validation against this event type.

The following example shows how to use onsubmit. Here we are calling a validate() function before submitting a form data to the webserver. If validate() function returns true, the form will be submitted, otherwise it will not submit the data.

Example

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

function validation() {
    all validation goes here
    ........
    return either true or false
}

</script>
</head>
<body>
<form method="POST" action="t.cgi" onsubmit="return validate()">
    ........
    <input type="submit" value="Submit" />
</form>
</body>
</html>
```
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

```html
<html>
<head>
<script type="text/javascript">
function over() {
    document.write ("Mouse Over");
}
function out() {
    document.write ("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>
```

The following output is displayed on successful execution of the above code.

Bring your mouse inside the division to see the result:

**This is inside the division**

HTML 5 Standard Events

The standard HTML 5 events are listed in the following table for your reference. The script indicates a JavaScript function to be executed against that event.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline</td>
<td>script</td>
<td>Triggers when the document goes offline</td>
</tr>
<tr>
<td>Onabort</td>
<td>script</td>
<td></td>
</tr>
<tr>
<td>Event Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>onafterprint</td>
<td>script</td>
<td>Triggers after the document is printed</td>
</tr>
<tr>
<td>onbeforeonload</td>
<td>script</td>
<td>Triggers before the document loads</td>
</tr>
<tr>
<td>onbeforeprint</td>
<td>script</td>
<td>Triggers before the document is printed</td>
</tr>
<tr>
<td>onblur</td>
<td>script</td>
<td>Triggers when the window loses focus</td>
</tr>
<tr>
<td>oncanplay</td>
<td>script</td>
<td>Triggers when the media can start play, but might have to stop for buffering</td>
</tr>
<tr>
<td>oncanplaythrough</td>
<td>script</td>
<td>Triggers when the media can be played to the end, without stopping for buffering</td>
</tr>
<tr>
<td>onchange</td>
<td>script</td>
<td>Triggers when an element changes</td>
</tr>
<tr>
<td>onclick</td>
<td>script</td>
<td>Triggers on a mouse click</td>
</tr>
<tr>
<td>oncontextmenu</td>
<td>script</td>
<td>Triggers when a context menu is triggered</td>
</tr>
<tr>
<td>ondblclick</td>
<td>script</td>
<td>Triggers on a mouse double-click</td>
</tr>
<tr>
<td>ondrag</td>
<td>script</td>
<td>Triggers when an element is dragged</td>
</tr>
<tr>
<td>ondragend</td>
<td>script</td>
<td>Triggers at the end of a drag operation</td>
</tr>
<tr>
<td>ondragenter</td>
<td>script</td>
<td>Triggers when an element has been dragged to a valid drop target</td>
</tr>
<tr>
<td>Event Name</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ondragleave</td>
<td>script</td>
<td>Triggers when an element leaves a valid drop target</td>
</tr>
<tr>
<td>ondragover</td>
<td>script</td>
<td>Triggers when an element is being dragged over a valid drop target</td>
</tr>
<tr>
<td>ondragstart</td>
<td>script</td>
<td>Triggers at the start of a drag operation</td>
</tr>
<tr>
<td>ondrop</td>
<td>script</td>
<td>Triggers when the dragged element is being dropped</td>
</tr>
<tr>
<td>ondurationchange</td>
<td>script</td>
<td>Triggers when the length of the media is changed</td>
</tr>
<tr>
<td>onemptied</td>
<td>script</td>
<td>Triggers when a media resource element suddenly becomes empty</td>
</tr>
<tr>
<td>onended</td>
<td>script</td>
<td>Triggers when the media has reached the end</td>
</tr>
<tr>
<td>onerror</td>
<td>script</td>
<td>Triggers when an error occurs</td>
</tr>
<tr>
<td>onfocus</td>
<td>script</td>
<td>Triggers when the window gets focus</td>
</tr>
<tr>
<td>onformchange</td>
<td>script</td>
<td>Triggers when a form changes</td>
</tr>
<tr>
<td>onforminput</td>
<td>script</td>
<td>Triggers when a form gets user input</td>
</tr>
<tr>
<td>onhaschange</td>
<td>script</td>
<td>Triggers when the document has changed</td>
</tr>
<tr>
<td>oninput</td>
<td>script</td>
<td>Triggers when an element gets user input</td>
</tr>
<tr>
<td>oninvalid</td>
<td>script</td>
<td>Triggers when an element is invalid</td>
</tr>
<tr>
<td>onkeydown</td>
<td>script</td>
<td>Triggers when a key is pressed</td>
</tr>
<tr>
<td>onkeypress</td>
<td>script</td>
<td>Triggers when a key is pressed and released</td>
</tr>
<tr>
<td>Event Name</td>
<td>Event Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>onkeyup</td>
<td>script</td>
<td>Triggers when a key is released</td>
</tr>
<tr>
<td>onload</td>
<td>script</td>
<td>Triggers when the document loads</td>
</tr>
<tr>
<td>onloadeddata</td>
<td>script</td>
<td>Triggers when media data is loaded</td>
</tr>
<tr>
<td>onloadedmetadata</td>
<td>script</td>
<td>Triggers when the duration and other media data of a media element is loaded</td>
</tr>
<tr>
<td>onloadstart</td>
<td>script</td>
<td>Triggers when the browser starts to load the media data</td>
</tr>
<tr>
<td>onmessage</td>
<td>script</td>
<td>Triggers when the message is triggered</td>
</tr>
<tr>
<td>onmousedown</td>
<td>script</td>
<td>Triggers when a mouse button is pressed</td>
</tr>
<tr>
<td>onmousemove</td>
<td>script</td>
<td>Triggers when the mouse pointer moves</td>
</tr>
<tr>
<td>onmouseout</td>
<td>script</td>
<td>Triggers when the mouse pointer moves out of an element</td>
</tr>
<tr>
<td>onmouseover</td>
<td>script</td>
<td>Triggers when the mouse pointer moves over an element</td>
</tr>
<tr>
<td>Event Name</td>
<td>Event Category</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>onmouseup</td>
<td>script</td>
<td>Triggers when a mouse button is released</td>
</tr>
<tr>
<td>onmousewheel</td>
<td>script</td>
<td>Triggers when the mouse wheel is being rotated</td>
</tr>
<tr>
<td>onoffline</td>
<td>script</td>
<td>Triggers when the document goes offline</td>
</tr>
<tr>
<td>ononline</td>
<td>script</td>
<td>Triggers when the document comes online</td>
</tr>
<tr>
<td>onpagehide</td>
<td>script</td>
<td>Triggers when the window is hidden</td>
</tr>
<tr>
<td>onpageshow</td>
<td>script</td>
<td>Triggers when the window becomes visible</td>
</tr>
<tr>
<td>onpause</td>
<td>script</td>
<td>Triggers when the media data is paused</td>
</tr>
<tr>
<td>onplay</td>
<td>script</td>
<td>Triggers when the media data is going to start playing</td>
</tr>
<tr>
<td>onplaying</td>
<td>script</td>
<td>Triggers when the media data has start playing</td>
</tr>
<tr>
<td>onpopstate</td>
<td>script</td>
<td>Triggers when the window's history changes</td>
</tr>
<tr>
<td>onprogress</td>
<td>script</td>
<td>Triggers when the browser is fetching the media data</td>
</tr>
<tr>
<td>onratechange</td>
<td>script</td>
<td>Triggers when the media data's playing rate has changed</td>
</tr>
<tr>
<td>onreadystatechange</td>
<td>script</td>
<td>Triggers when the ready-state changes</td>
</tr>
<tr>
<td>Event</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>onredo</td>
<td>script</td>
<td>Triggers when the document performs a redo</td>
</tr>
<tr>
<td>onresize</td>
<td>script</td>
<td>Triggers when the window is resized</td>
</tr>
<tr>
<td>onscroll</td>
<td>script</td>
<td>Triggers when an element's scrollbar is being scrolled</td>
</tr>
<tr>
<td>onseeked</td>
<td>script</td>
<td>Triggers when a media element's seeking attribute is no longer true, and the seeking has ended</td>
</tr>
<tr>
<td>onseeking</td>
<td>script</td>
<td>Triggers when a media element's seeking attribute is true, and the seeking has begun</td>
</tr>
<tr>
<td>onselect</td>
<td>script</td>
<td>Triggers when an element is selected</td>
</tr>
<tr>
<td>onstalled</td>
<td>script</td>
<td>Triggers when there is an error in fetching media data</td>
</tr>
<tr>
<td>onstorage</td>
<td>script</td>
<td>Triggers when a document loads</td>
</tr>
<tr>
<td>onsubmit</td>
<td>script</td>
<td>Triggers when a form is submitted</td>
</tr>
<tr>
<td>onsuspend</td>
<td>script</td>
<td>Triggers when the browser has been fetching media data, but stopped before the entire media file was fetched</td>
</tr>
<tr>
<td>ontimeupdate</td>
<td>script</td>
<td>Triggers when the media changes its playing position</td>
</tr>
<tr>
<td>onundo</td>
<td>script</td>
<td>Triggers when a document performs an undo</td>
</tr>
<tr>
<td>Event</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td>onunload</td>
<td>script</td>
<td>Triggers when the user leaves the document</td>
</tr>
<tr>
<td>onvolumechange</td>
<td>script</td>
<td>Triggers when the media changes the volume, also when the volume is set to &quot;mute&quot;</td>
</tr>
<tr>
<td>onwaiting</td>
<td>script</td>
<td>Triggers when the media has stopped playing, but is expected to resume</td>
</tr>
</tbody>
</table>
Web Browsers and Servers use HTTP protocol to communicate. HTTP is stateless protocol, i.e., it doesn’t maintain the client’s data across multiple requests made by the client. This complete request-response cycle between the client and the server is defined as a session. Cookies are the default mechanism used by browsers to store data pertaining to a user’s session.

**How It Works?**

Your server sends some data to the visitor’s browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the browser sends the same cookie to the server for retrieval. Once retrieved, your server knows/remembers what was stored earlier.

Cookies are plain text data record of 5 variable-length fields.

- **Expires**: The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.

- **Domain**: The domain name of your site.

- **Path**: The path to the directory or web page that sets the cookie. This may be blank, if you want to retrieve the cookie from any directory or page.

- **Secure**: If this field contains the word "secure", then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.

- **Name=Value**: Cookies are set and retrieved in the form of key-value pairs.

Cookies were originally designed for CGI programming. The data contained in a cookie is automatically transmitted between the web browser and the web server, so CGI scripts on the server can read and write cookie values that are stored on the client side.

JavaScript can also manipulate cookies using the cookie property of the Document object. JavaScript can read, create, modify, and delete the cookies that apply to the current web page.

**Storing Cookies**

The simplest way to create a cookie is to assign a string value to the `document.cookie` object, which looks like this.

```javascript
document.cookie = "key1=value1;key2=value2;expires=date";
```
Here, the ‘expires’ attribute is optional. If you provide this attribute with a valid date or time, then the cookie will expire on the given date or time and thereafter, the cookies' value will not be accessible.

**Note:** Cookie values may not include semicolons, commas, or whitespace. For this reason, you may want to use the JavaScript `escape()` function to encode the value before storing it in the cookie. If you do this, you will also have to use the corresponding `unescape()` function when you read the cookie value.

**Example**

```html
<html>
<head>
<script type="text/javascript">
function WriteCookie()
{
    if( document.myform.customer.value == "" ){
        alert ("Enter some value!");
        return;
    }
    cookievalue= escape(document.myform.customer.value) + ";";
    document.cookie="name=" + cookievalue;
    document.write ("Setting Cookies : " + "name=" + cookievalue );
}
</script>
</head>
<body>
<form name="myform" action="">
Enter name: <input type="text" name="customer"/>
<input type="button" value="Set Cookie" onclick="WriteCookie();"/>
</form>
</body>
</html>
```
The following output is displayed on successful execution of the above code.

```
Enter name: [Input Field]   Set
```

Now your machine has a cookie called name. You can set multiple cookies using multiple key=value pairs separated by comma.

**Reading Cookies**

Reading a cookie is just as simple as writing one, because the value of the `document.cookie` object is the cookie. So you can use this string whenever you want to access the cookie. The `document.cookie` string will keep a list of name=value pairs separated by semicolons, where the name is the name of a cookie and the value is its string value.

You can use strings' `split()` function to break a string into key and values as shown in the following example.

**Example**

```html
<html>
<head>
<script type="text/javascript">
function ReadCookie()
{
    var allcookies = document.cookie;
    document.write ("All Cookies : " + allcookies );
}
// Get all the cookies pairs in an array
cookiearray = allcookies.split(';');
// Now take key value pair out of this array
for(var i=0; i<cookiearray.length; i++){
    name = cookiearray[i].split('=')[0];
    value = cookiearray[i].split('=')[1];
    document.write ("Key is : " + name + " and Value is : " + value);
}
}
</script>
</head>
<body>
</body>
</html>
```
Note: Here, length is a method of Array class which returns the length of an array.

There may be some other cookies already set on your machine. The above code will display all the cookies set on your machine.

The following output is displayed on successful execution of the above code.

**click the following button and see the result:**

Get Cookie

---

**Setting Cookies Expiry Date**

You can extend the life of a cookie beyond the current browser session by setting an expiry date and saving the expiry date within the cookie. This can be done by setting the 'expires' attribute to a date and time. The following example illustrates how to extend the expiry date of a cookie by 1 month.

**Example**

```html
<html>
<head>
<script type="text/javascript">
function WriteCookie()
{
    var now = new Date();
    now.setMonth( now.getMonth() + 1 );
    cookievalue = escape(document.myform.customer.value) + ";"
    document.cookie="name=" + cookievalue;
</script>
</head>
<body>
<form name="myform" action="">
<p> click the following button and see the result:</p>
<input type="button" value="Get Cookie" onclick="ReadCookie()"/>
</form>
</body>
</html>
```
The following output is displayed on successful execution of the above code.

Enter Cookie Name:  

**Deleting a Cookie**

Sometimes you will want to delete a cookie so that subsequent attempts to read the cookie return nothing. To do this, you just need to set the expiry date to a time in the past. The following example illustrates how to delete a cookie by setting its expiry date to one month behind the current date.

**Example**

```html
<html>
<head>
<script type="text/javascript">
function WriteCookie()
{
    var now = new Date();
    now.setMonth( now.getMonth() - 1 );
    cookievalue = escape(document.myform.customer.value) + ";";
    document.cookie = "expires=" + now.toUTCString() + ";";
    document.write ("Setting Cookies : " + "name=" + cookievalue);
}
</script>
</head>
<body>
<form name="formname" action="">
Enter name: <input type="text" name="customer"/>
<input type="button" value="Set Cookie" onclick="WriteCookie()"/>
</form>
</body>
</html>
```
The following output is displayed on successful execution of the above code.

Enter Cookie Name:  

Set Cookie
Redirect is a way to send both users and search engines to a different URL from the one they originally requested. Page redirection is a way to automatically redirect a web page to another web page. The redirected page is often on the same website, or it can be on a different website or a web server.

JavaScript Page Redirection

window.location and window.location.href

In JavaScript, you can use many methods to redirect a web page to another one. Almost all methods are related to window.location object, which is a property of the Window object. It can be used to get the current URL address (web address) and to redirect the browser to a new page. Both usages are same in terms of behavior. window.location returns an object. If .href is not set, window.location defaults to change the parameter .href.

Example

```html
<!DOCTYPE html>
<html>
<head>
<script>
function newLocation() {
    window.location="http://www.xyz.com";
}
</script>
</head>
<body>
<input type="button" value="Go to new location" onclick="newLocation()">
</body>
</html>
```

location.replace()

The other most frequently used method is the replace() method of window.location object, it will replace the current document with a new one. In replace() method, you can pass a new URL to replace() method and it will perform an HTTP redirect.
Following is the syntax for the same.

```javascript
window.location.replace("http://www.abc.com");
```

**location.assign()**
The `location.assign()` method loads a new document in the browser window.

Following is the syntax for the same.

```javascript
window.location.assign("http://www.abc.org");
```

**assign() vs. replace()**
The difference between `assign()` and `replace()` method is that the `location.replace()` method deletes the current URL from the document history, so it is unable to navigate back to the original document. You can’t use the browsers "Back" button in this case. If you want to avoid this situation, you should use `location.assign()` method, because it loads a new Document in the browser.

**location.reload()**
The `location.reload()` method reloads the current document in the browser window.

Following is the syntax for the same.

```javascript
window.location.reload("http://www.yahoo.com");
```

**window.navigate()**
The `window.navigate()` method is similar to assigning a new value to the `window.location.href` property. Because it is only available in MS Internet Explorer, so you should avoid using this in cross-browser development.

Following is the syntax for the same.

```javascript
window.navigate("http://www.abc.com");
```

**Redirection and Search Engine Optimization**
If you want to notify the search engines (SEO) about your URL forwarding, you should add the rel="canonical" meta tag to your website head part because search engines don’t analyze JavaScript to check the redirection.

Following is the syntax for the same.

```html
<link rel="canonical" href="http://abc.com/" />
```
JavaScript supports three important types of dialog boxes. These dialog boxes can be used to raise and alert, or to get confirmation on any input or to have a kind of input from the users. Here we will discuss each dialog box one by one.

**Alert Dialog Box**

An alert dialog box is mostly used to send a warning message to the users. For example, if one input field requires to enter some text but the user does not provide any input, then as a part of validation, you can use an alert box to send a warning message.

Nonetheless, an alert box can still be used for friendlier messages. Alert box provides only one button "OK" to select and proceed.

**Example**

```html
<html>
<head>
<script type="text/javascript">
    function Warn() {
        alert("This is a warning message!");
        document.write("This is a warning message!");
    }
</script>
</head>
<body>
<p>Click the following button to see the result: </p>
<form>
    <input type="button" value="Click Me" onclick="Warn();" />
</form>
</body>
</html>
```

The following output is displayed on successful execution of the above code.

**Click the following button to see the result:**

**Click Me**
Confirmation Dialog Box

A confirmation dialog box is mostly used to take the user's consent on any option. It displays a dialog box with two buttons: OK and Cancel.

If the user clicks on the OK button, the window method `confirm()` will return true. If the user clicks on the Cancel button, then `confirm()` returns false. You can use a confirmation dialog box as follows.

Example

```html
<html>
<head>
<script type="text/javascript">
function getConfirmation(){
    var retVal = confirm("Do you want to continue ?");
    if( retVal == true ){
        document.write ("User wants to continue!");
        return true;
    }
    else{
        Document.write ("User does not want to continue!");
        return false;
    }
}
</script>
</head>
<body>
<p>Click the following button to see the result: </p>
<form>
<input type="button" value="Click Me" onclick="getConfirmation();" />
</form>
</body>
</html>
```
The following output is displayed on successful execution of the above code.

```
Click the following button to see the result:

Click Me
```

### Prompt Dialog Box

The prompt dialog box is very useful when you want to pop-up a text box to get a user input. Thus, it enables you to interact with the user. The user needs to fill in the field and then click OK.

This dialog box is displayed using a method called `prompt()` which takes two parameters: (i) a label which you want to display in the text box and (ii) a default string to display in the text box.

This dialog box has two buttons: OK and Cancel. If the user clicks the OK button, the window method `prompt()` will return the entered value from the text box. If the user clicks the Cancel button, the window method `prompt()` returns null.

### Example

```
<html>
<head>
<script type="text/javascript">
function getValue(){
    var retVal = prompt("Enter your name : ", "your name here");
    document.write("You have entered : " + retVal);
}
</script>
</head>
<body>
<p>Click the following button to see the result: </p>
<form>
<input type="button" value="Click Me" onclick="getValue();" />
</form>
</body>
</html>
```
The following output is displayed on successful execution of the above code.

Click the following button to see the result:

Click Me
**void** is an important keyword in JavaScript which can be used as a unary operator that appears before its single operand, which may be of any type. This operator specifies an expression to be evaluated without returning a value. The operator evaluates a given expression and then returns undefined.

Following is the syntax for the same.

```
void expression
```

### Void and Immediately Invoked Function Expressions

When using an immediately-invoked function expression, void can be used to force the function keyword to be treated as an expression instead of a declaration. Consider the following example:

```
void function iife_void()
{
    var msg = function () {console.log("hello world");
    msg();

}()
```

The following output is displayed on successful execution of the above code.

```
hello world
```

### Void and JavaScript URIs

The **JavaScript: URI** is a commonly encountered syntax in a HTML page. The browser evaluates the URI and replaces the content of the page with the value returned. This is true unless the value returned is undefined. The most common use of this operator is in a client-side **JavaScript: URL**, where it allows you to evaluate an expression for its side-effects without the browser displaying the value of the evaluated expression.

Consider the following code snippet:

```
<a href="javascript:void(javascript:alert('hello world!!'))">Click here to do nothing</a>
<br/>
<br/>
<br/>
<a href="javascript:alert('hello');">Click here for an alert</a>
```
Save the above file as an HTML document and open it in the browser. The first hyperlink, when clicked evaluates the javascript:alert(“hello”) and is passed to the void() operator. However, since the void operator returns undefined, no result is displayed on the page.

On the other hand, the second hyperlink when clicked displays an alert dialog.
Many times you would like to place a button on your webpage to print the content of that web page via an actual printer. JavaScript helps you implement this functionality using the print function of the window object.

The JavaScript print function `window.print()` prints the current webpage when executed. You can call this function directly using the onclick event as shown in the following example.

**Example**

```html
<html>
<body>
<form>
<input type="button" value="Print" onclick="window.print()"/>
</form>
</body>
</html>
```

The following output is displayed on successful execution of the above code.
JavaScript supports extending data types. JavaScript objects are a great way to define custom data types.

An object is an instance which contains a set of key value pairs. Unlike primitive data types, objects can represent multiple or complex values and can change over their lifetime. The values can be scalar values or functions or even array of other objects.

The syntactic variations for defining an object is discussed further.

### Object Initializers

Like the primitive types, objects have a literal syntax: **curly braces** ({ and }). Following is the syntax for defining an object.

```javascript
var identifier= {Key1:value,
    Key2: function () {
        //functions
    },
    Key3: ["content1"," content2"]
}
```

The contents of an object are called **properties** (or members), and properties consist of a **name** (or key) and **value**. Property names must be strings or symbols, and values can be any type (including other objects).

Like all JavaScript variables, both the object name (which could be a normal variable) and the property name are case sensitive. You access the properties of an object with a simple dot-notation.

Following is the syntax for accessing Object Properties.

```javascript
objectName.propertyName
```

**Example: Object Initializers**

```javascript
var person={
    firstname:"Tom",
    lastname:"Hanks",
    func:function(){return "Hello!!"},
};
//access the object values
console.log(person.firstname)
```
The above Example, defines an object person. The object has three properties. The third property refers to a function.

The following output is displayed on successful execution of the above code.

```
Tom
Hanks
Hello!!
```

In ES6, assigning a property value that matches a property name, you can omit the property value.

**Example**

```
var foo = 'bar'
var baz = { foo }
console.log(baz.foo)
```

The above code snippet defines an object `baz`. The object has a property `foo`. The property value is omitted here as ES6 implicitly assigns the value of the variable `foo` to the object’s key `foo`.

Following is the ES5 equivalent of the above code.

```
var foo = 'bar'
var baz = { foo:foo }
console.log(baz.foo)
```

The following output is displayed on successful execution of the above code.

```
bar
```

With this shorthand syntax, the JS engine looks in the containing scope for a variable with the same name. If it is found, that variable’s value is assigned to the property. If it is not found, a Reference Error is thrown.

**The Object() Constructor**

JavaScript provides a special constructor function called `Object()` to build the object. The new operator is used to create an instance of an object. To create an object, the new operator is followed by the constructor method.
Following is the syntax for defining an object.

```javascript
var obj_name = new Object();
obj_name.property = value;

OR

obj_name["key"] = value
```

Following is the syntax for accessing a property.

```javascript
Object_name.property_key

OR

Object_name["property_key"]
```

**Example**

```javascript
var myCar = new Object();
myCar.make = "Ford";     //define an object
myCar.model = "Mustang";
myCar.year = 1987;

console.log(myCar["make"]) //access the object property
console.log(myCar["model"])  
console.log(myCar["year"]) 
```

The following output is displayed on successful execution of the above code.

```
Ford
Mustang
1987
```

Unassigned properties of an object are undefined.

**Example**

```javascript
var myCar = new Object();
myCar.make = "Ford";
console.log(myCar["model"]) 
```

The following output is displayed on successful execution of the above code.

```javascript
undefined
```
**Note:** An object property name can be any valid JavaScript string, or anything that can be converted to a string, including the empty string. However, any property name that is not a valid JavaScript identifier (for example, a property name that has a space or a hyphen, or that starts with a number) can only be accessed using the square bracket notation.

Properties can also be accessed by using a string value that is stored in a variable. In other words, the object’s property key can be a dynamic value. For example: a variable. The said concept is illustrated in the following example.

**Example**

```
var myCar = new Object();
var propertyName = "make";
myCar[propertyName] = "Ford";
console.log(myCar.make)
```

The following output is displayed on successful execution of the above code.

```
Ford
```

**Constructor Function**

An object can be created using the following two steps:

**Step 1:** Define the object type by writing a constructor function.

Following is the syntax for the same.

```
function function_name()
{
    this.property_name=value
}
```

The ‘this’ keyword refers to the current object in use and defines the object’s property.

**Step 2:** Create an instance of the object with the new syntax.

```
var Object_name= new function_name()
//Access the property value

Object_name.property_name
```

The new keyword invokes the function constructor and initializes the function’s property keys.
Example: Using a Function Constructor

```javascript
function Car()
{
    this.make="Ford"
    this.model="F123"
}

var obj=new Car()
console.log(obj.make)
console.log(obj.model)
```

The above example uses a function constructor to define an object.
The following output is displayed on successful execution of the above code.

```
Ford
F123
```

A new property can always be added to a previously defined object. For example, consider the following code snippet:

```javascript
function Car()
{
    this.make="Ford"
}
var obj=new Car()
obj.model="F123"
console.log(obj.make)
console.log(obj.model)
```

The following output is displayed on successful execution of the above code.

```
Ford
F123
```
The Object.create Method

Objects can also be created using the `Object.create()` method. It allows you to create the prototype for the object you want, without having to define a constructor function.

Example

```javascript
var roles = {
  type: "Admin", // Default value of properties
  displayType: function() { // Method which will display type of role
    console.log(this.type);
  }
}

// Create new role type called super_role
var super_role = Object.create(roles);
super_role.displayType(); // Output:Admin

// Create new role type called Guest
var guest_role = Object.create(roles);
guest_role.type = "Guest";
guest_role.displayType(); // Output:Guest
```

The above example defines an object `roles` and sets the default values for the properties. Two new instances are created that override the default properties value for the object. The following output is displayed on successful execution of the above code.

```
Admin
Guest
```

The Object.assign() Function

The `Object.assign()` method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

Following is the syntax for the same.

```
Object.assign(target, ...sources)
```
Example: Cloning an Object

"use strict"
var det = { name:"Tom", ID:"E1001" };
var copy = Object.assign({}, det);
close.log(copy);
for (let val in copy)
{
    console.log(copy[val])
}

The following output is displayed on successful execution of the above code.

Tom
E1001

Example: Merging Objects

var o1 = { a: 10 };
var o2 = { b: 20 };
var o3 = { c: 30 };
var obj = Object.assign(o1, o2, o3);
close.log(obj);
close.log(o1);

The following output is displayed on successful execution of the above code.

{ a: 10, b: 20, c: 30 }
{ a: 10, b: 20, c: 30 }

Note: Unlike copying objects, when objects are merged, the larger object doesn't maintain a new copy of the properties. Rather it holds the reference to the properties contained in the original objects. The following example explains this concept.

var o1 = { a: 10 };
var obj = Object.assign(o1);
obj.a++
console.log("Value of 'a' in the Merged object after increment ")
close.log(obj.a);
close.log("value of 'a' in the Original Object after increment ")
close.log(o1.a);
The following output is displayed on successful execution of the above code.

<table>
<thead>
<tr>
<th>Value of 'a' in the Merged object after increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
</tr>
<tr>
<td>value of 'a' in the Original Object after increment</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

### Deleting Properties

You can remove a property by using the delete operator. The following code shows how to remove a property.

**Example**

```javascript
// Creates a new object, myobj, with two properties, a and b.
var myobj = new Object;
myobj.a = 5;
myobj.b = 12;
// Removes the ‘a’ property
delete myobj.a;
console.log("a" in myobj) // yields "false"
```

The following output is displayed on successful execution of the above code.

| false |

The code snippet deletes the property from the object. The example prints false as the in operator doesn't find the property in the object.

### Comparing Objects

In JavaScript, objects are a reference type. Two distinct objects are never equal, even if they have the same properties. This is because, they point to a completely different memory address. Only those objects that share a common reference yields true on comparison.

**Example 1: Different Object References**

```javascript
var val1 = {name: "Tom"};
var val2 = {name: "Tom"};
console.log(val1 == val2)   // return false
console.log(val1 === val2) // return false
```
In the above example, `val1` and `val2` are two distinct objects that refer to two different memory addresses. Hence on comparison for equality, the operator will return false.

### Example 2: Single Object Reference

```javascript
var val1 = {name: "Tom"};
var val2 = val1

console.log(val1 == val2) // return true
console.log(val1 === val2) // return true
```

In the above example, the contents in `val1` are assigned to `val2`, i.e. the reference of the properties in `val1` are shared with `val2`. Since, the objects now share the reference to the property, the equality operator will return true for two distinct objects that refer to two different memory addresses. Hence on comparison for equality, the operator will return false.

### Object De-structuring

The term **destructuring** refers to breaking up the structure of an entity. The destructuring assignment syntax in JavaScript makes it possible to extract data from arrays or objects into distinct variables. The same is illustrated in the following example.

### Example

```javascript
var emp = { name: 'John', Id: 3 }
var {name, Id} = emp

console.log(name)
console.log(Id)
```

The following output is displayed on successful execution of the above code.

```
John
3
```

**Note:** To enable destructuring, execute the file in node as `node – harmony_destructuring file_name`. 
The Number object represents numerical data, either integers or floating-point numbers. In general, you do not need to worry about Number objects because the browser automatically converts number literals to instances of the number class.

Following is the syntax for creating a number object.

```javascript
var val = new Number(number);
```

In the place of `number`, if you provide any non-number argument, then the argument cannot be converted into a `number`, it returns NaN (Not-a-Number).

### Number Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number.EPSILON</td>
<td>The smallest interval between two representable numbers</td>
</tr>
<tr>
<td>Number.MAX_SAFE_INTEGER</td>
<td>The maximum safe integer in JavaScript (2^{53} - 1)</td>
</tr>
<tr>
<td>Number.MAX_VALUE</td>
<td>The largest positive representable number</td>
</tr>
<tr>
<td>Number.MIN_SAFE_INTEGER</td>
<td>The minimum safe integer in JavaScript ((-2^{53} - 1))</td>
</tr>
<tr>
<td>Number.MIN_VALUE</td>
<td>The smallest positive representable number - that is, the positive number closest to zero (without actually being zero)</td>
</tr>
<tr>
<td>Number.NaN</td>
<td>Special &quot;not a number&quot; value</td>
</tr>
<tr>
<td>Number.NEGATIVE_INFINITY</td>
<td>Special value representing negative infinity; returned on overflow</td>
</tr>
<tr>
<td>Number.POSITIVE_INFINITY</td>
<td>Special value representing infinity; returned on overflow</td>
</tr>
<tr>
<td>Number.prototype</td>
<td>Allows the addition of properties to a Number object</td>
</tr>
</tbody>
</table>
**EPSILON**

This property represents the smallest interval between two representable numbers.

**Example**

```javascript
var interval = Number.EPSILON
console.log(interval)
```

**Output**

`2.220446049250313e-16`

**MAX_SAFE_INTEGER**

This property represents the maximum safe integer in JavaScript i.e. \(2^{53} - 1\)

**Example**

```javascript
var interval = Number.MAX_SAFE_INTEGER
console.log(interval)
```

**Output**

`9007199254740991`

**MAX_VALUE**

The `Number.MAX_VALUE` property belongs to the static `Number` object. It represents constants for the largest possible positive numbers that JavaScript can work with.

The actual value of this constant is \(1.7976931348623157 \times 10^{308}\)

**Syntax**

```javascript
var val = Number.MAX_VALUE;
```

**Example**

```javascript
var val = Number.MAX_VALUE;
console.log("Value of Number.MAX_VALUE : " + val);
```

**Output**

`Value of Number.MAX_VALUE : 1.7976931348623157e+308`
**MIN_SAFE_INTEGER**

The `Number.MIN_SAFE_INTEGER` constant represents the minimum safe integer in JavaScript \((-2^{53} - 1))\). The `MIN_SAFE_INTEGER` constant has a value of `-9007199254740991`.

**Syntax**

```javascript
var val = Number.MIN_SAFE_INTEGER;
```

**Example**

```javascript
var val = Number.MIN_SAFE_INTEGER;
console.log("Value of Number.MIN_SAFE_INTEGER: " + val);
```

**Output**

```
Value of Number.MIN_SAFE_INTEGER: -9007199254740991
```

**MIN_VALUE**

The `Number.MIN_VALUE` property belongs to the static `Number` object. It represents constants for the smallest possible positive numbers that JavaScript can work with. This constant has actual value \(5 \times 10^{-324}\)

**Syntax**

```javascript
var val = Number.MIN_VALUE;
```

**Example**

```javascript
var val = Number.MAX_VALUE;
console.log("Value of Number.MIN_VALUE: " + val);
```

**Output**

```
Value of Number.MIN_VALUE: 1.7976931348623157e+308
```
Nan

Unquoted literal constant NaN is a special value representing Not-a-Number. Since NaN always compares unequal to any number, including NaN, it is usually used to indicate an error condition for a function that should return a valid number.

Syntax

```javascript
var val = Number.NaN;
```

Example

```javascript
var dayOfMonth = 50;
if (dayOfMonth < 1 || dayOfMonth > 31)
{
    dayOfMonth = Number.NaN
    console.log("Day of Month must be between 1 and 31.");
}
else
{
    console.log("day of month " + dayOfMonth)
}
```

The following output is displayed on successful execution of the above code.

```
Day of Month must be between 1 and 31.
```

NEGATIVE_INFINITY

This is a special numeric value representing a value less than Number.MIN_VALUE. This value is represented as "-Infinity". It resembles an infinity in its mathematical behavior. For example, anything multiplied by NEGATIVE_INFINITY is NEGATIVE_INFINITY, and anything divided by NEGATIVE_INFINITY is zero. Because NEGATIVE_INFINITY is a constant, it is a read-only property of Number.

Syntax

```javascript
var val = Number.NEGATIVE_INFINITY;
```

Example

```javascript
var val = Number.NEGATIVE_INFINITY;
console.log("Value of Number.NEGATIVE_INFINITY : " + val);
```
Output

Value of Number.NEGATIVE_INFINITY: -Infinity

**POSITIVE_INFINITY**

This is a special numeric value representing any value greater than Number.MAX_VALUE. This value is represented as "Infinity". It resembles an infinity in its mathematical behavior. For example, anything multiplied by POSITIVE_INFINITY is POSITIVE_INFINITY, and anything divided by POSITIVE_INFINITY is zero. As POSITIVE_INFINITY is a constant, it is a read-only property of Number.

**Syntax**

```javascript
var val = Number.POSITIVE_INFINITY;
```

**Example**

```javascript
var val = Number.POSITIVE_INFINITY;
console.log("Value of Number.POSITIVE_INFINITY : " + val);
```

**Output:**

Value of Number.POSITIVE_INFINITY : Infinity

**Number Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number.isNaN()</td>
<td>Determines whether the passed value is NaN</td>
</tr>
<tr>
<td>Number.isFinite()</td>
<td>Determines whether the passed value is a finite number</td>
</tr>
<tr>
<td>Number.isInteger()</td>
<td>Determines whether the passed value is an integer</td>
</tr>
<tr>
<td>Number.isSafeInteger()</td>
<td>Determines whether the passed value is a safe integer (number between -(253 - 1) and 253- 1)</td>
</tr>
<tr>
<td>Number.parseFloat()</td>
<td>The value is the same as parseFloat() of the global object</td>
</tr>
<tr>
<td>Number.parseInt()</td>
<td>The value is the same as parseInt() of the global object</td>
</tr>
</tbody>
</table>
**Number.isNaN()**

The `Number.isNaN()` method determines whether the passed value is NaN.

**Syntax**

```javascript
var res=Number.isNaN(value);
```

**Parameter Details**

Value to determine if it is a NaN.

**Return Value**

Returns a Boolean value `true` if the value is a not a number.

**Example**

```javascript
var res=Number.isNaN(10);
console.log(res);
```

---

**Number.isFinite**

The `Number.isFinite()` method determines whether the passed value is a finite number.

**Syntax**

```javascript
var res=Number.isFinite(value);
```

**Parameter Details**

Value to be tested for finiteness.

**Return Value**

Returns a Boolean value, true or false.

**Example**

```javascript
var res=Number.isFinite(10);
console.log(res);
```

**Output**

```
true
```
**Number.isInteger()**

The `Number.isInteger()` method determines whether the passed value is an integer.

**Syntax**

```javascript
var res=Number.isInteger(value);
```

**Parameter Details**

Value to be tested for being an integer.

**Return Value**

Returns a Boolean value, true or false.

**Example**

```javascript
console.log(Number.isInteger(0));          // true
console.log(Number.isInteger(1));          // true
console.log(Number.isInteger(-100000));    // true
console.log(Number.isInteger(0.1));        // false
console.log(Number.isInteger(Infinity));   // false
console.log(Number.isInteger("10"));      // false
console.log(Number.isInteger(true));       // false
console.log(Number.isInteger(false));      // false
```

**Output**

true

---

**Number.isSafeInteger()**

The `Number.isSafeInteger()` method determines whether the passed value is a safe integer.

**Syntax**

```javascript
var res=Number.isSafeInteger(value);
```

**Parameter Details**

Value to be tested for being a safe integer.
Return Value
Returns a Boolean value, true or false.

Example

```javascript
var res=Number.isSafeInteger(10);
console.log(res);
```

Output
```
true
```

Number.parseInt()
This method parses a string argument and returns an integer of the specified radix.

Syntax
```
Number.parseInt(string,[ radix ])
```

Parameters
- `string`: value to parse
- `radix`: integer between 2 and 36 that represents the base.

Return value
An integer representation of the string.

Example

```javascript
console.log(Number.parseInt("10"));
console.log(Number.parseInt("10.23"));
```

Output
```
10
10
```

Number.parseFloat()
This method parses a string argument and returns a float representation of the passed string.
**Syntax**

```javascript
Number.parseFloat(string)
```

**Parameters**

- `string`: value to parse

**Return value**

A float representation of the string.

**Example**

```javascript
console.log(Number.parseFloat("10");
console.log(Number.parseFloat("10.23");
```

**Output**

```
10
10.23
```

**Number Instances Methods**

The Number object contains only the default methods that are a part of every object's definition.

<table>
<thead>
<tr>
<th>Instance Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>toExponential()</code></td>
<td>Returns a string representing the number in exponential notation</td>
</tr>
<tr>
<td><code>toFixed()</code></td>
<td>Returns a string representing the number in fixed-point notation</td>
</tr>
<tr>
<td><code>toLocaleString()</code></td>
<td>Returns a string with a language sensitive representation of this number</td>
</tr>
<tr>
<td><code>toPrecision()</code></td>
<td>Returns a string representing the number to a specified precision in fixed-point or exponential notation</td>
</tr>
<tr>
<td><code>toString()</code></td>
<td>Returns a string representing the specified object in the specified radix (base)</td>
</tr>
<tr>
<td><code>valueOf()</code></td>
<td>Returns the primitive value of the specified object</td>
</tr>
</tbody>
</table>

**`toExponential()`**

This method returns a string representing the number object in exponential notation.

**Syntax**

```javascript
number.toExponential( [fractionDigits] )
```
**Parameter Details**

- **fractionDigits** – An integer specifying the number of digits after the decimal point. Defaults to as many digits as necessary to specify the number.

**Return Value**

A string representing a Number object in exponential notation with one digit before the decimal point, rounded to fractionDigits digits after the decimal point. If the fractionDigits argument is omitted, the number of digits after the decimal point defaults to the number of digits necessary to represent the value uniquely.

**Example**

```javascript
//toExponential()
var num1=1225.30
var val= num1.toExponential();
console.log(val)
```

**Output**

```
1.2253e+
```

**toFixed()**

This method formats a number with a specific number of digits to the right of the decimal.

**Syntax**

```
number.toFixed( [digits] )
```

**Parameter Details**

- **digits** – The number of digits to appear after the decimal point.

**Return Value**

A string representation of number that does not use exponential notation and has the exact number of digits after the decimal place.

**Example**

```javascript
var num3=177.234
console.log("num3.toFixed() is "+num3.toFixed())
console.log("num3.toFixed(2) is "+num3.toFixed(2))
console.log("num3.toFixed(6) is "+num3.toFixed(6))
```
### Output

<table>
<thead>
<tr>
<th>num3.toFixed()</th>
<th>num3.toFixed(2)</th>
<th>num3.toFixed(6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>is 177</td>
<td>is 177.23</td>
<td>is 177.234000</td>
</tr>
</tbody>
</table>

### toLocaleString()

This method converts a number object into a human readable string representing the number using the locale of the environment.

#### Syntax

```
number.toLocaleString()
```

#### Return Value

Returns a human readable string representing the number using the locale of the environment.

#### Example

```javascript
var num = new Number(177.1234);
console.log( num.toLocaleString());
```

#### Output

```
177.1234
```

### toPrecision()

This method returns a string representing the number object to the specified precision.

#### Syntax

```
number.toPrecision( [ precision ] )
```

#### Parameter Details

- **precision** – An integer specifying the number of significant digits.

#### Return Value

Returns a string representing a Number object in fixed-point or exponential notation rounded to the precision specified significant digits.
Example

```javascript
var num = new Number(7.123456);
console.log(num.toPrecision());
console.log(num.toPrecision(1));
console.log(num.toPrecision(2));
```

Output

```
7.123456
7
7.1
```

toString()

This method returns a string representing the specified object. The toString() method parses its first argument, and attempts to return a string representation in the specified radix (base).

Syntax

```
number.toString( [radix] )
```

Parameter Details

- `radix` – An integer between 2 and 36 specifying the base to use for representing numeric values.

Return Value

Returns a string representing the specified Number object.

Example

```javascript
var num = new Number(10);
console.log(num.toString());
console.log(num.toString(2));
console.log(num.toString(8));
```
valueOf()

This method returns the primitive value of the specified number object.

Syntax

```javascript
number.valueOf()
```

Return Value

Returns the primitive value of the specified Number object.

Example

```javascript
var num = new Number(10);
console.log(num.valueOf());
```

Output

<table>
<thead>
<tr>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>

Binary and Octal Literals

Before ES6, your best bet when it comes to binary or octal representation of integers was to just pass them to parseInt() with the radix. In ES6, you could use the `0b` and `0o` prefix to represent binary and octal integer literals respectively. Similarly, to represent a hexadecimal value, use the `0x` prefix.

The prefix can be written in upper or lower case. However, it is suggested to stick to the lowercase version.

Example: Binary Representation

```javascript
console.log(0b001)
console.log(0b010)
console.log(0b011)
console.log(0b100)
```

The following output is displayed on successful execution of the above code.
Example: Octal Representation

```
console.log(0o010)
console.log(0o100)
```

The following output is displayed on successful execution of the above code.

```
8
64
```

Example: Hexadecimal Representation

```
console.log(0o010)
console.log(0o100)
```

The following output is displayed on successful execution of the above code.

```
255
384
```
The Boolean object represents two values, either "true" or "false". If the value parameter is omitted or is 0, -0, null, false, NaN, undefined, or the empty string (""), the object has an initial value of false.

Use the following syntax to create a **boolean** object.

```
var val = new Boolean(value);
```

### Boolean Properties

Following is a list of the properties of Boolean object.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor</td>
<td>Returns a reference to the Boolean function that created the object.</td>
</tr>
<tr>
<td>prototype</td>
<td>The prototype property allows you to add properties and methods to an object.</td>
</tr>
</tbody>
</table>

In the following sections, we will look at a few examples to illustrate the properties of Boolean object.

**constructor ()**

Javascript Boolean **constructor()** method returns a reference to the Boolean function that created the instance's prototype.

Use the following syntax to create a Boolean constructor() method. It returns the function that created this object's instance.

```
boolean.constructor()
```

### Example

```
<html>
<head>
<title>JavaScript constructor() Method</title>
</head>
<body>
<script type="text/javascript">
    var bool = new Boolean( );
    document.write("bool.constructor() is : " + bool.constructor);
</script>
</body>
```
The following output is displayed on successful execution of the above code.

`bool.constructor() is : function Boolean() { [native code] }`

**Prototype**

The **prototype** property allows you to add properties and methods to any object (Number, Boolean, String and Date, etc.).

**Note:** Prototype is a global property which is available with almost all the objects.

Use the following syntax to create a Boolean prototype.

`object.prototype.name = value`

**Example**

The following example shows how to use the prototype property to add a property to an object.

```html
<html>
<head>
<title>User-defined objects</title>
<script type="text/javascript">
    function book(title, author){
        this.title = title;
        this.author = author;
    }
</script>
</head>
<body>
<script type="text/javascript">
    var myBook = new book("Perl", "Tom");
    book.prototype.price = null;
    myBook.price = 100;
    document.write("Book title is : " + myBook.title + "<br>");
    document.write("Book author is : " + myBook.author + "<br>");
    document.write("Book price is : " + myBook.price + "<br>");
</script>
</body>
</html>
```
ES6

The following output is displayed on successful execution of the above code.

<table>
<thead>
<tr>
<th>Book title is : Perl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book author is : Tom</td>
</tr>
<tr>
<td>Book price is : 100</td>
</tr>
</tbody>
</table>

Boolean Methods

Following is a list of the methods of Boolean object and their description.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toSource()</td>
<td>Returns a string containing the source of the Boolean object; you can use this string to create an equivalent object.</td>
</tr>
<tr>
<td>toString()</td>
<td>Returns a string of either &quot;true&quot; or &quot;false&quot; depending upon the value of the object.</td>
</tr>
<tr>
<td>valueOf()</td>
<td>Returns the primitive value of the Boolean object.</td>
</tr>
</tbody>
</table>

In the following sections, we will take a look at a few examples to demonstrate the usage of the Boolean methods.

**toSource ()**

Javascript boolean **toSource()** method returns a string representing the source code of the object.

**Note:** This method is not compatible with all the browsers.

Following is the syntax for the same.

```javascript
boolean.toSource()
```

**Example**

```html
<html>
<head>
<title>JavaScript toSource() Method</title>
</head>
<body>
<script type="text/javascript">
function book(title, publisher, price)
{
    this.title = title;
    this.publisher = publisher;
</script>
</body>
</html>
```
this.price = price;
}
var newBook = new book("Perl","Leo Inc",200);
document.write("newBook.toSource() is : "+ newBook.toSource());
</script>
</body>
</html>

The following output is displayed on successful execution of the above code.

({title:"Perl", publisher:"Leo Inc", price:200})

toString ()

This method returns a string of either "true" or "false" depending upon the value of the object.

Following is the syntax for the same.

boolean.toString()

Example

<html>
<head>
<title>JavaScript toString() Method</title>
</head>
<body>
<script type="text/javascript">
var flag = new Boolean(false);
document.write( "flag.toString is : " + flag.toString() );
</script>
</body>
</html>

The following output is displayed on successful execution of the above code.

flag.toString is : false
**valueOf ()**

Javascript **valueOf()** method returns the primitive value of the specified **Boolean** object.

Following is the syntax for the same.

```javascript
boolean.valueOf()
```

**Example**

```html
<html>
<head>
<title>JavaScript toString() Method</title>
</head>
<body>
<script type="text/javascript">
var flag = new Boolean(false);
document.write( "flag.valueOf is : " + flag.valueOf() );
</script>
</body>
</html>
```

The following output is displayed on successful execution of the above code.

```
flag.valueOf is : false
```
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.

Use the following syntax to create a String object.

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

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

### String Properties

Following is a list of the properties of String object and its description.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>constructor</code></td>
<td>Returns a reference to the String function that created the object</td>
</tr>
<tr>
<td><code>length</code></td>
<td>Returns the length of the string</td>
</tr>
<tr>
<td><code>prototype</code></td>
<td>The prototype property allows you to add properties and methods to an object</td>
</tr>
</tbody>
</table>

### Constructor

A constructor returns a reference to the string function that created the instance's prototype.

#### Syntax

```javascript
string.constructor
```

#### Return Value

Returns the function that created this object’s instance.

#### Example: String constructor property

```javascript
var str = new String("This is string");
console.log("str.constructor is:");
str.constructor
```

#### Output

```
str.constructor is: String
```

114
str.constructor is:function String() { [native code] }

Length

This property returns the number of characters in a string.

Syntax

string.length

Example: String length property

```javascript
var uname = new String("Hello World")
console.log(uname)
console.log("Length "+uname.length)  // returns the total number of characters
    // including whitespace
```

Output

```
Hello World
Length 11
```

Prototype

The prototype property allows you to add properties and methods to any object (Number, Boolean, String, Date, etc.).

Note: Prototype is a global property which is available with almost all the objects.

Syntax

string.prototype

Example: Object prototype

```javascript
function employee(id, name) {
    this.id = id;
    this.name = name;
}
var emp = new employee(123, "Smith");
employee.prototype.email = "smith@abc.com";
console.log("Employee 's Id: " + emp.id);
console.log("Employee's name: " + emp.name);
```
```javascript
console.log("Employee's Email ID: " + emp.email);
```

**Output**

Employee’s Id: 123  
Employee’s name: Smith  
Employee’s Email ID: smith@abc.com

**String Methods**

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>charAt()</code></td>
<td>Returns the character at the specified index</td>
</tr>
<tr>
<td><code>charCodeAt()</code></td>
<td>Returns a number indicating the Unicode value of the character at the given index</td>
</tr>
<tr>
<td><code>concat()</code></td>
<td>Combines the text of two strings and returns a new string</td>
</tr>
<tr>
<td><code>indexOf()</code></td>
<td>Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found</td>
</tr>
<tr>
<td><code>lastIndexOf()</code></td>
<td>Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found</td>
</tr>
<tr>
<td><code>localeCompare()</code></td>
<td>Returns a number indicating whether a reference string comes before or after or is the same as the given string in a sorted order</td>
</tr>
<tr>
<td><code>match()</code></td>
<td>Used to match a regular expression against a string</td>
</tr>
<tr>
<td><code>replace()</code></td>
<td>Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring</td>
</tr>
<tr>
<td><code>search()</code></td>
<td>Executes the search for a match between a regular expression and a specified string</td>
</tr>
<tr>
<td><code>slice()</code></td>
<td>Extracts a section of a string and returns a new string</td>
</tr>
<tr>
<td><code>split()</code></td>
<td>Splits a String object into an array of strings by separating the string into substrings</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>substr()</td>
<td>Returns the characters in a string beginning at the specified location through the specified number of characters</td>
</tr>
<tr>
<td>substring()</td>
<td>Returns the characters in a string between two indexes into the string</td>
</tr>
<tr>
<td>toLocaleLowerCase()</td>
<td>The characters within a string are converted to lower case while respecting the current locale</td>
</tr>
<tr>
<td>toLocaleUpperCase()</td>
<td>The characters within a string are converted to uppercase while respecting the current locale</td>
</tr>
<tr>
<td>toLowerCase()</td>
<td>Returns the calling string value converted to lowercase</td>
</tr>
<tr>
<td>toString()</td>
<td>Returns a string representing the specified object</td>
</tr>
<tr>
<td>toUpperCase()</td>
<td>Returns the calling string value converted to uppercase</td>
</tr>
<tr>
<td>valueOf()</td>
<td>Returns the primitive value of the specified object</td>
</tr>
</tbody>
</table>

### 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**

```javascript
string.charAt(index);
```

**Argument Details**

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

**Return Value**

Returns the character from the specified index.

**Example**
var str = new String("This is string");
console.log("str.charAt(0) is:" + str.charAt(0));
console.log("str.charAt(1) is:" + str.charAt(1));
console.log("str.charAt(2) is:" + str.charAt(2));
console.log("str.charAt(3) is:" + str.charAt(3));
console.log("str.charAt(4) is:" + str.charAt(4));
console.log("str.charAt(5) is:" + str.charAt(5));

Output
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

charCodeAt()
This method returns a number indicating the Unicode value of the character at the given index. Unicode code points range from 0 to 1,114,111. The first 128 Unicode code points are a direct match of the ASCII character encoding. charCodeAt() always returns a value that is less than 65,536.

Syntax
string.charCodeAt(index);

Argument Details
- index – An integer between 0 and 1 less than the length of the string; if unspecified, defaults to 0.

Return Value
Returns a number indicating the Unicode value of the character at the given index. It returns NaN if the given index is not between 0 and 1 less than the length of the string.

Example
var str = new String("This is string");
console.log("str.charAt(0) is:" + str.charCodeAt(0));
console.log("str.charAt(1) is:" + str.charCodeAt(1));
console.log("str.charAt(2) is:" + str.charCodeAt(2));
console.log("str.charAt(3) is:" + str.charCodeAt(3));
console.log("str.charAt(4) is:" + str.charCodeAt(4));
console.log("str.charAt(5) is:" + str.charCodeAt(5));

Output

str.charAt(0) is:84
str.charAt(1) is:104
str.charAt(2) is:105
str.charAt(3) is:115
str.charAt(4) is:32
str.charAt(5) is:105

concat()

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

Syntax

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

Argument Details

- string2...stringN – These are the strings to be concatenated.

Return Value

Returns a single concatenated string.

Example

var str1 = new String( "This is string one" );
var str2 = new String( "This is string two" );
var str3 = str1.concat( str2 );
console.log("str1 + str2 : "+str3)

Output

str1 + str2 : This is string oneThis is string two

indexOf()

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

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

**Argument Details**

- **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.

**Return Value**

Returns the index of the found occurrence, otherwise -1 if not found.

**Example**

```javascript
var str1 = new String( "This is string one" );

var index = str1.indexOf( "string" );
console.log("indexOf found String :" + index );

var index = str1.indexOf( "one" );
console.log("indexOf found String :" + index );
```

**Output**

indexOf found String :8
indexOf found String :15

---

**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**

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

**Argument Details**

- **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.

**Return Value**

Returns the index of the last found occurrence, otherwise -1 if not found.

**Example**

```javascript
var str1 = new String( "This is string one and again string" );
var index = str1.lastIndexOf( "string" );
console.log("lastIndexOf found String :" + index );

index = str1.lastIndexOf( "one" );
console.log("lastIndexOf found String :" + index );
```

**Output**

```plaintext
lastIndexOf found String :29
lastIndexOf found String :15
```

**localeCompare()**

This method returns a number indicating whether a reference string comes before or after or is the same as the given string in sorted order.

**Syntax**

```javascript
string.localeCompare( param )
```

**Argument Details**

- **param** – A string to be compared with string object.

**Return Value**

- 0 – If the string matches 100%.
- 1 – no match, and the parameter value comes before the string object’s value in the locale sort order.
- A negative value – no match, and the parameter value comes after the string object’s value in the local sort order.

**Example**
```javascript
var str1 = new String( "This is beautiful string" );
var index = str1.localeCompare( "This is beautiful string" );
console.log("localeCompare first :" + index );
```

**Output**

localeCompare first :0

**replace()**

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

The replacement string can include the following special replacement patterns –

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Inserts</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>Inserts a &quot;$&quot;.</td>
</tr>
<tr>
<td>$&amp;</td>
<td>Inserts the matched substring.</td>
</tr>
<tr>
<td>$'</td>
<td>Inserts the portion of the string that precedes the matched substring.</td>
</tr>
<tr>
<td>$'</td>
<td>Inserts the portion of the string that follows the matched substring.</td>
</tr>
<tr>
<td>$n or $nn</td>
<td>Where n or nn are decimal digits, inserts the nth parenthesized submatch string, provided the first argument was a RegExp object.</td>
</tr>
</tbody>
</table>

**Syntax**

```
string.replace(regexp/substr, newSubStr/function[, flags]);
```

**Argument Details**

- **regexp** – A RegExp object. The match is replaced by the return value of parameter #2.
- **substr** – A String that is to be replaced by newSubStr.
- **newSubStr** – The String that replaces the substring received from parameter #1.
- **function** – A function to be invoked to create the new substring.
- **flags** – A String containing any combination of the RegExp flags: g

**Return Value**

It simply returns a new changed string.
Example

```javascript
var re = /apples/gi;
var str = "Apples are round, and apples are juicy.";
var newstr = str.replace(re, "oranges");
console.log(newstr)
```

Output

oranges are round, and oranges are juicy.

Example

```javascript
var re = /\w+/s(\w+)/;
var str = "zara ali"
var newstr = str.replace(re, "$2, $1");
console.log(newstr);
```

Output

ali, zara

search()

This method executes the search for a match between a regular expression and this String object.

Syntax

```javascript
string.search(regexp);
```

Argument Details

dexp – A regular expression object. If a non-RegExp object obj is passed, it is implicitly converted to a RegExp by using new RegExp(obj).

Return Value

If successful, the search returns the index of the regular expression inside the string. Otherwise, it returns -1.

Example

```javascript
var re = /apples/gi;
```
var str = "Apples are round, and apples are juicy.";

if ( str.search(re) == -1 )
{
    console.log("Does not contain Apples");
}
else
{
    console.log("Contains Apples");
}

Output
Contains Apples

slice()
This method extracts a section of a string and returns a new string.

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

Argument Details
- **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.

Return Value
If successful, **slice** returns the index of the regular expression inside the string. Otherwise, it returns -1.

Example
```
var str = "Apples are round, and apples are juicy.";
var sliced = str.slice(3, -2);
console.log(sliced);
```

Output
les are round, and apples are juic

split()

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

Syntax

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

Argument Details

- **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.

Return Value

The split method returns the new array. Also, when the string is empty, split returns an array containing one empty string, rather than an empty array.

Example

```javascript
var str = "Apples are round, and apples are juicy.";
var splitted = str.split(" ", 3);
console.log(splitted)
```

Output

```javascript
[ 'Apples', 'are', 'round,' ]
```

substr()

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

Syntax

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

Argument Details
- **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.

**Note** – If **start** is negative, then **substr** uses it as a character index from the end of the string.

**Return Value**
The **substr()** method returns the new sub-string based on given parameters.

**Example**
```
var str = "Apples are round, and apples are juicy.";
console.log("(1,2): " + str.substr(1,2));
console.log("(-2,2): " + str.substr(-2,2));
console.log("(1): " + str.substr(1));
console.log("(-20, 2): " + str.substr(-20,2));
console.log("(20, 2): " + str.substr(20,2));
```

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

**substring()**
This method returns a subset of a String object.

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

**Argument Details**
- **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.

**Return Value**
The **substring** method returns the new sub-string based on given parameters.

**Example**

```
var str = "Apples are round, and apples are juicy.";
console.log("(1,2): " + str.substring(1,2));
console.log("(0,10): " + str.substring(0, 10));
console.log("(5): " + str.substring(5));
```

**Output**

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

**toLocaleLowerCase()**

This method is used to convert the characters within a string to lowercase while respecting the current locale. For most languages, it returns the same output as `toLowerCase`.

**Syntax**

```
string.toLocaleLowerCase( )
```

**Return Value**

Returns a string in lowercase with the current locale.

**Example**

```
var str = "Apples are round, and Apples are Juicy.";
console.log(str.toLocaleLowerCase( ));
```

**Output**

```
apples are round, and apples are juicy.
```

**toLowerCase()**

This method returns the calling string value converted to lowercase.
**Syntax**

```javascript
string.toLowerCase( )
```

**Return Value**

Returns the calling string value converted to lowercase.

**Example**

```javascript
var str = "Apples are round, and Apples are Juicy.";
console.log(str.toLowerCase( ))
```

**Output**

```
apples are round, and apples are juicy.
```

**toString()**

This method returns a string representing the specified object.

**Syntax**

```javascript
string.toString( )
```

**Return Value**

Returns a string representing the specified object.

**Example**

```javascript
var str = "Apples are round, and Apples are Juicy.";
console.log(str.toString( ))
```

**Output**

```
Apples are round, and Apples are Juicy.
```

**toUpperCase()**

This method returns the calling string value converted to uppercase.

**Syntax**

```javascript
string.toUpperCase( )
```
string.toUpperCase()

**Return Value**
Returns a string representing the specified object.

**Example**
```
var str = "Apples are round, and Apples are Juicy.";
console.log(str.toUpperCase());
```

**Output**
APPLES ARE ROUND, AND APPLES ARE JUICY.

**valueOf()**
This method returns the primitive value of a String object.

**Syntax**
```
string.valueOf()
```

**Return Value**
Returns the primitive value of a String object.

**Example**
```
var str = new String("Hello world");
console.log(str.valueOf());
```
Following is a list of methods with their description.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String.prototype.startsWith(searchString, position=0)</td>
<td>Returns true if the receiver starts with searchString; the position lets you specify where the string to be checked starts</td>
</tr>
<tr>
<td>String.prototype.endsWith(searchString, endPosition=searchString.length)</td>
<td>Returns true if the receiver ends with searchString; endPosition lets you specify where the string to be checked ends</td>
</tr>
<tr>
<td>String.prototype.includes(searchString, position=0)</td>
<td>Returns true if the receiver contains searchString; position lets you specify where the string to be searched starts</td>
</tr>
<tr>
<td>String.prototype.repeat(count)</td>
<td>Returns the receiver, concatenated count times</td>
</tr>
</tbody>
</table>

**startsWith**

The method determines if a string starts with the specified character.

**Syntax**

```
str.startsWith(searchString[, position])
```

**Parameters**

- **searchString**: The characters to be searched for at the start of this string.
- **Position**: The position in this string at which to begin searching for searchString; defaults to 0

**Return Value**

true if the string begins with the characters of the search string; otherwise, false.

**Example**

```javascript
var str = 'hello world!!!';
console.log(str.startsWith('hello'));
```

19. ES6 – New String Methods
endsWith

This function determines whether a string ends with the characters of another string.

Syntax

```
str.endsWith(matchstring[, position])
```

Parameters

- **matchstring**: The characters that the string must end with. It is case sensitive.
- **Position**: The position to match the matchstring. This parameter is optional.

Return Value

true if the string ends with the characters of the match string; otherwise, false.

Example

```
var str = 'Hello World !!! ';

console.log(str.endsWith('Hello'));
console.log(str.endsWith('Hello',5));
```

Output

```
false
true
```

includes()

The method determines if a string is a substring of the given string.

Syntax

```
str.includes(searchString[, position])
```
Parameters

- **searchString**: The substring to search for.
- **Position**: The position in this string at which to begin searching for searchString; defaults to 0

Return Value

true if the string contains the substring; otherwise, false.

Example

```javascript
var str = 'Hello World';

console.log(str.includes('hell'))
console.log(str.includes('Hell'));

console.log(str.includes('or'));
console.log(str.includes('or',1))
```

Output

false
true
true
true

`repeat()`

This function repeats the specified string for a specified number of times.

Syntax

```javascript
str.repeat(count)
```

Parameters

- **Count**: number of times the string should be repeated.

Return Value

Returns a new string.
Example

```javascript
var myBook = new String("Perl");
console.log(myBook.repeat(2));
```

The following output is displayed on successful execution of the above code.

```
PerlPerl
```

Template Literals

**Template literals** are string literals that allow embedded expressions. **Template strings** use back-ticks (``) rather than the single or double quotes. A template string could thus be written as:

```javascript
var greeting = `Hello World!`;
```

String Interpolation and Template literals

Template strings can use placeholders for string substitution using the ${ } syntax, as demonstrated.

**Example 1**

```javascript
var name = "Brendan";
console.log('Hello, ${name}!');
```

The following output is displayed on successful execution of the above code.

```
Hello, Brendan!
```

**Example 2: Template literals and expressions**

```javascript
var a = 10;
var b = 10;
console.log(`The sum of ${a} and ${b} is ${a+b}`);
```

The following output is displayed on successful execution of the above code.

```
The sum of 10 and 10 is 20
```

**Example 3: Template literals and function expression**

```javascript
function fn() { return "Hello World"; }
console.log(`Message: ${fn()} !!`);
```
The following output is displayed on successful execution of the above code.

```
Message: Hello World !
```

**Multiline Strings and Template Literals**

Template strings can contain multiple lines.

**Example**

```javascript
var multiLine = 'This is a string with multiple lines';
console.log(multiLine)
```

The following output is displayed on successful execution of the above code.

```
This is a string with multiple line
```

**String.raw()**

ES6 includes the tag function String.raw for raw strings, where backslashes have no special meaning. `String.raw` enables us to write the backslash as we would in a regular expression literal. Consider the following example.

```javascript
var text = "Hello \n World"
console.log(text)

var raw_text=String.raw`Hello \n World`
console.log(raw_text)
```

The following output is displayed on successful execution of the above code.

```
Hello World
Hello \n World
```
The static `String.fromCharCode()` method returns a string created by using the specified sequence of unicode code points. The function throws a RangeError if an invalid code point is passed.

```javascript
console.log(String.fromCharCode(42))
console.log(String.fromCharCode(65, 90))
```

The following output is displayed on successful execution of the above code.

```
* 
AZ
```
The use of variables to store values poses the following limitations:

- Variables are scalar in nature. In other words, a variable declaration can only contain a single at a time. This means that to store \( n \) values in a program, \( n \) variable declarations will be needed. Hence, the use of variables is not feasible when one needs to store a larger collection of values.

- Variables in a program are allocated memory in random order, thereby making it difficult to retrieve/read the values in the order of their declaration.

JavaScript introduces the concept of arrays to tackle the same.

An array is a homogenous collection of values. To simplify, an array is a collection of values of the same data type. It is a user-defined type.

**Features of an Array**

- An array declaration allocates sequential memory blocks.

- Arrays are static. This means that an array once initialized cannot be resized.

- Each memory block represents an array element.

- Array elements are identified by a unique integer called as the subscript/index of the element.

- Arrays too, like variables, should be declared before they are used.

- Array initialization refers to populating the array elements.

- Array element values can be updated or modified but cannot be deleted.

**Declaring and Initializing Arrays**

To declare and initialize an array in JavaScript use the following syntax:

```javascript
var array_name; //declaration
array_name=[val1,val2,valn..]   //initialization
OR
var array_name=[val1,val2..valn]
```

**Note:** The pair of [] is called the dimension of the array.
For example, a declaration like: `var numlist = [2,4,6,8]` will create an array as shown in the following figure.

![Array Diagram](image)

The array pointer refers to the first element by default.

**Accessing Array Elements**

The array name followed by the subscript is used to refer to an array element.

Following is the syntax for the same.

```
array_name[subscript]
```

**Example: Simple Array**

```javascript
var alphas;
alphas=['1','2','3','4']
console.log(alphas[0]);
console.log(alphas[1]);
```

The following output is displayed on successful execution of the above code.

```
1
2
```

**Example: Single Statement Declaration and Initialization**

```javascript
var nums=[1,2,3,3]
console.log(nums[0]);
console.log(nums[1]);
console.log(nums[2]);
console.log(nums[3]);
```

The following output is displayed on successful execution of the above code.

```
1
2
3
3
```
Array Object

An array can also be created using the Array object. The Array constructor can be passed as -

- A numeric value that represents the size of the array or
- A list of comma separated values

The following Examples create an array using this method.

Example

```javascript
var arr_names = new Array(4)

for(var i=0;i<arr_names.length;i++)
{
    arr_names[i] = i * 2
    console.log(arr_names[i])
}
```

The following output is displayed on successful execution of the above code.

```
0
2
4
6
```

Example: Array Constructor Accepts Comma-separated Values

```javascript
var names = new Array("Mary","Tom","Jack","Jill")

for(var i=0;i<names.length;i++)
{
    console.log(names[i])
}
```

The following output is displayed on successful execution of the above code.

```
Mary
Tom
Jack
Jill
```
# Array Methods

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>concat()</code></td>
<td>Returns a new array comprised of this array joined with other array(s) and/or value(s)</td>
</tr>
<tr>
<td><code>every()</code></td>
<td>Returns true if every element in this array satisfies the provided testing function</td>
</tr>
<tr>
<td><code>filter()</code></td>
<td>Creates a new array with all of the elements of this array for which the provided filtering function returns true</td>
</tr>
<tr>
<td><code>forEach()</code></td>
<td>Calls a function for each element in the array</td>
</tr>
<tr>
<td><code>indexOf()</code></td>
<td>Returns the first (least) index of an element within the array equal to the specified value, or -1 if none is found</td>
</tr>
<tr>
<td><code>join()</code></td>
<td>Joins all elements of an array into a string</td>
</tr>
<tr>
<td><code>lastIndexOf()</code></td>
<td>Returns the last (greatest) index of an element within the array equal to the specified value, or -1 if none is found</td>
</tr>
<tr>
<td><code>map()</code></td>
<td>Creates a new array with the results of calling a provided function on every element in this array</td>
</tr>
<tr>
<td><code>pop()</code></td>
<td>Removes the last element from an array and returns that element</td>
</tr>
<tr>
<td><code>push()</code></td>
<td>Adds one or more elements to the end of an array and returns the new length of the array</td>
</tr>
<tr>
<td><code>reduce()</code></td>
<td>Applies a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value</td>
</tr>
<tr>
<td><code>reduceRight()</code></td>
<td>Applies a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value</td>
</tr>
<tr>
<td><code>reverse()</code></td>
<td>Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>shift()</code></td>
<td>Removes the first element from an array and returns that element</td>
</tr>
<tr>
<td><code>slice()</code></td>
<td>Extracts a section of an array and returns a new array</td>
</tr>
<tr>
<td><code>some()</code></td>
<td>Returns true if at least one element in this array satisfies the provided testing function</td>
</tr>
<tr>
<td><code>toSource()</code></td>
<td>Represents the source code of an object</td>
</tr>
<tr>
<td><code>sort()</code></td>
<td>Sorts the elements of an array</td>
</tr>
<tr>
<td><code>splice()</code></td>
<td>Adds and/or removes elements from an array</td>
</tr>
<tr>
<td><code>toString()</code></td>
<td>Returns a string representing the array and its elements</td>
</tr>
<tr>
<td><code>unshift()</code></td>
<td>Adds one or more elements to the front of an array and returns the new length of the array</td>
</tr>
</tbody>
</table>

**concat()**

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

**Syntax**

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

**Parameters**

- `valueN` – Arrays and/or values to concatenate to the resulting array.

**Return Value**

Returns a new array.

**Example**

```javascript
var alpha = ["a", "b", "c"];
var numeric = [1, 2, 3];
```
```javascript
var alphaNumeric = alpha.concat(numeric);
console.log("alphaNumeric : " + alphaNumeric );
```

**Output**

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

**every()**

every method tests whether all the elements in an array passes the test implemented by the provided function.

**Syntax**

```
array.every(callback[, thisObject]);
```

**Parameter Details**

- **callback** – Function to test for each element.
- **thisObject** – Object to use as this when executing callback.

**Return Value**

Returns `true` if every element in this array satisfies the provided testing function.

**Example**

```javascript
function isBigEnough(element, index, array) {
    return (element >= 10);
}

var passed = [12, 5, 8, 130, 44].every(isBigEnough);
console.log("Test Value : " + passed );
```

**Output**

```
Test Value : false
```

**filter()**

filter() method creates a new array with all elements that pass the test implemented by the provided function.
ES6

Syntax

array.filter(callback[, thisObject]);

Parameter Details

- **callback** – Function to test for each element.
- **thisObject** – Object to use as this when executing callback.

Return Value

Returns created array.

Example

```
function isBigEnough(element, index, array) {
    return (element >= 10);
}

var passed = [12, 5, 8, 130, 44].filter(isBigEnough);
console.log("Test Value : " + passed);
```

Output

Test Value : 12, 130, 44

forEach()

forEach() method calls a function for each element in the array

Syntax

array.forEach(callback[, thisObject]);

Parameter Details

- **callback** – Function to test for each element.
- **thisObject** – Object to use as this when executing callback.

Return Value

Returns created array.
Example: forEach()

```javascript
var nums=new Array(12,13,14,15)

console.log("Printing original array......")

nums.forEach(function(val,index){
    console.log(val)
});

nums.reverse()  // reverses the array element

console.log("Printing Reversed array....")

nums.forEach(function(val,index){
    console.log(val)
});
```

The following output is displayed on successful execution of the above code.

```
Printing Original Array....
12
13
14
15

Printing Reversed array...
15
14
13
12
```

indexOf()

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

Syntax
array.indexOf(searchElement[, fromIndex]);

Parameter Details

- **searchElement** – Element to locate in the array.
- **fromIndex** – The index at which to begin the search. Defaults to 0, i.e. the whole array will be searched. If the index is greater than or equal to the length of the array, -1 is returned.

Return Value

Returns the index of the found element.

Example

```javascript
var index = [12, 5, 8, 130, 44].indexOf(8);
console.log("index is : " + index );
```

Output

```
index is : 2
```

join()

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

Syntax

```javascript
array.join(separator);
```

Parameter Details

- **separator** – Specifies a string to separate each element of the array. If omitted, the array elements are separated with a comma.

Return Value

Returns a string after joining all the array elements.

Example

```javascript
var arr = new Array("First","Second","Third");
var str = arr.join();
```
```javascript
console.log("str : " + str);

var str = arr.join(", ");
console.log("str : " + str);

var str = arr.join(" + ");
console.log("str : " + str);
```

**Output**

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

---

**lastIndexOf()**

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.

**Syntax**

```javascript
array.lastIndexOf(searchElement[, fromIndex]);
```

**Parameter Details**

- **searchElement** – Element to locate in the array.

- **fromIndex** – The index at which to start searching backwards. Defaults to the array's length, i.e., the whole array will be searched. If the index is greater than or equal to the length of the array, the whole array will be searched. If negative, it is taken as the offset from the end of the array.

**Return Value**

Returns the index of the found element from the last.

---

**Example**

```javascript
var index = [12, 5, 8, 130, 44].lastIndexOf(8);
console.log("index is : " + index);
```
Output

index is : 3

map()

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

Syntax

array.map(callback[, thisObject]);

Parameter Details

- **callback** – Function that produces an element of the new Array from an element of the current one.

- **thisObject** – Object to use as this when executing callback.

Return Value

Returns the created array.

Example

```javascript
var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
console.log("roots is : " + roots);
```

Output

roots is : 1,2,3

pop()

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

Syntax
array.pop();

**Return Value**

Returns the removed element from the array.

**Example**

```javascript
var numbers = [1, 4, 9];

var element = numbers.pop();
console.log("element is : " + element);

var element = numbers.pop();
console.log("element is : " + element);
```

**Output**

```
element is : 9
element is : 4
```

**push()**

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

**Syntax**

```javascript
array.push(element1, ..., elementN);
```

**Parameter Details**

- element1, ..., elementN: The elements to add to the end of the array.

**Return Value**

Returns the length of the new array.

**Example**

```javascript
var numbers = new Array(1, 4, 9);
```
```javascript
var length = numbers.push(10);
console.log("new numbers is : "+ numbers);
length = numbers.push(20);
console.log("new numbers is : "+ numbers);
```

**Output**

```
new numbers is : 1,4,9,10
new numbers is : 1,4,9,10,20
```

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

**Syntax**

```javascript
array.reduce(callback[, initialValue]);
```

**Parameter Details**
- **callback** – Function to execute on each value in the array.
- **initialValue** – Object to use as the first argument to the first call of the callback.

**Return Value**
Returns the reduced single value of the array.

**Example**

```javascript
var total = [0, 1, 2, 3].reduce(function(a, b){ return a + b; });
console.log("total is : " + total);
```

**Output**

```
total is : 6
```

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

**Syntax**

```javascript
```
array.reduceRight(callback[, initialValue]);

Parameter Details

- **callback** – Function to execute on each value in the array.
- **initialValue** – Object to use as the first argument to the first call of the callback.

Return Value

Returns the reduced right single value of the array.

Example

```javascript
var total = [0, 1, 2, 3].reduceRight(function(a, b){ return a + b; });
console.log("total is : " + total);
```

Output

```
total is : 6
```

reverse()

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

Syntax

```javascript
array.reverse();
```

Return Value

Returns the reversed single value of the array.

Example

```javascript
var arr = [0, 1, 2, 3].reverse();
console.log("Reversed array is : " + arr);
```

Output

```
Reversed array is : 3,2,1,0
```
**shift()**

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

**Syntax**

```javascript
array.shift();
```

**Return Value**

Returns the removed single value of the array

**Example**

```javascript
var arr = [10, 1, 2, 3].shift();
console.log("Shifted value is : ", arr);
```

**Output**

```
Shifted value is : 10
```

**slice()**

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

**Syntax**

```javascript
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.

**Return Value**

Returns the extracted array based on the passed parameters.

**Example**

```javascript
var arr = ["orange", "mango", "banana", "sugar", "tea"]; 
console.log("arr.slice( 1, 2) : ", arr.slice( 1, 2));
```
```javascript
console.log("arr.slice( 1, 3) : " + arr.slice( 1, 3));
```

**Output**

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

### some()

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

**Syntax**

```javascript
array.some(callback[, thisObject]);
```

**Parameter Details**

- **callback** – Function to test for each element.
- **thisObject** – Object to use as this when executing callback.

**Return Value**

If some element passes the test, then it returns true, otherwise false.

**Example**

```javascript
function isBigEnough(element, index, array)
{
    return (element >= 10);
}

var retval = [2, 5, 8, 1, 4].some(isBigEnough);
console.log("Returned value is : " + retval);

var retval = [12, 5, 8, 1, 4].some(isBigEnough);
console.log("Returned value is : " + retval);
```

**Output**

```
Returned value is : false
```
sort()

sort() method sorts the elements of an array.

Syntax

array.sort( compareFunction );

Parameter Details

compareFunction – Specifies a function that defines the sort order. If omitted, the array is sorted lexicographically.

Return Value

Returns a sorted array.

Example

```javascript
var arr = new Array("orange", "mango", "banana", "sugar");
var sorted = arr.sort();
console.log("Returned string is : " + sorted);
```

Output

```
Returned string is : banana,mango,orange,sugar
```

splice()

splice() method changes the content of an array, adding new elements while removing old elements.

Syntax

array.splice(index, howMany, [element1][, ..., elementN]);

Parameter Details

- **index** – Index at which to start changing the array.
- **howMany** – An integer indicating the number of old array elements to remove. If howMany is 0, no elements are removed.
- **element1, ..., elementN** – The elements to add to the array. If you don’t specify any elements, splice simply removes the elements from the array.
**Return Value**

Returns the extracted array based on the passed parameters.

**Example**

```javascript
var arr = ['orange', 'mango', 'banana', 'sugar', 'tea'];

var removed = arr.splice(2, 0, 'water');

console.log("After adding 1: " + arr);
console.log("removed is: " + removed);

removed = arr.splice(3, 1);

console.log("After adding 1: " + arr);
console.log("removed is: " + removed);
```

On compiling, it will generate the same code in JavaScript

**Output**

```
After adding 1: orange,mango,water,banana,sugar,tea
removed is:
After adding 1: orange,mango,water,sugar,tea
removed is: banana
```

**toString()**

toString() method returns a string representing the source code of the specified array and its elements.

**Syntax**

```javascript
array.toString();
```

**Return Value**

Returns a string representing the array.

**Example**
var arr = new Array("orange", "mango", "banana", "sugar");
var str = arr.toString();
console.log("Returned string is : " + str);

Output

Returned string is : orange,mango,banana,sugar

unshift()

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

Syntax

array.unshift( element1, ..., elementN );

Parameter Details

- element1, ..., elementN – The elements to add to the front of the array.

Return Value

Returns the length of the new array. It returns undefined in IE browser.

Example

var arr = new Array("orange", "mango", "banana", "sugar");
var length = arr.unshift("water");
console.log("Returned array is : " + arr);
console.log("Length of the array is : " + length);

Output

Returned array is : water,orange,mango,banana,sugar
Length of the array is : 5

ES6 – Array Methods

Following are some new array methods introduced in ES6.

Array.prototype.find
**find** lets you iterate through an array and get the first element back that causes the given callback function to return true. Once an element has been found, the function immediately returns. It's an efficient way to get at just the first item that matches a given condition.

### Example

```javascript
var numbers = [1, 2, 3];
var oddNumber = numbers.find((x) => x % 2 == 1);
console.log(oddNumber); // 1
```

The following output is displayed on successful execution of the above code.

```
1
```

**Note:** The **ES5 filter()** and the **ES6 find()** are not synonymous. Filter always returns an array of matches (and will return multiple matches), find always returns the actual element.

### Array.prototype.findIndex

**findIndex** behaves similar to find, but instead of returning the element that matched, it returns the index of that element.

```javascript
var numbers = [1, 2, 3];
var oddNumber = numbers.findIndex((x) => x % 2 == 1);
console.log(oddNumber); // 0
```

The above example will return the index of the value 1 (0) as output.

### Array.prototype.entries

**entries** is a function that returns an Array Iterator that can be used to loop through the array’s keys and values. Entries will return an array of arrays, where each child array is an array of [index, value].

```javascript
var numbers = [1, 2, 3];
var val= numbers.entries();
console.log(val.next().value);
console.log(val.next().value);
console.log(val.next().value);
```

The following output is displayed on successful execution of the above code.

```
[0,1]
[1.2]
```
Alternatively, we can also use the spread operator to get back an array of the entries in one go.

```javascript
var numbers = [1, 2, 3];
var val = numbers.entries();
console.log([...val]);
```

The following output is displayed on successful execution of the above code.

```
[[0,1],[1,2],[2,3]]
```

**Array.from**

*Array.from()* enables the creation of a new array from an array like object. The basic functionality of *Array.from()* is to convert two kinds of values to Arrays:

- Array-like values
- Iterable values like Set and Map

**Example**

```javascript
"use strict"
for (let i of Array.from('hello'))
{
    console.log(i)
}
```

The following output is displayed on successful execution of the above code.

```
h
e
l
l
o
```

**Array.prototype.keys()**

This function returns the array indexes.
Example

```javascript
console.log(Array.from(['a', 'b'].keys()))
```

The following output is displayed on successful execution of the above code.

```
[ 0, 1 ]
```

Array Traversal using for...in loop

One can use the for... in loop to traverse through an array.

```javascript
"use strict"
var nums=[1001,1002,1003,1004]
for(let j in nums)
{
    console.log(nums[j])
}
```

The loop performs an index-based array traversal. The following output is displayed on successful execution of the above code.

```
1001
1002
1003
1004
```

Arrays in JavaScript

JavaScript supports the following concepts about Arrays:

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-dimensional arrays</td>
<td>JavaScript supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array</td>
</tr>
<tr>
<td>Passing arrays to functions</td>
<td>You can pass to the function a pointer to an array by specifying the array's name without an index</td>
</tr>
<tr>
<td>Return array from functions</td>
<td>Allows a function to return an array</td>
</tr>
</tbody>
</table>

Multidimensional Arrays
An array element can reference another array for its value. Such arrays are called as **multi-dimensional arrays**. ES6 supports the concept of multi-dimensional arrays. The simplest form of a multi-dimensional array is a two-dimensional array.

**Declaring a Two-dimensional Array**

```javascript
var arr_name=[ [val1,val2,val3],[v1,v2,v3] ]
```

**Accessing a Two-dimensional Array Element**

```javascript
var arr_name[initial_array_index][referenced_array_index]
```

The following example better explains this concept.

**Example**

```javascript
var multi=[[1,2,3],[23,24,25]]
console.log(multi[0][0])
console.log(multi[0][1])
console.log(multi[0][2])
console.log(multi[1][0])
console.log(multi[1][1])
console.log(multi[1][2])
```

The above example initially declares an array with 2 elements. Each of these elements refer to another array having 3 elements. Following is the pictorial representation of the above array.

While referring to an array element here, the subscript of the initial array element must be followed by the subscript of the referenced array element. This is illustrated in the above code.

The following output is displayed on successful execution of the above code.
Example: Passing Arrays to Functions

```javascript
var names=new Array("Mary","Tom","Jack","Jill")
function disp(arr_names)
{
    for(var i=0;i<arr_names.length;i++)
    {
        console.log(names[i])
    }
}
disp(names)
```

The following output is displayed on successful execution of the above code.

Mary
Tom
Jack
Jill

Example: Function Returning an Array

```javascript
function disp()
{
    return new Array("Mary","Tom","Jack","Jill")
}
var nums=disp()
for(var i in nums)
{
    console.log(nums[i])
}
```

The following output is displayed on successful execution of the above code.
Array De-structuring

JavaScript supports de-structuring in the context of an array.

Example

```javascript
var arr=[12,13]
var[x,y]=arr
console.log(x)
console.log(y)
```

The following output is displayed on successful execution of the above code.

```
12
13
```
The Date object is a datatype built into the JavaScript language. Date objects are created with the new Date() as shown in the following syntax.

Once a Date object is created, a number of methods allow you to operate on it. Most methods simply allow you to get and set the year, month, day, hour, minute, second, and millisecond fields of the object, using either local time or UTC (universal, or GMT) time.

The ECMAScript standard requires the Date object to be able to represent any date and time, to millisecond precision, within 100 million days before or after 1/1/1970. This is a range of plus or minus 273,785 years, so JavaScript can represent date and time till the year 275755.

You can use any of the following syntax to create a Date object using Date() constructor.

- new Date()
- new Date(milliseconds)
- new Date(datestring)
- new Date(year,month,date[,hour,minute,second,millisecond])

**Note:** Parameters in the brackets are always optional.

### Date Properties

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor</td>
<td>Specifies the function that creates an object's prototype</td>
</tr>
<tr>
<td>prototype</td>
<td>The prototype property allows you to add properties and methods to an object</td>
</tr>
</tbody>
</table>

### Constructor

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

**Syntax**

date.constructor
**Return value**

Returns the function that created this object’s instance.

**Example: constructor**

```javascript
var dt = new Date();
console.log("dt.constructor is : "+ dt.constructor);
```

The following output is displayed on successful execution of the above code.

```javascript
dt.constructor is : function Date() { [native code] }
```

**prototype**

The prototype property allows you to add properties and methods to any object (Number, Boolean, String, Date, etc.). Note: Prototype is a global property which is available with almost all the objects.

**Syntax**

```javascript
object.prototype.name = value
```

**Example: Date.prototype**

```javascript
var myBook = new book("Perl", "Mohtashim");
book.prototype.price = null;
myBook.price = 100;
console.log("Book title is : " + myBook.title + "<br>");
console.log("Book author is : " + myBook.author + "<br>");
console.log("Book price is : " + myBook.price + "<br>");
```

The output of the above code will be as given below:

**Book title is : Perl**
**Book author is : Mohtashim**
**Book price is : 100**
Date Methods

Following is a list of different date methods along with the description.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date()</td>
<td>Returns today's date and time</td>
</tr>
<tr>
<td>getDate()</td>
<td>Returns the day of the month for the specified date according to the local time</td>
</tr>
<tr>
<td>getDay()</td>
<td>Returns the day of the week for the specified date according to the local time</td>
</tr>
<tr>
<td>getFullYear()</td>
<td>Returns the year of the specified date according to the local time</td>
</tr>
<tr>
<td>getHours()</td>
<td>Returns the hour in the specified date according to the local time</td>
</tr>
<tr>
<td>getMilliseconds()</td>
<td>Returns the milliseconds in the specified date according to the local time</td>
</tr>
<tr>
<td>getMinutes()</td>
<td>Returns the minutes in the specified date according to the local time</td>
</tr>
<tr>
<td>getMonth()</td>
<td>Returns the month in the specified date according to the local time</td>
</tr>
<tr>
<td>getSeconds()</td>
<td>Returns the seconds in the specified date according to the local time</td>
</tr>
<tr>
<td>getTime()</td>
<td>Returns the numeric value of the specified date as the number of milliseconds since January 1, 1970, 00:00:00 UTC</td>
</tr>
<tr>
<td>getTimezoneOffset()</td>
<td>Returns the time-zone offset in minutes for the current locale</td>
</tr>
<tr>
<td>getUTCDate()</td>
<td>Returns the day (date) of the month in the specified date according to the universal time</td>
</tr>
<tr>
<td>getUTCDay()</td>
<td>Returns the day of the week in the specified date according to the universal time</td>
</tr>
<tr>
<td>getUTCFullYear()</td>
<td>Returns the year in the specified date according to the universal time</td>
</tr>
<tr>
<td>getUTCHours()</td>
<td>Returns the hours in the specified date according to the universal time</td>
</tr>
<tr>
<td>getUTCMilliseconds()</td>
<td>Returns the milliseconds in the specified date according to the universal time</td>
</tr>
<tr>
<td>getUTCMinutes()</td>
<td>Returns the minutes in the specified date according to the universal time</td>
</tr>
<tr>
<td>getUTCMonth()</td>
<td>Returns the month in the specified date according to the universal time</td>
</tr>
<tr>
<td>getUTCSeconds()</td>
<td>Returns the seconds in the specified date according to the universal time</td>
</tr>
<tr>
<td>setDate()</td>
<td>Sets the day of the month for a specified date according to the local time</td>
</tr>
<tr>
<td>setFullYear()</td>
<td>Sets the full year for a specified date according to the local time</td>
</tr>
<tr>
<td>setHours()</td>
<td>Sets the hours for a specified date according to the local time</td>
</tr>
<tr>
<td>setMilliseconds()</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>setMinutes()</code></td>
<td>Sets the minutes for a specified date according to the local time</td>
</tr>
<tr>
<td><code>setMonth()</code></td>
<td>Sets the month for a specified date according to the local time</td>
</tr>
<tr>
<td><code>setSeconds()</code></td>
<td>Sets the seconds for a specified date according to the local time</td>
</tr>
<tr>
<td><code>setTime()</code></td>
<td>Sets the Date object to the time represented by a number of milliseconds since January 1, 1970, 00:00:00 UTC</td>
</tr>
<tr>
<td><code>setUTCDate()</code></td>
<td>Sets the day of the month for a specified date according to the universal time</td>
</tr>
<tr>
<td><code>setUTCFullYear()</code></td>
<td>Sets the full year for a specified date according to the universal time</td>
</tr>
<tr>
<td><code>setUTCHours()</code></td>
<td>Sets the hour for a specified date according to the universal time</td>
</tr>
<tr>
<td><code>setUTCMilliseconds()</code></td>
<td>Sets the milliseconds for a specified date according to the universal time</td>
</tr>
<tr>
<td><code>setUTCMinutes()</code></td>
<td>Sets the minutes for a specified date according to the universal time</td>
</tr>
<tr>
<td><code>setUTCMonth()</code></td>
<td>Sets the month for a specified date according to the universal time</td>
</tr>
<tr>
<td><code>setUTCSeconds()</code></td>
<td>Sets the seconds for a specified date according to the universal time</td>
</tr>
<tr>
<td><code>toDateString()</code></td>
<td>Returns the &quot;date&quot; portion of the Date as a human-readable string</td>
</tr>
<tr>
<td><code>toLocaleDateString()</code></td>
<td>Returns the &quot;date&quot; portion of the Date as a string, using the current locale's conventions</td>
</tr>
<tr>
<td><code>toLocaleString()</code></td>
<td>Converts a date to a string, using the current locale's conventions</td>
</tr>
<tr>
<td><code>toLocaleTimeString()</code></td>
<td>Returns the &quot;time&quot; portion of the Date as a string, using the current locale's conventions</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>toString()</code></td>
<td>Returns a string representing the specified Date object</td>
</tr>
<tr>
<td><code>toTimeString()</code></td>
<td>Returns the &quot;time&quot; portion of the Date as a human-readable string</td>
</tr>
<tr>
<td><code>toUTCString()</code></td>
<td>Converts a date to a string, using the universal time convention</td>
</tr>
<tr>
<td><code>valueOf()</code></td>
<td>Returns the primitive value of a Date object</td>
</tr>
</tbody>
</table>

**Date()**

Javascript Date() method returns today’s date and time and does not need any object to be called.

**Syntax**

```javascript
Date()
```

**Return Value**

Returns today's date and time.

**Example**

```javascript
var dt = Date();
console.log("Current Date ",dt);
```

**Output**

```
Current Date Tue Sep 20 2016 22:24:31 GMT+0530 (India Standard Time)
```

**getDate()**

Javascript date getDate() method returns the day of the month for the specified date according to local time. The value returned by getDate is an integer between 1 and 31.

**Syntax**

```javascript
Date.getDate()
```

**Return Value**

Returns today's date and time.
**Example**

```javascript
var dt = new Date("December 25, 1995 23:15:00");
console.log("getDate() : " + dt.getDate() );
```

**Output**

`getDate() : 25`

**getDay()**

Javascript date `getDay()` method returns the day of the week for the specified date according to local time. The value returned by `getDay` is an integer corresponding to the day of the week: 0 for Sunday, 1 for Monday, 2 for Tuesday, and so on.

**Syntax**

```javascript
Date.getDay()
```

**Return Value**

Returns the day of the week for the specified date according to local time.

**Example**

```javascript
var dt = new Date("December 25, 1995 23:15:00");
console.log("getDay() : " + dt.getDay() );
```

**Output**

`getDay() : 1`

**getFullYear()**

Javascript date `getFullYear()` method returns the year of the specified date according to local time. The value returned by `getFullYear` is an absolute number. For dates between the years 1000 and 9999, `getFullYear` returns a four-digit number, for example, 2008.

**Syntax**

```javascript
Date.getFullYear()
```

**Return Value**

Returns the year of the specified date according to local time.
Example

```javascript
var dt = new Date("December 25, 1995 23:15:00");
console.log("getFullYear() : " + dt.getFullYear());
```

Output

getFullYear() : 1995

getHours()

Javascript Date getHours() method returns the hour in the specified date according to local time. The value returned by getHours is an integer between 0 and 23.

Syntax

```javascript
Date.getHours()
```

Return Value

Returns the hour in the specified date according to local time.

Example

```javascript
var dt = new Date("December 25, 1995 23:15:00");
console.log("getHours() : " + dt.getHours());
```

Output

getHours() : 23

getMilliseconds()

Javascript Date getMilliseconds() method returns the milliseconds in the specified date according to local time. The value returned by getMilliseconds is a number between 0 and 999.

Syntax

```javascript
Date.getMilliseconds()
```

Return Value

Returns the milliseconds in the specified date according to local time.

Example
```
var dt = new Date("December 25, 1995 23:15:00");
console.log("getMilliseconds() : " + dt.getMilliseconds());
```

**Output**
```
getMilliseconds() : 0
```

### getMinutes()

Javascript date `getMinutes()` method returns the minutes in the specified date according to local time. The value returned by `getMinutes` is an integer between 0 and 59.

**Syntax**
```
Date.getMinutes()
```

**Return Value**

Returns the minutes in the specified date according to local time.

**Example**
```
var dt = new Date("December 25, 1995 23:15:00");
console.log("getMinutes() : " + dt.getMinutes());
```

**Output**
```
getMinutes() : 15
```

### getMonth()

Javascript date `getMonth()` method returns the month in the specified date according to local time. The value returned by `getMonth` is an integer between 0 and 11. 0 corresponds to January, 1 to February, and so on.

**Syntax**
```
Date.getMonth()
```

**Return Value**

Returns the Month in the specified date according to local time.

**Example**
```
var dt = new Date("December 25, 1995 23:15:00");
```
```javascript
console.log("getMinutes() : " + dt.getMinutes());
```

**Output**

```
getMonth() : 11
```

### getSeconds()

Javascript date `getSeconds()` method returns the seconds in the specified date according to local time. The value returned by `getSeconds` is an integer between 0 and 59.

**Syntax**

```javascript
Date.getSeconds()
```

**Return Value**

Returns the seconds in the specified date according to local time.

**Example**

```javascript
var dt = new Date("December 25, 1995 23:15:00");
console.log("getSeconds() : " + dt.getSeconds());
```

**Output**

```
getSeconds() : 0
```

### getTime()

Javascript date `getTime()` method returns the numeric value corresponding to the time for the specified date according to universal time. The value returned by the `getTime` method is the number of milliseconds since 1 January 1970 00:00:00.

You can use this method to help assign a date and time to another Date object.

**Syntax**

```javascript
Date.getTime()
```

**Return Value**

Returns the numeric value corresponding to the time for the specified date according to universal time.

**Example**
```javascript
var dt = new Date("December 25, 1995 23:15:00");
console.log("getTime() : " + dt.getTime());
```

**Output**

```
getTime() : 819913500000
```

---

**getTimezoneOffset()**

Javascript date `getTimezoneOffset()` method returns the time-zone offset in minutes for the current locale. The time-zone offset is the minutes in difference, the Greenwich Mean Time (GMT) is relative to your local time.

For example, if your time zone is GMT+10, -600 will be returned. Daylight savings time prevents this value from being a constant.

**Syntax**

```
Date.getTimezoneOffset()
```

**Return Value**

Returns the time-zone offset in minutes for the current locale.

**Example**

```javascript
var dt = new Date("December 25, 1995 23:15:00");
console.log("getTimezoneoffset() : " + dt.getTimezoneOffset());
```

**Output**

```
getTimezoneoffset() : -330
```

---

**getUTCDate()**

Javascript date `getUTCDate()` method returns the day of the month in the specified date according to universal time. The value returned by `getUTCDate` is an integer between 1 and 31.

**Syntax**

```
Date.getUTCDate()
```

**Return Value**

Returns the day of the month in the specified date according to universal time.
Example

```javascript
var dt = new Date( "December 25, 1995 23:15:20" );
console.log("getUTCDate() : " + dt.getUTCDate() );
```

Output

```
getUTCDate() : 25
```

g UTC D ay()

Javascript date getUTCDay() method returns the day of the week in the specified date according to universal time. The value returned by getUTCDay is an integer corresponding to the day of the week: 0 for Sunday, 1 for Monday, 2 for Tuesday, and so on.

Syntax

```
Date.getUTCDay()
```

Return Value

Returns the day of the week in the specified date according to universal time.

Example

```javascript
var dt = new Date("December 25, 1995 23:15:20");
console.log("getUTCDay() : " + dt.getUTCDay() );
```

Output

```
getUTCDay() : 1
```

g UTC F ullYear()

Javascript date getUTCFullYear() method returns the year in the specified date according to universal time. The value returned by getUTCFullYear is an absolute number that is compliant with year-2000, for example, 2008.

Syntax

```
Date.getUTCFullYear()
```

Return Value

Returns the year in the specified date according to universal time.
Example

```javascript
var dt = new Date( "December 25, 1995 23:15:20" );
console.log("getUTCFullYear() : " + dt.getUTCFullYear() );
```

Output

```
getUTCFullYear() : 1995
```

getUTCHours()

Javascript date getUTCHours() method returns the hours in the specified date according to universal time. The value returned by getUTCHours is an integer between 0 and 23.

Syntax

```
Date.getUTCHours ()
```

Return Value

Returns the hours in the specified date according to universal time.

Example

```javascript
var dt = new Date( "December 25, 1995 23:15:20" );
console.log("getUTCHours() : " + dt.getUTCHours() );
```

Output

```
getUTCHours() : 17
```

getUTCMilliseconds()

Javascript date getUTCMilliseconds() method returns the milliseconds in the specified date according to universal time. The value returned by getUTCMilliseconds is an integer between 0 and 999.

Syntax

```
Date.getUTCMilliseconds ()
```

Return Value

Returns the milliseconds in the specified date according to universal time.
Example

```javascript
var dt = new Date( "December 25, 1995 23:15:20" );
console.log("getUTCMilliseconds() : " + dt.getUTCMilliseconds())
```

Output

```
getUTCMilliseconds() : 0
```

**getUTCMilliseconds()**

Javascript date getUTCMilliseconds() method returns the minutes in the specified date according to universal time. The value returned by getUTCMilliseconds is an integer between 0 and 59.

**Syntax**

```javascript
Date.getUTCMilliseconds ()
```

**Return Value**

Returns the minutes in the specified date according to universal time.

Example

```javascript
var dt = new Date( "December 25, 1995 23:15:20" );
console.log("getUTCMinutes() : " + dt.getUTCMinutes() );
```

Output

```
getUTCMinutes() : 45
```

**getUTCMinutes()**

Javascript date getUTCMinutes() method returns the minutes in the specified date according to universal time. The value returned by getUTCMinutes is an integer between 0 and 59.

**Syntax**

```javascript
Date.getUTCMinutes ()
```

**Return Value**

Returns the minutes in the specified date according to universal time.

**getUTCMonth()**

Javascript date getUTCMonth() method returns the month in the specified date according to universal time. The value returned by getUTCMonth is an integer between 0 and 11 corresponding to the month. 0 for January, 1 for February, 2 for March, and so on.

**Syntax**

```javascript
Date.getUTCMonth ()
```

**Return Value**

Returns the month in the specified date according to universal time.
Example

```javascript
var dt = new Date( "December 25, 1995 23:15:20" );
console.log("getUTCMonth() : " + dt.getUTCMonth() );
```

Output

```
getUTCMonth() : 11
```

getUTCSeconds()

Javascript date getUTCSeconds() method returns the seconds in the specified date according to universal time. The value returned by getUTCSeconds is an integer between 0 and 59.

Syntax

```javascript
Date.getUTCSeconds ()
```

Return Value

Returns the seconds in the specified date according to universal time.

Example

```javascript
var dt = new Date( "December 25, 1995 23:15:20" );
console.log("getUTCSeconds() : " + dt.getUTCSeconds() );
```

The following output is displayed on successful execution of the above code.

```
getUTCSeconds() : 20
```

setDate()

Javascript date setDate() method sets the day of the month for a specified date according to local time.

Syntax

```javascript
Date.setDate( dayValue )
```

Parameter

- **dayValue** – An integer from 1 to 31, representing the day of the month.

Example
```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setDate( 24 );
console.log( dt )
```

**Output**

```
Sun Aug 24 2008 23:30:00 GMT+0530 (India Standard Time)
```

### setFullYear()

Javascript date `setFullYear()` method sets the full year for a specified date according to local time.

```
Date.getFullYear(yearValue[, monthValue[, dayValue]])
```

**Parameter**

- `yearValue` − An integer specifying the numeric value of the year, for example, 2008.
- `monthValue` − An integer between 0 and 11 representing the months January through December.
- `dayValue` − An integer between 1 and 31 representing the day of the month. If you specify the `dayValue` parameter, you must also specify the `monthValue`.
- If you do not specify the `monthValue` and `dayValue` parameters, the values returned from the `getMonth` and `getDate` methods are used.

**Example**

```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.getFullYear( 2000 );
console.log( dt )
```

**Output**

```
Mon Aug 28 2000 23:30:00 GMT+0530 (India Standard Time)
```

### setHours()

Javascript date `setHours()` method sets the hours for a specified date according to local time.

**Syntax**
**Date.setHours(hoursValue[, minutesValue[, secondsValue[, msValue]]]])**

**Parameter**
- **hoursValue** – An integer between 0 and 23, representing the hour.
- **minutesValue** – An integer between 0 and 59, representing the minutes.
- **secondsValue** – An integer between 0 and 59, representing the seconds. If you specify the secondsValue parameter, you must also specify the minutesValue.
- **msValue** – A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

If you do not specify the minutesValue, secondsValue, and msValue parameters, the values returned from the getUTCMinutes, getUTCSeconds, and getMilliseconds methods are used.

**Example**

```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setHours(02);
console.log(dt);
```

**Output**

```
Thu Aug 28 2008 02:30:00 GMT+0000 (UTC)
```

---

**setMilliseconds()**

Javascript date setMilliseconds() method sets the milliseconds for a specified date according to local time.

**Syntax**

```
Date.setMilliseconds(millisecondsValue)
```

**Parameter**
- **millisecondsValue** – A number between 0 and 999, representing the milliseconds.

If you specify a number outside the expected range, the date information in the Date object is updated accordingly. For example, if you specify 1010, the number of seconds is incremented by 1, and 10 is used for the milliseconds.

**Example**

```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setMilliseconds(10);
console.log(dt);
```
```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setMilliseconds( 1010 );
console.log( dt );
```

**Output**

Thu Aug 28 2008 23:30:01 GMT+0530 (India Standard Time)

---

### setMinutes()

Javascript date setMinutes() method sets the minutes for a specified date according to local time.

**Syntax**

Date.setMinutes(minutesValue[, secondsValue[, msValue]])

**Parameter**

- **minutesValue** – An integer between 0 and 59, representing the minutes.

- **secondsValue** – An integer between 0 and 59, representing the seconds. If you specify the secondsValue parameter, you must also specify the minutesValue.

- **msValue** – A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

If you do not specify the secondsValue and msValue parameters, the values returned from getSeconds and getMilliseconds methods are used.

**Example**

```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setMinutes( 45 );
console.log( dt );
```

**Output**

Thu Aug 28 2008 23:45:00 GMT+0530 (India Standard Time)

---

### setMonth()  

Javascript date setMonth() method sets the month for a specified date according to local time.
Syntax

```javascript
Date.setMonth(monthValue[, dayValue])
```

Parameter

- **monthValue** – An integer between 0 and 11 (representing the months January through December).
- **dayValue** – An integer from 1 to 31, representing the day of the month.
- **msValue** – A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

If you do not specify the dayValue parameter, the value returned from the getDate method is used. If a parameter you specify is outside of the expected range, setMonth attempts to update the date information in the Date object accordingly. For example, if you use 15 for monthValue, the year will be incremented by 1 (year + 1), and 3 will be used for month.

Example

```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setMonth( 2 );
console.log( dt );
```

Output

```
Fri Mar 28 2008 23:30:00 GMT+0530 (India Standard Time)
```

`setSeconds()`

Javascript date `setSeconds()` method sets the seconds for a specified date according to local time

Syntax

```javascript
Date.setSeconds(secondsValue[, msValue])
```

Parameter

- **secondsValue** – An integer between 0 and 59.
- **msValue** – A number between 0 and 999, representing the milliseconds.
If you do not specify the msValue parameter, the value returned from the getMilliseconds method is used. If a parameter you specify is outside of the expected range, setSeconds attempts to update the date information in the Date object accordingly. For example, if you use 100 for secondsValue, the minutes stored in the Date object will be incremented by 1, and 40 will be used for seconds.

Example

```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setSeconds( 80 );
console.log( dt );
```

Output


**setTime()**

Javascript date setTime() method sets the Date object to the time represented by a number of milliseconds since January 1, 1970, 00:00:00 UTC.

Syntax

```javascript
Date.setTime(timeValue)
```

Parameter

- **timeValue** — An integer representing the number of milliseconds since 1 January 1970, 00:00:00 UTC.

Example

```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setTime( 5000000 );
console.log( dt );
```

Output

Thu Jan 01 1970 06:53:20 GMT+0530 (India Standard Time)
**setUTCDate()**

Javascript date `setUTCDate()` method sets the day of the month for a specified date according to universal time.

**Syntax**

```
Date.setUTCDate(dayValue)
```

**Parameter**

- `dayValue` – An integer from 1 to 31, representing the day of the month.

**Example**

```
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setUTCDate( 20 );
console.log( dt );
```

**Output**

```
Wed Aug 20 2008 23:30:00 GMT+0530 (India Standard Time)
```

**setUTCFullYear()**

Javascript date `setUTCFullYear()` method sets the full year for a specified date according to universal time.

**Syntax**

```
Date.setUTCFullYear(yearValue[, monthValue[, dayValue]])
```

**Parameter**

- `yearValue` – An integer specifying the numeric value of the year, for example, 2008.
- `monthValue` – An integer between 0 and 11 representing the months January through December.
- `dayValue` – An integer between 1 and 31 representing the day of the month. If you specify the `dayValue` parameter, you must also specify the `monthValue`. 
If you do not specify the monthValue and dayValue parameters, the values returned from the getMonth and getDate methods are used. If a parameter you specify is outside of the expected range, setUTCFullYear attempts to update the other parameters and the date information in the Date object accordingly. For example, if you specify 15 for monthValue, the year is incremented by 1 (year + 1), and 3 is used for the month.

Example

```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setUTCFullYear( 2006 );
console.log( dt );
```

Output

```
Mon Aug 28 2006 23:30:00 GMT+0530 (India Standard Time)
```

setUTCHours()

Javascript date setUTCHours() method sets the hour for a specified date according to local time.

Syntax

```
Date.setUTCHours(hoursValue[, minutesValue[, secondsValue[, msValue]]])
```

Parameter

- **hoursValue** – An integer between 0 and 23, representing the hour.
- **minutesValue** – An integer between 0 and 59, representing the minutes.
- **secondsValue** – An integer between 0 and 59, representing the seconds. If you specify the secondsValue parameter, you must also specify the minutesValue.
- **msValue** – A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

If you do not specify the minutesValue, secondsValue, and msValue parameters, the values returned from the getUTCMilliseconds, getUTCSeconds, and getUTCMilliseconds methods are used.

If a parameter you specify is outside the expected range, setUTCHours attempts to update the date information in the Date object accordingly. For example, if you use 100 for secondsValue, the minutes will be incremented by 1 (min + 1), and 40 will be used for seconds.
Example

```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setUTCHours( 12);
console.log( dt );
```

Output

```
Thu Aug 28 2008 17:30:00 GMT+0530 (India Standard Time)
```

**setUTCMilliseconds()**

Javascript date setUTCMilliseconds() method sets the milliseconds for a specified date according to universal time.

Syntax

```javascript
Date.setUTCMilliseconds(millisecondsValue)
```

Parameter

- millisecondsValue – A number between 0 and 999, representing the milliseconds

If a parameter you specify is outside the expected range, setUTCMilliseconds attempts to update the date information in the Date object accordingly. For example, if you use 1100 for millisecondsValue, the seconds stored in the Date object will be incremented by 1, and 100 will be used for milliseconds.

Example

```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setUTCMilliseconds( 1100);
console.log( dt );
```

Output

```
Thu Aug 28 2008 23:30:01 GMT+0530 (India Standard Time)
```

**setUTCMinutes()**

Javascript date setUTCMinutes() method sets the minutes for a specified date according to universal time.

Syntax
Date.setUTCMinutes(minutesValue[, secondsValue[, msValue]])

Parameter

- **minutesValue** – An integer between 0 and 59, representing the minutes.

- **secondsValue** – An integer between 0 and 59, representing the seconds. If you specify the secondsValue parameter, you must also specify the minutesValue.

- **msValue** – A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

If you do not specify the secondsValue and msValue parameters, the values returned from getUTCSeconds and getUTCMilliseconds methods are used.

If a parameter you specify is outside of the expected range, setUTCMinutes attempts to update the date information in the Date object accordingly. For example, if you use 100 for secondsValue, the minutes (minutesValue) will be incremented by 1 (minutesValue + 1), and 40 will be used for seconds.

**Example**

```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setUTCMinutes(65);
console.log( dt );
```

**Output**

```
Fri Aug 29 2008 00:35:00 GMT+0530 (India Standard Time)
```

**setUTCMonth()**

Javascript date setUTCMonth() method sets the month for a specified date according to universal time.

**Syntax**

```javascript
Date.setUTCMonth ( monthvalue )
```

**Parameter**

- **monthValue** – An integer between 0 and 11, representing the month.

**Example**

```javascript
var dt = new Date( "Aug 28, 2008 23:30:00" );
dt.setUTCMonth(5);
```
```javascript
console.log( dt );
```

**Output**
```
Sat Jun 28 2008 23:30:00 GMT+0530 (India Standard Time)
```

### setUTCSeconds()

Javascript date `setUTCSeconds()` method sets the seconds for a specified date according to universal time.

#### Syntax
```javascript
Date.setUTCSeconds(secondsValue[, msValue])
```

#### Parameter
- **secondsValue** – An integer between 0 and 59, representing the seconds.
- **msValue** – A number between 0 and 999, representing the milliseconds.

If you do not specify the msValue parameter, the value returned from the `getUTCMilliseconds` methods is used.

If a parameter you specify is outside the expected range, `setUTCSeconds` attempts to update the date information in the Date object accordingly. For example, if you use 100 for `secondsValue`, the minutes stored in the Date object will be incremented by 1, and 40 will be used for seconds.

### Example
```javascript
var dt = new Date("Aug 28, 2008 23:30:00");

dt.setUTCSeconds(65);

console.log( dt );
```

**Output**
```
Thu Aug 28 2008 23:31:05 GMT+0530 (India Standard Time)
```

### toDateString()

Javascript date `toDateString()` method returns the date portion of a Date object in human readable form.
Syntax
Date.toDateString()

Return Value
Returns the date portion of a Date object in human readable form.

Example
var dt = new Date(1993, 6, 28, 14, 39, 7);
console.log( "Formated Date : " + dt.toDateString() )

Output
Formated Date : Wed Jul 28 1993

toLocaleDateString()
Javascript date toLocaleDateString() method converts a date to a string, returning the "date" portion using the operating system's locale's conventions.

Syntax
Date.toLocaleString()

Return Value
Returns the "date" portion using the operating system's locale's conventions.

Example
var dt = new Date(1993, 6, 28, 14, 39, 7);
console.log( "Formated Date : " + dt.toLocaleDateString() )

Output
Formated Date : 7/28/1993

toLocaleString()
Javascript date toLocaleString() method converts a date to a string, using the operating system's local conventions.

The toLocaleString method relies on the underlying operating system in formatting dates. It converts the date to a string using the formatting convention of the operating system
where the script is running. For example, in the United States, the month appears before the date (04/15/98), whereas in Germany the date appears before the month (15.04.98).

**Syntax**

```javascript
Date.toLocaleString()
```

**Return Value**

Returns the formatted date in a string format.

**Example**

```javascript
var dt = new Date(1993, 6, 28, 14, 39, 7);
console.log( "Formated Date : " + dt.toLocaleString() );
```

**Output**

Formated Date : 7/28/1993, 2:39:07 PM

---

**toLocaleTimeString()**

Javascript date `toLocaleTimeString()` method converts a date to a string, returning the "date" portion using the current locale's conventions.

The `toLocaleTimeString` method relies on the underlying operating system in formatting dates. It converts the date to a string using the formatting convention of the operating system where the script is running. For example, in the United States, the month appears before the date (04/15/98), whereas in Germany, the date appears before the month (15.04.98).

**Syntax**

```javascript
Date.toLocaleTimeString()
```

**Return Value**

Returns the formatted date in a string format.

**Example**

```javascript
var dt = new Date(1993, 6, 28, 14, 39, 7);
console.log( "Formated Date : " + dt.toLocaleTimeString() );
```

**Output**

Formated Date : 2:39:07 PM
**toString()**

This method returns a string representing the specified Date object.

**Syntax**

```javascript
Date.toString()
```

**Return Value**

Returns a string representing the specified Date object.

**Example**

```javascript
var dateobject = new Date(1993, 6, 28, 14, 39, 7);
stringobj = dateobject.toString();
console.log( "String Object : " + stringobj );
```

**Output**

```
```

---

**toTimeString()**

This method returns the time portion of a Date object in human readable form.

**Syntax**

```javascript
Date.toTimeString()
```

**Return Value**

Returns the time portion of a Date object in human readable form.

**Example**

```javascript
var dateobject = new Date(1993, 6, 28, 14, 39, 7);
console.log( dateobject.toTimeString() );
```

**Output**

```
14:39:07
```
toUTCString()

This method converts a date to a string, using the universal time convention.

Syntax

```javascript
Date.toUTCString()
```

Return Value

Returns converted date to a string, using the universal time convention.

Example

```javascript
var dateobject = new Date(1993, 6, 28, 14, 39, 7);
console.log(dateobject.toUTCString());
```

Output

```
Wed, 28 Jul 1993 09:09:07 GMT
```

valueOf()

This method returns the primitive value of a Date object as a number data type, the number of milliseconds since midnight 01 January, 1970 UTC.

Syntax

```javascript
Date.valueOf()
```

Return Value

Returns the primitive value of a Date object.

Example

```javascript
var dateobject = new Date(1993, 6, 28, 14, 39, 7);
console.log(dateobject.valueOf());
```

Output

```
743850547000
```
The math object provides you properties and methods for mathematical constants and functions. Unlike other global objects, Math is not a constructor. All the properties and methods of Math are static and can be called by using Math as an object without creating it.

**Math Properties**

Following is a list of all Math properties and its description.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Euler's constant and the base of natural logarithms, approximately 2.718</td>
</tr>
<tr>
<td>LN2</td>
<td>Natural logarithm of 2, approximately 0.693</td>
</tr>
<tr>
<td>LN10</td>
<td>Natural logarithm of 10, approximately 2.302</td>
</tr>
<tr>
<td>LOG2E</td>
<td>Base 2 logarithm of E, approximately 1.442</td>
</tr>
<tr>
<td>LOG10E</td>
<td>Base 10 logarithm of E, approximately 0.434</td>
</tr>
<tr>
<td>PI</td>
<td>Ratio of the circumference of a circle to its diameter, approximately 3.14159</td>
</tr>
<tr>
<td>SQRT1_2</td>
<td>Square root of 1/2; equivalently, 1 over the square root of 2, approximately 0.707</td>
</tr>
<tr>
<td>SQRT2</td>
<td>Square root of 2, approximately 1.414</td>
</tr>
</tbody>
</table>

**Math-E**

This is an Euler's constant and the base of natural logarithms, approximately 2.718.

**Syntax**

Math.E

**Example**

```javascript
console.log(Math.E) // the root of the natural logarithm: ~2.718
```

**Output**

```
2.718281828459045
```

**Math-LN2**

It returns the natural logarithm of 2 which is approximately 0.693.

**Syntax**

Math.LN2
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Example</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.LN2</td>
<td>Natural logarithm of 2, approximately 0.693</td>
<td><code>console.log(Math.LN2)</code></td>
<td>0.6931471805599453</td>
</tr>
<tr>
<td>Math.LN10</td>
<td>Natural logarithm of 10, approximately 2.302</td>
<td><code>console.log(Math.LN10)</code></td>
<td>2.302585092994046</td>
</tr>
<tr>
<td>Math.LOG2E</td>
<td>Base 2 logarithm of E, approximately 1.442</td>
<td><code>console.log(Math.LOG2E)</code></td>
<td>1.4426950408889634</td>
</tr>
</tbody>
</table>
### Math - LOG10E

Base 10 logarithm of E, approximately 0.434.

**Syntax**

```
Math.LOG10E
```

**Example**

```
console.log(Math.LOG10E) // the base 10 logarithm of Math.E: 0.434
```

**Output**

```
0.4342944819032518
```

### Math - PI

Ratio of the circumference of a circle to its diameter, approximately 3.14159

**Syntax**

```
Math.PI
```

**Example**

```
console.log(Math.PI) // the ratio of a circle's circumference to its diameter: ~3.142
```

**Output**

```
3.141592653589793
```

### Math - SQRT1_2

Square root of 1/2; equivalently, 1 over the square root of 2, approximately 0.707

**Syntax**

```
Math.SQRT1_2
```

**Example**

```
console.log(Math.SQRT1_2) // the square root of 1/2: ~0.707
```
Output

0.7071067811865476

Math - SQRT2

Square root of 2, approximately 1.414

Syntax

Math.SQRT2

Example

console.log(Math.SQRT2) // the square root of 2: ~1.414

Output

1.4142135623730951

Exponential Functions

The basic exponential function is Math.pow(), and there are convenience functions for square root, cube root, and powers of e, as shown in the following table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.pow(x, y)</td>
<td>Returns x raised to the power y</td>
</tr>
<tr>
<td>Math.sqrt(x)</td>
<td>Returns the square root of the number x</td>
</tr>
<tr>
<td>Math.cbrt(x)</td>
<td>Returns the cube root of the number x</td>
</tr>
<tr>
<td>Math.exp(x)</td>
<td>Equivalent to Math.pow(Math.E, x)</td>
</tr>
<tr>
<td>Math.expm1(x)</td>
<td>Equivalent to Math.exp(x) – 1</td>
</tr>
<tr>
<td>Math.hypot(x1, x2,...)</td>
<td>Returns the square root of the sum of arguments</td>
</tr>
</tbody>
</table>

Pow()

This method returns the base to the exponent power, that is, base to the power exponent.
Syntax

```javascript
Math.pow(x, y)
```

**Parameter**
- `x`: represents base
- `y`: represents the exponent

**Return Value**
Returns the base to the exponent power.

**Example**

```javascript
console.log("---Math.pow()---")
console.log("math.pow(2,3) : "+Math.pow(2, 3))
console.log("Math.pow(1.7, 2.3) : "+Math.pow(1.7, 2.3))
```

**Output**

```javascript
---Math.pow()---
math.pow(2,3) : 8
Math.pow(1.7, 2.3) : 3.38695291147646
```

**sqrt()**

This method returns the square root of a number. If the value of a number is negative, sqrt returns NaN.

**Syntax**

```javascript
Math.sqrt ( x );
```

**Parameter**
- `x`: represents a number

**Return Value**
Returns the square root of the number.

**Example**

```javascript
console.log("---Math.sqrt()---")
console.log("Math.sqrt(16) : "+Math.sqrt(16))
```
```javascript
console.log("Math.sqrt(15.5) : "+Math.sqrt(15.5))
```

### Output

```javascript
---Math.sqrt()---
Math.sqrt(16) : 4
Math.sqrt(15.5) : 3.9370039370059056
```

### `cbrt()`

This method returns the cube root of a number.

#### Syntax

```javascript
Math.cbrt ( x );
```

#### Parameter

- `x`: represents a number

#### Return Value

Returns the cube root of the number.

#### Example

```javascript
console.log("---Math.cbrt()---")
console.log("Math.cbrt(27) : "+Math.cbrt(27))
console.log("Math.cbrt(22) : "+Math.cbrt(22))
```

### Output

```javascript
---Math.cbrt()---
Math.cbrt(27) : 3
Math.cbrt(22) : 2.802039330655387
```

### `exp()`

Equivalent to `Math.pow(Math.E, x)`

#### Syntax
Math.exp ( x );

Parameter
• x: represents a number

Return Value
Returns the exponential value of the variable x.

Example
```
console.log("---Math.exp()---")
console.log("Math.exp(1) : "+Math.exp(1))
console.log("Math.exp(5.5) : "+Math.exp(5.5))
```

Output
```
---Math.exp()---
Math.exp(1) : 2.718281828459045
Math.exp(5.5) : 244.69193226422036
```

expm1(X)
Equivalent to Math.exp(x) – 1

Syntax
```
Math.expm1( x );
```

Parameter
• x: represents a number

Return Value
Returns the value of Math.exp(x) – 1

Example
```
console.log("---Math.expm1()---")
console.log("Math.expm1(1) : "+Math.expm1(1))
console.log("Math.expm1(5.5) : "+Math.expm1(5.5))
```
Output

---Math.expm1()---
Math.expm1(1) : 1.718281828459045
Math.expm1(5.5) : 243.69193226422038

Math.hypot(x1, x2,...)

Returns the square root of the sum of the arguments

Syntax

Math.hypot( x1,x2.. ) ;

Parameter

- X1 and x2..: represents numbers

Return Value

Returns the square root of the sum of all the numbers passed a argument

Example

```javascript
console.log("---Math.hypot()---")
console.log("Math.hypot(3,4) : "+Math.hypot(3,4))
console.log("Math.hypot(2,3,4) : "+Math.hypot(2,3,4))
```

Output

---Math.hypot()---
Math.hypot(3,4) : 5
Math.hypot(2,3,4) : 5.385164807134504

Logarithmic Functions

The basic natural logarithm function is Math.log(). In JavaScript, “log” means “natural logarithm.” ES6 introduced Math.log10 for convenience.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.log(x)</td>
<td>Natural logarithm of x</td>
</tr>
</tbody>
</table>
Math.log10(x) | Base 10 logarithm of x
---|---
Math.log2(x) | Base 2 logarithm of x
Math.log1p(x) | Natural logarithm of 1 + x

Math.log(x)
Returns the natural logarithm of X

Syntax
Math.log(x)

Parameter
- x: represents a number

Example
```
console.log("---Math.log()---")
console.log("Math.log(17.5): "+Math.log(17.5))
```

Output
```
---Math.log()---
Math.log(Math.E): 1
Math.log(17.5): 2.8622008809294686
```

Math.log10(x)
Returns the base 10 logarithm of X

Syntax
Math.log10(x)

Parameter
- x: represents a number
Example

```javascript
console.log("---Math.log10()---")
console.log("Math.log10(10): " + Math.log10(10))
```

Output

```
---Math.log10()---
Math.log10(10): 1
Math.log10(16.7): 1
```

**Math.log2(x)**

Returns the base 2 logarithm of X

**Syntax**

```
Math.log2(x)
```

**Parameter**

- `x`: represents a number

**Example**

```javascript
console.log("---Math.log2()---")
```

Output

```
---Math.log2()---
Math.log2(2): 1
Math.log2(5): 2.321928094887362
```

**Math.log1p(x)**

Returns the natural logarithm of 1+x
## Syntax

```javascript
Math.log1p(x)
```

### Parameter

- `x`: represents a number

### Example

```javascript
console.log("---Math.log1p()---")
console.log("Math.log1p(17.5): "+Math.log1p(17.5))
```

The following output is displayed on successful execution of the above code.

---Math.log1p()---
Math.log1p(Math.E - 1): 1
Math.log1p(17.5): 2.917770732084279

### Miscellaneous Algebraic Functions

Following is a list of miscellaneous algebraic functions with their description.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Math.abs(x)</code></td>
<td>Absolute value of <code>x</code></td>
</tr>
<tr>
<td><code>Math.sign(x)</code></td>
<td>The sign of <code>x</code>: if <code>x</code> is negative, -1; if <code>x</code> is positive, 1; and if <code>x</code> is 0, 0</td>
</tr>
<tr>
<td><code>Math.ceil(x)</code></td>
<td>The ceiling of <code>x</code>: the smallest integer greater than or equal to <code>x</code></td>
</tr>
<tr>
<td><code>Math.floor(x)</code></td>
<td>The floor of <code>x</code>: the largest integer less than or equal to <code>x</code></td>
</tr>
<tr>
<td><code>Math.trunc(x)</code></td>
<td>The integral part of <code>x</code> (all fractional digits are removed)</td>
</tr>
<tr>
<td><code>Math.round(x)</code></td>
<td><code>x</code> rounded to the nearest integer</td>
</tr>
<tr>
<td><code>Math.min(x1, x2,...)</code></td>
<td>Returns the minimum argument</td>
</tr>
<tr>
<td><code>Math.max(x1, x2,...)</code></td>
<td>Returns the maximum argument</td>
</tr>
</tbody>
</table>

### Abs()

This method returns the absolute value of a number.

## Syntax

```javascript
Math.abs( x )
```

### Parameter

- `X`: represents a number
Return Value
Returns the absolute value of a number

Example
```javascript
console.log("---Math.abs()---")
console.log("Math.abs(-5.5) : "+Math.abs(-5.5))
console.log("Math.abs(5.5) : "+Math.abs(5.5))
```

Output
```javascript
---Math.abs()---
Math.abs(-5.5) : 5.5
Math.abs(5.5) : 5.5
```

sign()
Returns the sign of x

Syntax
```javascript
Math.sign( x ) ;
```

Parameter
- X: represents a number

Return Value
Returns -1 if x is negative; 1 if x is positive; 0 if x is 0.

Example
```javascript
console.log("---Math.sign()---")
console.log("Math.sign(-10.5) : "+Math.sign(-10.5))
console.log("Math.sign(6.77) : "+Math.sign(6.77))
```

Output
```javascript
---Math.sign()---
Math.sign(-10.5) : -1
Math.sign(6.77) : 1
```
round()

It rounds off the number to the nearest integer.

Syntax

Math.round( x )

Parameter

- X: represents a number

Example

```javascript
console.log("---Math.round()---")
console.log("Math.round(7.2) : " + Math.round(7.2))
console.log("Math.round(-7.7) : " + Math.round(-7.7))
```

Output

```
---Math.round()---
Math.round(7.2) : 7
Math.round(-7.7) : -8
```

trunc()

It returns the integral part of x (all fractional digits are removed).

Syntax

Math.trunc( x )

Parameter

- X: represents a number

Example

```javascript
console.log("---Math.trunc()---")
console.log("Math.trunc(7.7) : " + Math.trunc(7.7))
console.log("Math.trunc(-5.8) : " + Math.trunc(-5.8))
```
Output

---Math.trunc()---
Math.trunc(7.7) : 7
Math.trunc(-5.8) : -5

floor()
The floor of x: the largest integer less than or equal to x.

Syntax

Math.floor( x ) ;

Parameter

- X: represents a number

Example

console.log("---Math.floor()---")
console.log("Math.floor(2.8) : "+Math.floor(2.8))
console.log("Math.floor(-3.2) : "+Math.floor(-3.2))

Output

---Math.floor()---
Math.floor(2.8) : 2
Math.floor(-3.2) : -4

ceil()
This method returns the smallest integer greater than or equal to a number.

Syntax

Math.ceil ( x ) ;

Parameter

- X: represents a number
Example

```javascript
console.log("---Math.ceil()---")
console.log("Math.ceil(2.2) : " + Math.ceil(2.2))
console.log("Math.ceil(-3.8) : " + Math.ceil(-3.8))
```

Output

---Math.ceil()---
Math.ceil(2.2) : 3
Math.ceil(-3.8) : -3

**min()**

This method returns the smallest of zero or more numbers. If no arguments are given, the result is +Infinity.

**Syntax**

```javascript
Math.min( x1, x2, ... );
```

**Parameter**

- X1, x2, x3...: represents a series of numbers

Example

```javascript
console.log("---Math.min()---")
console.log("Math.min(1, 2) : " + Math.min(1, 2))
console.log("Math.min(3, 0.5, 0.66) : " + Math.min(3, 0.5, 0.66))
console.log("Math.min(3, 0.5, -0.66) : " + Math.min(3, 0.5, -0.66))
```

Output

---Math.min()---
Math.min(1, 2) : 1
Math.min(3, 0.5, 0.66) : 0.5
Math.min(3, 0.5, -0.66) : -0.66
**max()**

This method returns the largest of zero or more numbers. If no arguments are given, the results is –Infinity.

**Syntax**

```
Math.max(x1,x2,x3..)
```

**Parameter**
- x1, x2, x3.. : represents a series of numbers

**Example**

```javascript
console.log("---Math.max()---")
console.log("Math.max(3, 0.5, 0.66) : "+Math.max(3, 0.5, 0.66))
console.log("Math.max(-3, 0.5, -0.66) : "+Math.max(-3, 0.5, -0.66))
```

The following output is displayed on successful execution of the above code.

```
---Math.max()---
Math.max(3, 0.5, 0.66) : 3
Math.max(-3, 0.5, -0.66) : 0.5
```

**Trigonometric Functions**

All trigonometric functions in the Math library operate on radians, not degrees.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Math.sin(x)</code></td>
<td>Sine of x radians</td>
</tr>
<tr>
<td><code>Math.cos(x)</code></td>
<td>Cosine of x radians</td>
</tr>
<tr>
<td><code>Math.tan(x)</code></td>
<td>Tangent of x radians</td>
</tr>
<tr>
<td><code>Math.asin(x)</code></td>
<td>Inverse sine (arcsin) of x (result in radians)</td>
</tr>
<tr>
<td><code>Math.acos(x)</code></td>
<td>Inverse cosine (arccos) of x (result in radians)</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Math.atan(x)</td>
<td>Inverse tangent (arctan) of x (result in radians)</td>
</tr>
<tr>
<td>Math.atan2(y, x0)</td>
<td>Counterclockwise angle (in radians) from the x-axis to the point (x, y)</td>
</tr>
</tbody>
</table>

**Math.sin(x)**

This function returns the sine of x radians.

**Syntax**

```javascript
Math.sin(x)
```

**Parameter**

- X : represents a number

**Example**

```javascript
console.log("---Math.sin()---")
```

**Output**

---Math.sin()---
Math.sin(Math.PI/2): 1
Math.sin(Math.PI/4): 0.7071067811865475

**Math.cos(x)**

It returns the cosine of x radians.

**Syntax**

```javascript
Math.cos(x)
```

**Parameter**

- X : represents a number
Example

```javascript
console.log("---Math.cos()---")
```

Output

```plaintext
---Math.cos()---
Math.cos(Math.PI): -1
Math.cos(Math.PI/4): 0.7853981633974483
```

**Math.tan(x)**

This function returns the tangent of x.

**Syntax**

```javascript
Math.tan(x)
```

**Parameter**

- X : represents a number

Example

```javascript
console.log("---Math.tan()---")
console.log("Math.tan(0): "+Math.tan(0))
```

Output

```plaintext
---Math.tan()---
Math.tan(Math.PI/4): 0.9999999999999999
Math.tan(0): 0
```

**Math.asin(x)**

This function returns the inverse sine of x.
Syntax

```
Math.asin(x)
```

Parameter

- **X**: represents a number

Example

```javascript
console.log("---Math.asin()---")
console.log("Math.asin(0): "+Math.asin(0))
```

Output

```
---Math.asin()---
Math.asin(0): 0
Math.asin(Math.SQRT1_2): 0.7853981633974484
```

**Math.acos(x)**

This function returns the inverse cosine of x.

Syntax

```
Math.acos(x)
```

Parameter

- **X**: represents a number

Example

```javascript
console.log("---Math.acos()---")
console.log("Math.acos(0): "+Math.acos(0))
console.log("Math.acos(Math.SQRT1_2): "+Math.acos(Math.SQRT1_2))
```

Output

```
---Math.acos()---
Math.acos(0): 1.5707963267948966
Math.acos(Math.SQRT1_2): 0.7853981633974483
```
**Math.atan(x)**

It returns the inverse tangent of x.

**Syntax**

```javascript
Math.atan(x)
```

**Parameter**

- `x`: represents a number

**Example**

```javascript
console.log("---Math.atan()---")
console.log("Math.atan(0): " + Math.atan(0))
console.log("Math.atan(Math.SQRT1_2): " + Math.atan(Math.SQRT1_2))
```

**Output**

```
---Math.atan()---
Math.atan(0): 0
Math.atan(Math.SQRT1_2): 0.6154797086703874
```

**Math.atan2()**

This method returns the arctangent of the quotient of its arguments. The atan2 method returns a numeric value between -π and π representing the angle theta of an (x, y) point.

**Syntax**

```javascript
Math.atan2(x, y)
```

**Parameter**

- `x` and `y`: represent numbers

**Example**

```javascript
console.log("---Math.atan2()---")
console.log("Math.atan2(0): " + Math.atan2(0, 1))
console.log("Math.atan2(Math.SQRT1_2): " + Math.atan2(Math.SQRT1_2, 1))
```

**Output**

```javascript
---Math.atan2()---
Math.atan2(0): 0
Math.atan2(Math.SQRT1_2): 0.6154797086703874
```
Math.atan2()

Math.atan2(0): 0
Math.atan2(Math.SQRT1_2): 0.7853981633974483

Math.random()

The Math.random() function returns a pseudorandom number between 0 (inclusive) and 1 (exclusive).

Example: Pseudorandom Number Generation (PRNG)

```javascript
var value1 = Math.random();
console.log("First Test Value : " + value1 );
var value2 = Math.random();
console.log("Second Test Value : " + value2 );
var value3 = Math.random();
console.log("Third Test Value : " + value3 );
var value4 = Math.random();
console.log("Fourth Test Value : " + value4 );
```

Output

```
First Test Value : 0.5782922627404332
Second Test Value : 0.5624510529451072
Third Test Value : 0.9336334094405174
Fourth Test Value : 0.4002739654388279
```
A regular expression is an object that describes a pattern of characters. Regular expressions are often abbreviated “regex” or “regexp”.

The JavaScript RegExp class represents regular expressions, and both String and RegExp define methods that use regular expressions to perform powerful pattern-matching and search-and-replace functions on the text.

A regular expression can be defined as:

```
var pattern = new RegExp(pattern, attributes);
OR
var pattern = /pattern/attributes;
```

The attribute can have any combination of the following values.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Global match</td>
</tr>
<tr>
<td>I</td>
<td>Ignore case</td>
</tr>
<tr>
<td>M</td>
<td>Multiline; treat the beginning and end characters (^ and $) as working over multiple lines (i.e., match the beginning or the end of each line (delimited by \n or \r), not only the very beginning or end of the whole input string)</td>
</tr>
<tr>
<td>U</td>
<td>Unicode; treat the pattern as a sequence of unicode code points</td>
</tr>
<tr>
<td>Y</td>
<td>Sticky; matches only from the index indicated by the lastIndex property of this regular expression in the target string (and does not attempt to match from any later indexes)</td>
</tr>
</tbody>
</table>

**Constructing Regular Expressions**

**Brackets**

Brackets ([ ]) have a special meaning when used in the context of regular expressions. They are used to find a range of characters.
<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[...]</td>
<td>Any one character between the brackets</td>
</tr>
<tr>
<td>[^...]</td>
<td>Any one character not between the brackets</td>
</tr>
<tr>
<td>[0-9]</td>
<td>It matches any decimal digit from 0 through 9</td>
</tr>
<tr>
<td>[a-z]</td>
<td>It matches any character from lowercase a through lowercase z</td>
</tr>
<tr>
<td>[A-Z]</td>
<td>It matches any character from uppercase A through uppercase Z</td>
</tr>
<tr>
<td>[a-zA-Z]</td>
<td>It matches any character from lowercase a through uppercase Z</td>
</tr>
</tbody>
</table>

The ranges shown above are general; you could also use the range [0-3] to match any decimal digit ranging from 0 through 3, or the range [b-v] to match any lowercase character ranging from b through v.

**Quantifiers**

The frequency or position of the bracketed character sequences and the single characters can be denoted by a special character. Each special character has a specific connotation. The +, *, ?, and $ flags all follow a character sequence.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+</td>
<td>It matches any string containing at least one p</td>
</tr>
<tr>
<td>p*</td>
<td>It matches any string containing zero or more p's</td>
</tr>
<tr>
<td>p?</td>
<td>It matches any string containing one or more p's</td>
</tr>
<tr>
<td>p{N}</td>
<td>It matches any string containing a sequence of N p's</td>
</tr>
<tr>
<td>p{2,3}</td>
<td>It matches any string containing a sequence of two or three p's</td>
</tr>
<tr>
<td>p{2, }</td>
<td>It matches any string containing a sequence of at least two p's</td>
</tr>
<tr>
<td>p$</td>
<td>It matches any string with p at the end of it</td>
</tr>
<tr>
<td>Pattern</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>^p</td>
<td>It matches any string with p at the beginning of it</td>
</tr>
<tr>
<td>[^a-zA-Z]</td>
<td>It matches any string not containing any of the characters ranging from a through z and A through Z</td>
</tr>
<tr>
<td>p.p</td>
<td>It matches any string containing p, followed by any character, in turn followed by another p</td>
</tr>
<tr>
<td>^.{2}$</td>
<td>It matches any string containing exactly two characters</td>
</tr>
<tr>
<td>&lt;b&gt;(.*)&lt;/b&gt;</td>
<td>It matches any string enclosed within &lt;b&gt; and &lt;/b&gt;</td>
</tr>
<tr>
<td>p(hp)*</td>
<td>It matches any string containing a p followed by zero or more instances of the sequence hp</td>
</tr>
</tbody>
</table>

**Literal Characters**

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphanumeric</td>
<td></td>
</tr>
<tr>
<td>\0</td>
<td>The NULL character (\u0000)</td>
</tr>
<tr>
<td>\t</td>
<td>Tab (\u0009)</td>
</tr>
<tr>
<td>\n</td>
<td>Newline (\u000A)</td>
</tr>
<tr>
<td>\v</td>
<td>Vertical tab (\u000B)</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed (\u000C)</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return (\u000D)</td>
</tr>
<tr>
<td>\xnn</td>
<td>The Latin character specified by the hexadecimal number nn; for example, \x0A is the same as \n</td>
</tr>
<tr>
<td>\uxxxxx</td>
<td>The Unicode character specified by the hexadecimal number xxxxx; for example, \u0009 is the same as \t</td>
</tr>
</tbody>
</table>
The control character ^X; for example, \cJ is equivalent to the newline character \n
**Meta-characters**

A **meta-character** is simply an alphabetical character preceded by a backslash that acts to give the combination a special meaning.

For instance, you can search for a large sum of money using the '\d' meta-character: /([\d]+)000/. Here, \d will search for any string of the numerical character.

The following table lists a set of meta-characters which can be used in PERL Style Regular Expressions.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>A single character</td>
</tr>
<tr>
<td>\s</td>
<td>A whitespace character (space, tab, newline)</td>
</tr>
<tr>
<td>\S</td>
<td>Non-whitespace character</td>
</tr>
<tr>
<td>\d</td>
<td>A digit (0-9)</td>
</tr>
<tr>
<td>\D</td>
<td>A non-digit</td>
</tr>
<tr>
<td>\w</td>
<td>A word character (a-z, A-Z, 0-9, _)</td>
</tr>
<tr>
<td>\W</td>
<td>A non-word character</td>
</tr>
<tr>
<td>[\b]</td>
<td>A literal backspace (special case)</td>
</tr>
<tr>
<td>[aeiou]</td>
<td>Matches a single character in the given set</td>
</tr>
<tr>
<td>[^aeiou]</td>
<td>Matches a single character outside the given set</td>
</tr>
<tr>
<td>(foo</td>
<td>bar</td>
</tr>
</tbody>
</table>
### RegExp Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegExp.prototype.flags</td>
<td>A string that contains the flags of the RegExp object</td>
</tr>
<tr>
<td>RegExp.prototype.global</td>
<td>Whether to test the regular expression against all possible matches in a string, or only against the first</td>
</tr>
<tr>
<td>RegExp.prototype.ignoreCase</td>
<td>Whether to ignore case while attempting a match in a string</td>
</tr>
<tr>
<td>RegExp.prototype.multiline</td>
<td>Whether or not to search in strings across multiple lines</td>
</tr>
<tr>
<td>RegExp.prototype.source</td>
<td>The text of the pattern</td>
</tr>
<tr>
<td>RegExp.prototype.sticky</td>
<td>Whether or not the search is sticky</td>
</tr>
</tbody>
</table>

### RegExp Constructor

It returns a reference to the array function that created the instance's prototype.

**Syntax**

```javascript
RegExp.constructor
```

**Return Value**

Returns the function that created this object's instance.

**Example**

```javascript
var re = new RegExp("string");
console.log("re.constructor is:" + re.constructor);
```

**Output**

```
re.constructor is:function RegExp() { [native code] }
```
**global**

global is a read-only boolean property of RegExp objects. It specifies whether a particular regular expression performs global matching, i.e., whether it was created with the "g" attribute.

**Syntax**

```javascript
RegExpObject.global
```

**Return Value**

- Returns "TRUE" if the "g" modifier is set, "FALSE" otherwise.

**Example**

```javascript
var re = new RegExp( "string" );

if ( re.global )
{
    console.log("Test1 - Global property is set");
}
else
{
    console.log("Test1 - Global property is not set");
}
re = new RegExp( "string", "g" );

if ( re.global )
{
    console.log("Test2 - Global property is set");
}
else
{
    console.log("Test2 - Global property is not set");
}
```

**Output**

```
Test1 - Global property is not set
Test2 - Global property is set
```
**ignoreCase**

ignoreCase is a read-only boolean property of RegExp objects. It specifies whether a particular regular expression performs case-insensitive matching, i.e., whether it was created with the "i" attribute.

**Syntax**

```javascript
RegExpObject.ignoreCase
```

**Return Value**

- Returns "TRUE" if the "i" modifier is set, "FALSE" otherwise.

**Example**

```javascript
var re = new RegExp( "string" );

    if ( re.ignoreCase ){
        console.log("Test1-ignoreCase property is set");
    }
else
{
    console.log("Test1-ignoreCase property is not set");
}
re = new RegExp( "string", "i" );

    if ( re.ignoreCase ){
        console.log("Test2-ignoreCase property is set");
    }
else
{
    console.log("Test2-ignoreCase property is not set");
}
```

**Output**

```
Test1-ignoreCase property is not set
Test2-ignoreCase property is set
```
lastIndex

lastIndex is a read/write property of RegExp objects. For regular expressions with the "g" attribute set, it contains an integer that specifies the character position immediately following the last match found by the RegExp.exec() and RegExp.test() methods. These methods use this property as the starting point for the next search they conduct.

This property allows you to call those methods repeatedly, to loop through all matches in a string and works only if the "g" modifier is set.

This property is read/write, so you can set it at any time to specify where in the target string, the next search should begin. exec() and test() automatically reset the lastIndex to 0 when they fail to find a match (or another match).

Syntax

RegExpObject.lastIndex

Return Value

Returns an integer that specifies the character position immediately following the last match.

Example

```
var str = "Javascript is an interesting scripting language";
var re = new RegExp("script", "g");
re.test(str);
console.log("Test 1 - Current Index: ", + re.lastIndex);
re.test(str);
console.log("Test 2 - Current Index: ", + re.lastIndex)
```

Output

```
Test 1 - Current Index: 10
Test 2 - Current Index: 35
```

multiline

multiline is a read-only boolean property of RegExp objects. It specifies whether a particular regular expression performs multiline matching, i.e., whether it was created with the "m" attribute.

Syntax

RegExpObject.multiline
Return Value

- Returns "TRUE" if the "m" modifier is set, "FALSE" otherwise.

Example

```javascript
var re = new RegExp( "string" );
    if ( re.multiline ){
        console.log("Test1-multiline property is set");
    }
else
{
    console.log("Test1-multiline property is not set");
}
re = new RegExp( "string", "m" );
    if ( re.multiline ){
        console.log("Test2-multiline property is set");
    }
else
{
    console.log("Test2-multiline property is not set");
}
```

Output

```
Test1-multiline property is not set
Test2-multiline property is set
```

source

source is a read-only string property of RegExp objects. It contains the text of the RegExp pattern. This text does not include the delimiting slashes used in regular-expression literals, and it does not include the "g", "i", and "m" attributes.

Syntax

```
RegExpObject.source
```

Return Value

Returns the text used for pattern matching.
Example

```javascript
var str = "Javascript is an interesting scripting language";
var re = new RegExp( "script", "g" );
re.test(str);
console.log("The regular expression is : " + re.source);
```

Output

```
The regular expression is : script
```

RegExp Methods

<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegExp.prototype.exec()</td>
<td>Executes a search for a match in its string parameter</td>
</tr>
<tr>
<td>RegExp.prototype.test()</td>
<td>Tests for a match in its string parameter</td>
</tr>
<tr>
<td>RegExp.prototype.match()</td>
<td>Performs a match to the given string and returns the match result</td>
</tr>
<tr>
<td>RegExp.prototype.replace()</td>
<td>Replaces matches in the given string with a new substring</td>
</tr>
<tr>
<td>RegExp.prototype.search()</td>
<td>Searches the match in the given string and returns the index the pattern found in the string</td>
</tr>
<tr>
<td>RegExp.prototype.split()</td>
<td>Splits the given string into an array by separating the string into substring</td>
</tr>
<tr>
<td>RegExp.prototype.toString()</td>
<td>Returns a string representing the specified object. Overrides theObject.prototype.toString() method</td>
</tr>
</tbody>
</table>

exec()

The exec method searches string for text that matches regexp. If it finds a match, it returns an array of results; otherwise, it returns null.

Syntax

```javascript
RegExpObject.exec( string );
```
Parameter Details
- **string** – The string to be searched

Return Value
It returns the matched text if a match is found, and NULL if not.

Example
```javascript
var str = "Javascript is an interesting scripting language";
var re = new RegExp( "script", "g" );
var result = re.exec(str);
console.log("Test 1 - returned value : " + result);
re = new RegExp( "pushing", "g" );
var result = re.exec(str);
console.log("Test 2 - returned value : " + result)
```

**Output**
```
Test 1 - returned value : script
Test 2 - returned value : null
```

test()

The test method searches string for text that matches regexp. If it finds a match, it returns true; otherwise, it returns false.

Syntax
```
RegExpObject.test( string );
```

Parameter Details
- **string** – The string to be searched

Return Value
Returns the matched text if a match is found, and NULL if not.

Example
```javascript
var str = "Javascript is an interesting scripting language";
var re = new RegExp( "script", "g" );
var result = re.test(str);
```
```javascript
console.log("Test 1 - returned value : " + result);
re = new RegExp( "pushing", "g" );
var result = re.test(str);
console.log("Test 2 - returned value : " + result);
```

**Output**

Test 1 - returned value : true  
Test 2 - returned value : false

---

**match()**

This method retrieves the matches.

**Syntax**

```
str.match(regexp)
```

**Parameter Details**

- **RegExp**: A regular expression object.

**Return Value**

Returns an array of matches and null if no matches are found.

**Example**

```javascript
var str = 'Welcome to ES6.We are learning ES6';
var re = new RegExp("We");
var found = str.match(re);
console.log(found);
```

**Output**

We

---

**replace()**

This method returns a new string after replacing the matched pattern.

**Syntax**

```
str.replace(regex|substr, newSubStr|function)
```
Parameter Details
- **Regexp**: A regular expression object.
- **Substr**: String to be replaced
- **newSubStr**: The replacement string
- **function**: the function to create a new string

Return Value
A new string after replacing all matches.

Example
```javascript
var str = 'Good Morning';
var newstr = str.replace('Morning', 'Night');
console.log(newstr);
```

Output
Good Night

**search()**
This method returns the index where the match was found in the string. If no match is found, it returns -1.

Syntax
```javascript
str.replace(regexp|substr, newSubStr|function)
```

Parameter Details
- **Regexp**: A regular expression object.
- **Substr**: String to be replaced
- **newSubStr**: The replacement string
- **function**: the function to create a new string

Return Value
Returns the index at which the match was found in the string

Example
```javascript
var str = 'Welcome to ES6. We are learning ES6';
var re = new RegExp(/We/);
```
```javascript
var found = str.search(re);
console.log(found);
```

Output

\[0\]

**split()**

This method splits a string object on the basis of the separator specified and returns an array of strings.

**Syntax**

```
str.split([separator[, limit]])
```

**Parameter Details**

- **separator**: Optional. Specifies the separator character for the string
- **limit**: Optional. Specifies a limit on the number of splits to be found

**Return Value**

Returns the index at which the match was found in the string

**Example**

```javascript
var names = 'Monday;Tuesday;Wednesday';

console.log(names);

var re = /\s*;\s*/;
var nameList = names.split(re);

close.log(nameList));
```

Output

Monday;Tuesday;Wednesday
Monday,Tuesday,Wednesday
**toString()**

The `toString` method returns a string representation of a regular expression in the form of a regular-expression literal.

**Syntax**

```javascript
RegExpObject.toString();
```

**Return Value**

Returns the string representation of a regular expression.

**Example**

```javascript
var str = "Javascript is an interesting scripting language";
var re = new RegExp( "script", "g" );
var result = re.toString(str);
console.log("Test 1 - returned value : "+ result);
re = new RegExp( "/\", "g" );
var result = re.toString(str);
console.log("Test 2 - returned value : "+ result)
```

**Output**

```
Test 1 - returned value : /script/g
Test 2 - returned value : /\//g
```
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 the 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.

Following is a simple hierarchy of a few important objects:

![DOM Diagram](image)

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 the 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.
The Legacy DOM

This is the model which was introduced in the early versions of JavaScript language. It is well supported by all browsers, but allows access only to certain key portions of the 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 the document property values.

Document Properties in Legacy DOM

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

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Property and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>alinkColor</strong></td>
</tr>
<tr>
<td></td>
<td>Deprecated - A string that specifies the color of activated links.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong>: document.alinkColor</td>
</tr>
<tr>
<td>2</td>
<td><strong>anchors</strong>[]</td>
</tr>
<tr>
<td></td>
<td>An array of anchor objects, one for each anchor that appears in the document.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong>: document.anchors[0], document.anchors[1] and so on</td>
</tr>
<tr>
<td>3</td>
<td><strong>applets</strong>[]</td>
</tr>
<tr>
<td></td>
<td>An array of applet objects, one for each applet that appears in the document.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong>: document.applets[0], document.applets[1] and so on</td>
</tr>
<tr>
<td>4</td>
<td><strong>bgColor</strong></td>
</tr>
<tr>
<td></td>
<td>Deprecated - A string that specifies the background color of the document.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong>: document.bgColor</td>
</tr>
<tr>
<td>5</td>
<td><strong>Cookie</strong></td>
</tr>
<tr>
<td></td>
<td>A string valued property with special behavior that allows the cookies associated with this document to be queried and set.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong>: document.cookie</td>
</tr>
<tr>
<td>6</td>
<td><strong>Domain</strong></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>A string that specifies the Internet domain the document is from. Used for security purposes.</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong> document.domain</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7</th>
<th><strong>embeds[ ]</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>An array of objects that represent data embedded in the document with the <code>&lt;embed&gt;</code> tag. A synonym for plugins []. Some plugins and ActiveX controls can be controlled with JavaScript code.</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong> document.embeds[0], document.embeds[1] and so on</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
<th><strong>fgColor</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>A string that specifies the default text color for the document.</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong> document.fgColor</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9</th>
<th><strong>forms[ ]</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>An array of form objects, one for each HTML form that appears in the document.</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong> document.forms[0], document.forms[1] and so on</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10</th>
<th><strong>images[ ]</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>An array of image objects, one for each image that is embedded in the document with the HTML <code>&lt;img&gt;</code> tag.</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong> document.images[0], document.images[1] and so on</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>11</th>
<th><strong>lastModified</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>A read-only string that specifies the date of the most recent change to the document.</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong> document.lastModified</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12</th>
<th><strong>linkColor</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Deprecated - A string that specifies the color of unvisited links.</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong> document.linkColor</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>13</th>
<th><strong>links[ ]</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>It is a document link array.</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong> document.links[0], document.links[1] and so on</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>14</th>
<th><strong>Location</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>The URL of the document. Deprecated in favor of the URL property.</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong> document.location</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15</th>
<th><strong>plugins[ ]</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>A synonym for the embeds[ ]</td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong> document.plugins[0], document.plugins[1] and so on</td>
<td></td>
</tr>
</tbody>
</table>
### Document Methods in Legacy DOM

Following is a list of methods supported by Legacy DOM.

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

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.

**Example**

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

```html
<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">
</body>
</html>
```
<input type="button" value="Click Me" onclick="myFunc();" />
<input type="button" value="Cancel" />
</form>
<form name="SecondForm">
<input type="button" value="Don't Click Me" />
</form>
</body>
</html>

Output
The following output is displayed on successful execution of the above code.

This is main title
Click the following to see the result:

Click Me  Cancel  Don’t Click Me

Note: This example returns the objects for forms and elements. We would have to access their values by using those object properties which are not discussed in this tutorial.
ES6 introduces two new data structures: Maps and Sets.

- **Maps**: This data structure enables mapping a key to a value.
- **Sets**: Sets are similar to arrays. However, sets do not encourage duplicates.

### Maps

The Map object is a simple key/value pair. Keys and values in a map may be primitive or objects.

Following is the syntax for the same.

```javascript
new Map([iterable])
```

The parameter iterable represents any iterable object whose elements comprise of a key/value pair. Maps are ordered, i.e. they traverse the elements in the order of their insertion.

### Map Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map.prototype.size</td>
<td>This property returns the number of key/value pairs in the Map object.</td>
</tr>
</tbody>
</table>

### Syntax

```javascript
Map.size
```

### Example

```javascript
var myMap = new Map();
myMap.set("J", "john");
myMap.set("M", "mary");
myMap.set("T", "tom");

console.log(myMap.size);
```

### Output

3
Understanding basic Map operations

The `set()` function sets the value for the key in the Map object. The `set()` function takes two parameters namely, the key and its value. This function returns the Map object.

The `has()` function returns a boolean value indicating whether the specified key is found in the Map object. This function takes a key as parameter.

```javascript
var map = new Map();
map.set('name','Tutorial Point');
map.get('name'); // Tutorial point
```

The above example creates a map object. The map has only one element. The element key is denoted by `name`. The key is mapped to a value `Tutorial point`.

**Note:** Maps distinguish between similar values but bear different data types. In other words, an integer key 1 is considered different from a string key “1”. Consider the following example to better understand this concept.

```javascript
var map = new Map();
map.set(1,true);
console.log(map.has("1")); //false
map.set("1",true);
console.log(map.has("1")); //true
```

**Output**

```
false
true
```

The `set()` method is also chainable. Consider the following example.

```javascript
var roles=new Map();
roles.set('r1', 'User')
.set('r2', 'Guest')
.set('r3', 'Admin');
console.log(roles.has('r1'))
```

**Output**

```
True
```
The above example, defines a map object. The example chains the `set()` function to define the key/value pair.

The `get()` function is used to retrieve the value corresponding to the specified key.

The Map constructor can also be passed an array. Moreover, map also supports the use of spread operator to represent an array.

**Example**

```javascript
var roles = new Map([ ['r1', 'User'], ['r2', 'Guest'], ['r3', 'Admin'], ]); console.log(roles.get('r2'))
```

The following output is displayed on successful execution of the above code.

```
Guest
```

**Note:** The `get()` function returns undefined if the key specified doesn’t exist in the map.

The `set()` replaces the value for the key, if it already exists in the map. Consider the following example.

```javascript
var roles = new Map([ ['r1', 'User'], ['r2', 'Guest'], ['r3', 'Admin'], ]); console.log(`value of key r1 before set(): ${roles.get('r1')}`) roles.set('r1','superUser') console.log(`value of key r1 after set(): ${roles.get('r1')}`)
```

The following output is displayed on successful execution of the above code.

```
value of key r1 before set(): User
value of key r1 after set(): superUser
```
Map Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map.prototype.clear()</td>
<td>Removes all key/value pairs from the Map object.</td>
</tr>
<tr>
<td>Map.prototype.delete(key)</td>
<td>Removes any value associated to the key and returns the value that Map.prototype.has(key) would have previously returned. Map.prototype.has(key) will return false afterwards.</td>
</tr>
<tr>
<td>Map.prototype.entries()</td>
<td>Returns a new Iterator object that contains an array of [key, value] for each element in the Map object in insertion order.</td>
</tr>
<tr>
<td>Map.prototype.forEach(callbackFn[, thisArg])</td>
<td>Calls callbackFn once for each key-value pair present in the Map object, in insertion order. If a thisArg parameter is provided to forEach, it will be used as the 'this' value for each callback.</td>
</tr>
<tr>
<td>Map.prototype.keys()</td>
<td>Returns a new Iterator object that contains the keys for each element in the Map object in insertion order.</td>
</tr>
<tr>
<td>Map.prototype.values()</td>
<td>Returns a new Iterator object that contains an array of [key, value] for each element in the Map object in insertion order.</td>
</tr>
</tbody>
</table>

**clear()**

Removes all key/value pairs from the Map object.

**Syntax**

```javascript
myMap.clear();
```

**Parameters:** No parameters

**Return Value:** Undefined
Example

```javascript
var myMap = new Map();
myMap.set("bar", "baz");
console.log(myMap.size);
myMap.clear();
console.log(myMap.size)
```

Output

```
1
0
```

delete(key)

Removes any value associated to the key and returns the value that `Map.prototype.has(key)` would have previously returned. `Map.prototype.has(key)` will return `false` afterwards.

Syntax

```javascript
myMap.delete(key);
```

Parameters

- **Key**: key of the element to be removed from the Map

Return Value

Returns `true` if the element exited and was removed; else it returns `false`.

Example

```javascript
var myMap = new Map();
myMap.set("id", "admin");
myMap.set("pass", "admin@123");
console.log(myMap.has("id"));
myMap.delete("id");
console.log(myMap.has("id"));
```
entries()

Returns a new Iterator object that contains an array of [key, value] for each element in the Map object in insertion order.

Syntax

```javascript
myMap.entries()
```

Return Value

Returns a new iterator object.

Example

```javascript
var myMap = new Map();
myMap.set("id", "admin");
myMap.set("pass", "admin@123");
var itr = myMap.entries();
console.log(itr.next().value);
console.log(itr.next().value);
```

Output

```
id,admin
pass,admin@123
```

forEach

This function executes the specified function once per each Map entry.

Syntax

```javascript
myMap.forEach(callback[, thisArg])
```

Parameters

- `callback`: Function to execute for each element.
- `thisArg`: Value to use as this when executing callback.
Return Value

Undefined.

Example

```javascript
function userdetails(key, value) {
    console.log("m[" + key + "] = " + value);
}

var myMap = new Map();
myMap.set("id", "admin");
myMap.set("pass", "admin@123");
myMap.forEach(userdetails);
```

Output

```
m[admin] = id
m[admin@123] = pass
```

**keys()**

This function returns a new iterator object referring to the keys in the Map.

Syntax

```javascript
myMap.keys()
```

Return Value

Returns an Iterator object

Example

```javascript
var myMap = new Map();
myMap.set("id", "admin");
myMap.set("pass", "admin@123");
var itr=myMap.keys();
console.log(itr.next().value);
console.log(itr.next().value);
```
values()

This function returns a new iterator object referring to the values in the Map.

Syntax

```javascript
myMap.keys()
```

Return Value

Returns an Iterator object.

Example

```javascript
var myMap = new Map();
myMap.set("id", "admin");
myMap.set("pass", "admin@123");
var itr=myMap.values();
console.log(itr.next().value);
console.log(itr.next().value);
```

Output

```
Admin
admin@123
```

The for...of Loop

The following example illustrates traversing a map using the for...of loop.

```javascript
'use strict'
var roles = new Map([ ['r1', 'User'], ['r2', 'Guest'], ['r3', 'Admin'], ]); 
```
for(let r of roles.entries())
    console.log(`${r[0]}: ${r[1]}`);

The following output is displayed on successful execution of the above code.

r1: User
r2: Guest
r3: Admin

**Weak Maps**

A weak map is identical to a map with the following exceptions:

- Its keys must be objects.
- Keys in a weak map can be garbage collected. **Garbage collection** is a process of clearing the memory occupied by unreferenced objects in a program.
- A weak map cannot be iterated or cleared.

**Example: Weak Map**

```javascript
'use strict'
let weakMap = new WeakMap();
let obj ={};
console.log(weakMap.set(obj,"hello"));
console.log(weakMap.has(obj)); // true
```

The following output is displayed on successful execution of the above code.

WeakMap {}
true

**Sets**

A set is an ES6 data structure. It is similar to an array with an exception that it cannot contain duplicates. In other words, it lets you store unique values. Sets support both primitive values and object references.

Just like maps, sets are also ordered, i.e. elements are iterated in their insertion order. A set can be initialized using the following syntax.

```javascript
new Set([iterable]);
```
**Set Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Set.prototype.size</code></td>
<td>Returns the number of values in the Set object</td>
</tr>
</tbody>
</table>

**size()**

Returns the number of values in the Set object

**Syntax**

```
Myset.size
```

**Example**

```javascript
var mySet = new Set('tom','jim','jack');
var tot=mySet.size;
console.log(tot);
```

**Output**

3

**Set Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Set.prototype.add(value)</code></td>
<td>Appends a new element with the given value to the Set object. Returns the Set object.</td>
</tr>
<tr>
<td><code>Set.prototype.clear()</code></td>
<td>Removes all the elements from the Set object.</td>
</tr>
<tr>
<td><code>Set.prototype.delete(value)</code></td>
<td>Removes the element associated to the value.</td>
</tr>
<tr>
<td><code>Set.prototype.entries()</code></td>
<td>Returns a new Iterator object that contains an array of [value, value] for each element in the Set object, in insertion order. This is kept similar to the Map object, so that each entry has the same value for its key and value here.</td>
</tr>
</tbody>
</table>
Set.prototype.forEach(callbackFn[, thisArg])

Calls callbackFn once for each value present in the Set object, in insertion order. If thisArg parameter is provided to forEach, it will be used as the 'this' value for each callback.

Set.prototype.has(value)

Returns a boolean asserting whether an element is present with the given value in the Set object or not.

Set.prototype.values()

Returns a new Iterator object that contains the values for each element in the Set object in insertion order.

add()

This method appends a new element with the given value to the Set object.

Syntax

mySet.add(value);

Parameters

- Value: item to add to the list.

Return Value

Set Object.

Example

```javascript
var set = new Set();
set.add(10);
set.add(10); // duplicate not added
set.add(10); // duplicate not added
set.add(20);
set.add(30);
console.log(set.size);
```

The following output is displayed on successful execution of the above code.

3
clear()

This function clears all objects from the Set.

Syntax

mySet.clear();

Return Value

Undefined

Example

```javascript
var mySet = new Set('tom','jim','jack');
mySet.clear()
var tot=mySet.size;
console.log(tot);
```

The following output is displayed on successful execution of the above code.

0

delete()

This function removes the specified value from the Set.

Syntax

myMap.delete(key);

Parameters

- Key : key of the element to be removed from the Map

Return Value

Returns true if the element exited and was removed; else it returns false.

Example

```javascript
var set = new Set();
set.add(10);
set.add(20);
set.add(30);
```
```javascript
console.log(`Size of Set before delete() :${set.size}`);
console.log(`Set has 10 before delete() :${set.has(10)}`);
set.delete(10)
console.log(`Size of Set after delete() :${set.size}`);
console.log(`Set has 10 after delete() :${set.has(10)}`);
```

**Output**

Size of Set before delete() :3
Set has 10 before delete() :true
Size of Set after delete() :2
Set has 10 after delete() :false

**entries()**

Returns a new Iterator object that contains an array of [value,value] for each element in the Set.

**Syntax**

```
mySet.entries()
```

**Return Value**

Returns a new iterator object.

**Example**

```javascript
var mySet = new Set();
mySet.add("Jim");
mySet.add("Jack");
mySet.add("Jane");
var itr=mySet.entries()
console.log(itr.next().value);
console.log(itr.next().value);
console.log(itr.next().value);
```
Output

```
Jim, Jim
Jack, Jack
Jane, Jane
```

**forEach**

This function executes the specified function once per each Map entry.

**Syntax**

```
myMap.forEach(callback[, thisArg])
```

**Parameters**

- **callback**: Function to execute for each element.
- **thisArg**: Value to use as this when executing callback.

**Return Value**

Undefined

**Example**

```javascript
function userdetails(values) {
    console.log(values);
}

var mySet = new Set();
mySet.add("John");
mySet.add("Jane");
mySet.forEach(userdetails);
```

Output

```
John
Jane
```

**has()**

Returns a boolean indicating whether an element with the specified value exists in a Set object or not.
Syntax

```javascript
mySet.has(value);
```

Parameters
- **Value**: The value for which existence check should be done.

Return Value
Returns true if an element with the specified value exists in the Set object; otherwise false.

Example

```javascript
var mySet = new Set();
mySet.add("Jim");

console.log(mySet.has("Jim"));
console.log(mySet.has("Tom"));
```

Output

```
true
false
```

values() and keys()
The values method returns a new Iterator object that contains the values for each element in the Set object. The keys() function too behaves in the same fashion.

Syntax

```javascript
mySet.values();
mySet.keys();
```

Return Value
A new Iterator object containing the values for each element in the given Set.
mySet.add("Jack");
mySet.add("Jane");
console.log("Printing keys()------------------");

var keyitr=mySet.keys();
console.log(keyitr.next().value);
console.log(keyitr.next().value);
console.log(keyitr.next().value);

console.log("Printing values()------------------");
var valitr=mySet.values();
console.log(valitr.next().value);
console.log(valitr.next().value);
console.log(valitr.next().value);

Output

Printing keys()------------------
Jim
Jack
Jane
Printing values()------------------
Jim
Jack
Jane

Example: Iterating a Set

'use strict'
let set = new Set();
set.add('x');
set.add('y');
set.add('z');
for(let val of set){
  console.log(val);
}

The following output is displayed on successful execution of the above code.
**Weak Set**

Weak sets can only contain objects, and the objects they contain may be garbage collected. Like weak maps, weak sets cannot be iterated.

**Example: Using a Weak Set**

```javascript
'use strict'
let weakSet = new WeakSet();
let obj = { msg: 'hello' };
weakSet.add(obj);
console.log(weakSet.has(obj));
weakSet.delete(obj);
console.log(weakSet.has(obj));
```

The following output is displayed on successful execution of the above code.

```
true
false
```

**Iterator**

Iterator is an object which allows to access a collection of objects one at a time. Both set and map have methods which returns an iterator.

Iterators are objects with `next()` method. When `next()` method is invoked, it returns an object with `value` and `done` properties. 'done' is boolean, this will return true after reading all items in the collection.

**Example 1: Set and Iterator**

```javascript
var set = new Set(['a', 'b', 'c', 'd', 'e']);
var iterator = set.entries();
console.log(iterator.next());
```

The following output is displayed on successful execution of the above code.

```
{ value: [ 'a', 'a' ], done: false }
```
Since, the set does not store key/value, the value array contains similar key and value. done will be false as there are more elements to be read.

**Example 2: Set and Iterator**

```javascript
var set = new Set(['a','b','c','d','e']);
var iterator = set.values();
console.log(iterator.next());
```

The following output is displayed on successful execution of the above code.

```
{ value: 'a', done: false }
```

**Example 3: Set and Iterator**

```javascript
var set = new Set(['a','b','c','d','e']);
var iterator = set.keys();
console.log(iterator.next());
```

The following output is displayed on successful execution of the above code.

```
{ value: 'a', done: false }
```

**Example 4: Map and Iterator**

```javascript
var map = new Map([[1,'one'],[2,'two'],[3,'three']]);
var iterator = map.entries();
console.log(iterator.next());
```

The following output is displayed on successful execution of the above code.

```
{ value: [ 1, 'one' ], done: false }
```

**Example 5: Map and Iterator**

```javascript
var map = new Map([[1,'one'],[2,'two'],[3,'three']]);
var iterator = map.values();
console.log(iterator.next());
```

The following output is displayed on successful execution of the above code.

```
{value: "one", done: false}
```

**Example 6: Map and Iterator**
var map = new Map([[1,'one'],[2,'two'],[3,'three']]);
var iterator = map.keys();
console.log(iterator.next());

The following output is displayed on successful execution of the above code.

{value: 1, done: false}
Object Orientation is a software development paradigm that follows real-world modelling. Object Orientation, considers a program as a collection of objects that communicates with each other via mechanism called **methods**. ES6 supports these object-oriented components too.

### Object-Oriented Programming Concepts

To begin with, let us understand

- **Object**: An object is a real-time representation of any entity. According to Grady Brooch, every object is said to have 3 features:
  - **State**: Described by the attributes of an object.
  - **Behavior**: Describes how the object will act.
  - **Identity**: A unique value that distinguishes an object from a set of similar such objects.

- **Class**: A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object.

- **Method**: Methods facilitate communication between objects.

Let us translate these Object-Oriented concepts to the ones in the real world. For example: A car is an object that has data (make, model, number of doors, Vehicle Number, etc.) and functionality (accelerate, shift, open doors, turn on headlights, etc.)

Prior to ES6, creating a class was a fussy affair. Classes can be created using the **class** keyword in ES6.

Classes can be included in the code either by declaring them or by using class expressions.

### Syntax: Declaring a Class

```javascript
class Class_name
{
}
```
Syntax: Class Expressions

```javascript
var var_name=new  Class_name
{
}
```

The class keyword is followed by the class name. The rules for identifiers (already discussed) must be considered while naming a class.

A class definition can include the following:

- **Constructors**: Responsible for allocating memory for the objects of the class.

- **Functions**: Functions represent actions an object can take. They are also at times referred to as methods.

These components put together are termed as the data members of the class.

**Note**: A class body can only contain methods, but not data properties.

**Example: Declaring a class**

```javascript
class Polygon {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}
```

**Example: Class Expression**

```javascript
var Polygon = class {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
};
```
The above code snippet represents an unnamed class expression. A named class expression can be written as:

```javascript
var Polygon = class Polygon {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
};
```

**Note**: Unlike variables and functions, classes cannot be hoisted.

### Creating Objects

To create an instance of the class, use the `new` keyword followed by the class name. Following is the syntax for the same.

```javascript
var object_name = new class_name([ arguments ])
```

Where,

- The `new` keyword is responsible for instantiation.
- The right hand side of the expression invokes the constructor. The constructor should be passed values if it is parameterized.

**Example: Instantiating a class**

```javascript
var obj = new Polygon(10, 12)
```

### Accessing Functions

A class’s attributes and functions can be accessed through the object. Use the `.` **dot notation** (called as the period) to access the data members of a class.

```javascript
//accessing a function
obj.function_name()
```

**Example: Putting them together**

```javascript
'use strict'
class Polygon {
    constructor(height, width) {
        this.h = height;
        this.w = width;
    }
};
```
The Example given above declares a class 'Polygon'. The class’s constructor takes two arguments - height and width respectively. The ‘this’ keyword refers to the current instance of the class. In other words, the constructor above initializes two variables h and w with the parameter values passed to the constructor. The test () function in the class, prints the values of the height and width.

To make the script functional, an object of the class Polygon is created. The object is referred to by the polyObj variable. The function is then called via this object.

The following output is displayed on successful execution of the above code.

```
The height of the polygon: 10
The width of the polygon: 20
```

### The Static Keyword

The static keyword can be applied to functions in a class. Static members are referenced by the class name.

**Example**

```
'use strict'
class StaticMem
{
    static disp()
    {
        console.log("Static Function called")
    }
}
StaticMem.disp() //invoke the static method
```
Note: It is not mandatory to include a constructor definition. Every class by default has a constructor by default.

The following output is displayed on successful execution of the above code.

Static Function called

The instanceof operator

The instanceof operator returns true if the object belongs to the specified type.

Example

```
'use strict'
class Person{ }
var obj=new Person()
var isPerson=obj instanceof Person;
console.log(" obj is an instance of Person " + isPerson);
```

The following output is displayed on successful execution of the above code.

obj is an instance of Person True

Class Inheritance

ES6 supports the concept of Inheritance. Inheritance is the ability of a program to create new entities from an existing entity - here a class. The class that is extended to create newer classes is called the parent class/super class. The newly created classes are called the child/sub classes.

A class inherits from another class using the 'extends' keyword. Child classes inherit all properties and methods except constructors from the parent class.

Following is the syntax for the same.

```
class child_class_name extends parent_class_name
```

Example: Class Inheritance

```
'use strict'
class Shape
{
    constructor(a)
    {
        this.Area=a
    }
```
class Circle extends Shape
{
    disp()
    { console.log("Area of the circle: "+this.Area) }
}

var obj=new Circle(223);
obj.disp()

The above example declares a class Shape. The class is extended by the Circle class. Since, there is an inheritance relationship between the classes, the child class i.e., the class Car gets an implicit access to its parent class attribute i.e., area.

The following output is displayed on successful execution of the above code.

Area of Circle: 223

Inheritance can be classified as:

- **Single**: Every class can at the most extend from one parent class
- **Multiple**: A class can inherit from multiple classes. ES6 doesn’t support multiple inheritance.
- **Multi-level**: Consider the following example.

```javascript
'use strict'
class Root
{
    test()
    {
        console.log("call from parent class")
    }
}
class Child extends Root
{}
class Leaf extends Child
//indirectly inherits from Root by virtue of inheritance
{}
var obj= new Leaf();
```
The class Leaf derives the attributes from the Root and the Child classes by virtue of multi-level inheritance.

The following output is displayed on successful execution of the above code.

call from parent class

Class Inheritance and Method Overriding

Method Overriding is a mechanism by which the child class redefines the superclass method. The following example illustrates the same:

```
'use strict'

class PrinterClass{
    doPrint()
    {
        console.log("doPrint() from Parent called...
    }
}

class StringPrinter extends PrinterClass{
    doPrint()
    {
        console.log("doPrint() is printing a string...")
    }

var obj= new StringPrinter()
obj.doPrint()
```

In the above Example, the child class has changed the superclass function’s implementation.

The following output is displayed on successful execution of the above code.

doPrint() is printing a string...

The Super Keyword

ES6 enables a child class to invoke its parent class data member. This is achieved by using the super keyword. The super keyword is used to refer to the immediate parent of a class.
Consider the following example.

```javascript
'use strict'

class PrinterClass
{
    doPrint()
    {
        console.log("doPrint() from Parent called...")
    }
}

class StringPrinter extends PrinterClass
{
    doPrint()
    {
        super.doPrint()
        console.log("doPrint() is printing a string...")
    }
}

var obj = new StringPrinter()
obj.doPrint()
```

The `doPrint()` redefinition in the class `StringPrinter`, issues a call to its parent class version. In other words, the `super` keyword is used to invoke the `doPrint()` function definition in the parent class - `PrinterClass`.

The following output is displayed on successful execution of the above code.

```
doPrint() from Parent called.
doPrint() is printing a string.
```
Promises are a clean way to implement async programming in JavaScript (ES6 new feature). Prior to promises, Callbacks were used to implement async programming. Let’s begin by understanding what async programming is and its implementation, using Callbacks.

Understanding Callback

A function may be passed as a parameter to another function. This mechanism is termed as a Callback. A Callback would be helpful in events.

The following example will help us better understand this concept.

```javascript
<script>
function notifyAll(fnSms, fnEmail)
{
    console.log('starting notification process');
    fnSms();
    fnEmail();
}
notifyAll(function()
{
    console.log("Sms send ..");
}, function()
{
    console.log("email send ..");
});
console.log("End of script"); //executes last or blocked by other methods
</script>
```

In the `notifyAll()` method shown above, the notification happens by sending SMS and by sending an e-mail. Hence, the invoker of the notifyAll method has to pass two functions as parameters. Each function takes up a single responsibility like sending SMS and sending an e-mail.
The following output is displayed on successful execution of the above code.

```
starting notification process
Sms send ..
Email send ..
End of script
```

In the code mentioned above, the function calls are synchronous. It means the UI thread would be waiting to complete the entire notification process. Synchronous calls become blocking calls. Let's understand non-blocking or async calls now.

### Understanding AsyncCallback

Consider the above example.

To enable the script, execute an asynchronous or a non-blocking call to notifyAll() method. We shall use the `setTimeout()` method of JavaScript. This method is async by default.

The setTimeout() method takes two parameters:

- A callback function
- The number of seconds after which the method will be called

In this case, the notification process has been wrapped with timeout. Hence, it will take a two seconds delay, set by the code. The notifyAll() will be invoked and the main thread goes ahead like executing other methods. Hence, the notification process will not block the main JavaScript thread.

```javascript
<script>
function notifyAll(fnSms, fnEmail)
{
    setTimeout(function()
    {
        console.log('starting notification process');
        fnSms();
        fnEmail();
    }, 2000);
}
notifyAll(function()
{
    console.log("Sms send ..");
}, function()
```
```js
{
    console.log("email send ..");
});
console.log("End of script"); //executes first or not blocked by others
</script>

The following output is displayed on successful execution of the above code.

End of script
starting notification process
Sms send ..
Email send ..

In case of multiple callbacks, the code will look scary.

```js
<script>
    setTimeout(function()
    {
        console.log("one");
        setTimeout(function()
        {
            console.log("two");
            setTimeout(function()
            {
                console.log("three");
            }, 1000);
            }, 1000);
        }, 1000);
</script>
```

ES6 comes to your rescue by introducing the concept of promises. Promises are "Continuation events" and they help you execute the multiple async operations together in a much cleaner code style.

**Example**
Let's understand this with an example. Following is the syntax for the same.

```javascript
var promise = new Promise(function(resolve, reject) {
    // do a thing, possibly async, then..
    if(/*everthing turned out fine */) resolve("stuff worked");
    else reject(Error("It broke"));
});
return promise; // Give this to someone
```

The first step towards implementing the promises is to create a method which will use the promise. Let’s say in this example, the `getSum()` method is asynchronous i.e., its operation should not block other methods’ execution. As soon as this operation completes, it will later notify the caller.

The following example (Step 1) declares a Promise object ‘var promise’. The Promise Constructor takes to the functions first for the successful completion of the work and another in case an error happens.

The promise returns the result of the calculation by using the resolve callback and passing in the result, i.e., n1+n2.

**Step 1:** resolve(n1 + n2);

If the getSum() encounters an error or an unexpected condition, it will invoke the reject callback method in the Promise and pass the error information to the caller.

**Step 2:** reject(Error("Negatives not supported"));

The method implementation is given in the following code (STEP 1).

```javascript
function getSum(n1, n2) {
    var isAnyNegative = function() {
        return n1 < 0 || n2 < 0;
    }
    var promise = new Promise(function(resolve, reject) {
        if (isAnyNegative()) {
            reject(Error("Negatives not supported"));
        }
        resolve(n1 + n2);
    });
    return promise;
}
```
ES6

263

return promise;
}

The second step details the implementation of the caller (STEP 2).

The caller should use the 'then' method, which takes two callback methods - first for success and second for failure. Each method takes one parameter, as shown in the following code.

```javascript
getSum(5, 6)
  .then(function (result) {
    console.log(result);
  },
  function (error) {
    console.log(error);
  });
```

The following output is displayed on successful execution of the above code.

```
11
```

Since the return type of the getSum() is a Promise, we can actually have multiple 'then' statements. The first 'then' will have a return statement.

```javascript
getSum(5, 6)
  .then(function(result)
    {
      console.log(result);
      return getSum(10, 20); // this returns another promise
    },
    function(error)
    {
      console.log(error);
    })
  .then(function(result)
    {
      console.log(result);
    }, function(error)
    {
      console.log(error);
    });
```
The following output is displayed on successful execution of the above code.

```
11
30
```

The following example issues three then() calls with getSum() method.

```javascript
function getSum(n1, n2)
{
    var isAnyNegative = function()
    {
        return n1 < 0 || n2 < 0;
    }
    var promise = new Promise(function(resolve, reject)
    {
        if (isAnyNegative())
        {
            reject(Error("Negatives not supported"));
        }
        resolve(n1 + n2);
    });
    return promise;
}
getSum(5, 6)
    .then(function(result)
    {
        console.log(result);
        return getSum(10, 20); //this returns another Promise
    },
    function(error)
    {
        console.log(error);
    })
```
The following output is displayed on successful execution of the above code.

The program displays ‘end of script’ first and then results from calling getSum() method, one by one.

End of script
11
30
70

This shows getSum() is called in async style or non-blocking style. Promise gives a nice and clean way to deal with the Callbacks.
Consider a scenario where parts of JavaScript code need to be reused. ES6 comes to your rescue with the concept of Modules.

A **module** is nothing more than a chunk of JavaScript code written in a file. The functions or variables in a module are not available for use, unless the module file exports them.

In simpler terms, the modules help you to write the code in your module and expose only those parts of the code that should be accessed by other parts of your code.

ES6 modules will have to be transpiled to ES5 code so that we can run and test the code. **Transpilation** is the process of converting code from one language into its counterpart equivalent. The ES6 Module Transpiler is a tool that takes your ES6 module and compiles it into ES5 compatible code in the CommonJS or AMD style.

ES6 modules is a very powerful concept. Although support is not available everywhere yet, you can play with ES6 code today and transpile into ES5. You can use Grunt, Gulp, Babel or some other transpiler to compile the modules during a build process. For the purpose of demonstration, the lesson uses Node.js to transpile and execute the code as it is console based and easy to understand.

### Exporting a Module

To make available certain parts of the module, use the export keyword. Following is the syntax to export a module.

**Export a single value or element - Use export default**

```javascript
export default element_name
```

**Export multiple values or elements**

```javascript
export {element_name1, element_name2, ...}
```

### Importing a Module

To be able to consume a module, use the import keyword. Following is the syntax for the same.

**Import a single value or element**

```javascript
import element name from module_name
```
Export multiple values or elements

```javascript
import {element_name1, element_name2, ....} from module_name
```

Consider a JavaScript file, Message.js, with a method printMsg() in it. To be able to reuse the functionality provided by this method, include the following in the Message.js file:

```javascript
export default printMsg
```

The script file that intends to consume the function, say User.js, will have to import the function from the Message module by including the following:

```javascript
import printMsg from './Message.js'
```

**Note:** Multiple elements in the export statement should be delimited by a comma separator. The same holds true for the import.

**Example: Defining and Using ES6 modules**

**Defining a module: Message_module.js**

```javascript
function display_message()
{
    console.log("Hello World")
}
export default display_message
```

**Importing the module: consume_module.js**

```javascript
import display_message from './MessageModule.js'
display_message()
```

Install the es6-module-transpiler via `npm` using the following command:

```shell
npm install -g es6-module-transpiler
```

Assume the directory structure of the given JS project as below:

```plaintext
D:/
ES6/
    scripts/
        app.js
        utility.js
    out/
```
where, **scripts** is the directory containing my ES6 code samples. We shall transpile the ES6 code into ES5 and save them to the directory shown above.

Following are the steps for the same:

**Step 1:** Navigate to the D:/ ES6/scripts directory and transpile the ES6 code into **CommonJS** format. You may also choose to transpile into the **AMD** Format and then use a browser to run the same.

Type the following in the node window to transpile the code into CommonJS format:

```
compile-modules convert -I scripts -o out Message_module.js consume_module.js -format commonjs.
```

The above command will transpile all JS files in the script directory and place their transpiled versions into the out subdirectory.

**Step 2:** Execute the script code.

```
cd out
node consume_module.js
```

Following will be the output of the above code.

```
Hello World
```

**Note:** A module can also be re-exported, i.e. the code that imports a module can also export it.
There are three types of errors in programming: Syntax Errors, Runtime Errors, and Logical Errors.

**Syntax Errors**

Syntax errors, also called parsing errors, occur at compile time in traditional programming languages and at interpret time in JavaScript. When a syntax error occurs in JavaScript, only the code contained within the same thread as the syntax error is affected and the rest of the code in other threads get executed assuming nothing in them depends on the code containing the error.

**Runtime Errors**

Runtime errors, also called exceptions, occur during execution (after compilation/interpretation). Exceptions also affect the thread in which they occur, allowing other JavaScript threads to continue normal execution.

**Logical Errors**

Logic errors can be the most difficult type of errors to track down. These errors are not the result of a syntax or runtime error. Instead, they occur when you make a mistake in the logic that drives your script and you do not get the result as expected.

You cannot catch those errors, because it depends on your business requirement, what type of logic you want to put in your program.

JavaScript throws instances of the Error object when runtime errors occur. The following table lists predefined types of the Error object.

<table>
<thead>
<tr>
<th>Error Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EvalError</code></td>
<td>Creates an instance representing an error that occurs regarding the global function <code>eval()</code></td>
</tr>
<tr>
<td><code>RangeError</code></td>
<td>Creates an instance representing an error that occurs when a numeric variable or parameter is outside of its valid range</td>
</tr>
<tr>
<td><code>ReferenceError</code></td>
<td>Creates an instance representing an error that occurs when de-referencing an invalid reference</td>
</tr>
</tbody>
</table>
## Throwing Exceptions

An error (predefined or user defined) can be raised using the `throw` statement. Later these exceptions can be captured and you can take an appropriate action. Following is the syntax for the same.

### Syntax: Throwing a generic exception

```javascript
throw new Error([message])
OR
throw([message])
```

### Syntax: Throwing a specific exception

```javascript
throw new Error_name([message])
```

## Exception Handling

Exception handling is accomplished with a `try...catch` statement. When the program encounters an exception, the program will terminate in an unfriendly fashion. To safeguard against this unanticipated error, we can wrap our code in a try...catch statement.

The try block must be followed by either exactly one catch block or one finally block (or one of both). When an exception occurs in the try block, the exception is placed in `e` and the catch block is executed. The optional finally block executes unconditionally after try/catch.

Following is the syntax for the same.

```javascript
try {
    // Code to run
    [break;]
} catch (e) {
    // Code to run if an exception occurs
```
Example

```javascript
var a = 100;
var b = 0;
try {
    if (b == 0 ) {
        throw("Divide by zero error.");
    }
    else {
        var c = a / b;
    }
} catch( e ) {
    console.log("Error: " + e );
}
```

Output

The following output is displayed on successful execution of the above code.

Error: Divide by zero error.

**Note:** You can raise an exception in one function and then you can capture that exception either in the same function or in the caller function using a **try...catch** block.
The onerror() Method

The onerror event handler was the first feature to facilitate error handling in JavaScript. The error event is fired on the window object whenever an exception occurs on the page.

Example

```html
<html>
<head>
<script type="text/javascript">
    window.onerror = function () {
        document.write("An error occurred.");
    }
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

Output

The following output is displayed on successful execution of the above code.

Click the following to see the result:

Click Me

The onerror event handler provides three pieces of information to identify the exact nature of the error:

- **Error message**: The same message that the browser would display for the given error.
- **URL**: The file in which the error occurred.
- **Line number**: The line number in the given URL that caused the error.

The following example shows how to extract this information.
Custom Errors

JavaScript supports the concept of custom errors. The following example explains the same.

Example 1: Custom Error with default message

```javascript
function MyError(message) {
    this.name = 'CustomError';
    this.message = message || 'Error raised with default message';
}

try {
    throw new MyError();
} catch (e) {

    console.log(e.name);
    console.log(e.message);  // 'Default Message'
}
```
The following output is displayed on successful execution of the above code.

```javascript
CustomError
Error raised with default message
```

**Example 2: Custom Error with user-defined error message**

```javascript
function MyError(message) {
  this.name = 'CustomError';
  this.message = message || 'Default Error Message';
}

try {
  throw new MyError('Printing Custom Error message');
} catch (e) {
  console.log(e.name);
  console.log(e.message);
}
```

The following output is displayed on successful execution of the above code.

```javascript
CustomError
Printing Custom Error message
```
Form validation normally used to occur at the server, after the client had entered all the necessary data and then pressed the Submit button. If the data entered by the client was incorrect or was simply missing, the server would have to send all the data back to the client and request that the form be resubmitted with the correct information. This was really a lengthy process which used to put a lot of burden on the server.

JavaScript provides a way to validate the form's data on the client's computer before sending it to the web server. Form validation generally performs two functions.

- **Basic Validation**: First of all, the form must be checked to make sure all the mandatory fields are filled in. It would require just a loop through each field in the form and check for data.

- **Data Format Validation**: Secondly, the data that is entered must be checked for correct form and value. Your code must include appropriate logic to test the correctness of data.

**Example**

We will take an example to understand the process of validation. Here is a simple form in html format.

```html
<html>
<head>
<title>Form Validation</title>
<script type="text/javascript">
<!--
// Form validation code will come here.
//-->
</script>
</head>
<body>
<form action="/cgi-bin/test.cgi" name="myForm" onsubmit="return(validate());">
<table cellspacing="2" cellpadding="2" border="1">
<tr>
<td align="right">Name</td>
<td><input type="text" name="Name" /></td>
</tr>
</table>
</form>
</body>
</html>
```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EMail</td>
<td></td>
</tr>
<tr>
<td>Zip Code</td>
<td></td>
</tr>
<tr>
<td>Country</td>
<td></td>
</tr>
</tbody>
</table>

<form>
  <tr>
    <td align="right">EMail</td>
    <td><input type="text" name="EMail" /></td>
  </tr>
  <tr>
    <td align="right">Zip Code</td>
    <td><input type="text" name="Zip" /></td>
  </tr>
  <tr>
    <td align="right">Country</td>
    <td>
      <select name="Country">
        <option value="-1" selected>[choose yours]</option>
        <option value="1">USA</option>
        <option value="2">UK</option>
        <option value="3">INDIA</option>
      </select>
    </td>
  </tr>
  <tr>
    <td align="right"></td>
    <td><input type="submit" value="Submit" /></td>
  </tr>
</form>
Output

The following output is displayed on successful execution of the above code.

Basic Form Validation

First let us see how to do a basic form validation. In the above form, we are calling `validate()` to validate data when `onsubmit` event is occurring. The following code shows the implementation of this `validate()` function.

```javascript
<!--
// Form validation code will come here. function validate()
{
    if( document.myForm.Name.value == "" )
    {
        alert( "Please provide your name!" );
        document.myForm.Name.focus();
        return false;
    }
    if( document.myForm.EMail.value == "" )
    {
        alert( "Please provide your Email!" );
        document.myForm.EMail.focus();
        return false;
    }
    if( document.myForm.Zip.value == "" ||
        isNaN( document.myForm.Zip.value ) ||
        document.myForm.Zip.value.length != 5 )
```
Data Format Validation

Now we will see how we can validate our entered form data before submitting it to the web server.

The following example shows how to validate an entered email address. An email address must contain at least a '@' sign and a dot (.). Also, the '@' must not be the first character of the email address, and the last dot must at least be one character after the '@' sign.

Example

Try the following code for email validation.

```javascript
<script type="text/javascript">
<!--
function validateEmail()
{
  var emailID = document.myForm.EMail.value;    atpos = emailID.indexOf("@");
  dotpos = emailID.lastIndexOf(".");    if (atpos < 1 || ( dotpos - atpos < 2 ))
  {
    alert("Please enter correct email ID")
    document.myForm.EMail.focus();
    return false;
  }
  return( true );
}
//-->
</script>
```
You can use JavaScript to create a complex animation having, but not limited to, the following elements:

- Fireworks
- Fade effect
- Roll-in or Roll-out
- Page-in or Page-out
- Object movements

In this chapter, we will see how to use JavaScript to create an animation.

JavaScript can be used to move a number of DOM elements (<img />, <div>, or any other HTML element) around the page according to some sort of pattern determined by a logical equation or function.

JavaScript provides the following functions to be frequently used in animation programs.

- `setTimeout` (function, duration) - This function calls the function after duration milliseconds from now.
- `setInterval` (function, duration) - This function calls the function after every duration milliseconds.
- `clearTimeout` (setTimeout_variable) - This function clears any timer set by the `setTimeout()` function.

JavaScript can also set a number of attributes of a DOM object including its position on the screen. You can set the top and the left attribute of an object to position it anywhere on the screen. Following is the syntax for the same.

```javascript
// Set distance from left edge of the screen.
object.style.left = distance in pixels or points;

or

// Set distance from top edge of the screen.
object.style.top = distance in pixels or points;
```
Manual Animation

So let's implement one simple animation using DOM object properties and JavaScript functions as follows. The following list contains different DOM methods.

- We are using the JavaScript function `getElementById()` to get a DOM object and then assigning it to a global variable `imgObj`.
- We have defined an initialization function `init()` to initialize `imgObj` where we have set its position and left attributes.
- We are calling initialization function at the time of window load.
- We are calling `moveRight()` function to increase the left distance by 10 pixels. You could also set it to a negative value to move it to the left side.

Example

Try the following example.

```html
<html>
<head>
<title>JavaScript Animation</title>
<script type="text/javascript">
<!--
var imgObj = null; function init(){
    imgObj = document.getElementById('myImage');    imgObj.style.position= 'relative';     imgObj.style.left = '0px';
}
function moveRight(){
    imgObj.style.left = parseInt(imgObj.style.left) + 10 + 'px';
}
window.onload =init;
//-->
</script>
</head>
<body>
<form>
<img id="myImage" src="/images/html.gif" />
<p>Click button below to move the image to right</p>
<input type="button" value="Click Me" onclick="moveRight();" />
</form></body></html>

It is not possible to show the result, i.e., the animation in this tutorial.
Automated Animation

In the above example, we saw how an image moves to the right with every click. We can automate this process by using the JavaScript function `setTimeout()` as follows.

Here we have added more methods. So, let's see what is new here.

- The `moveRight()` function is calling `setTimeout()` function to set the position of `imgObj`.
- We have added a new function `stop()` to clear the timer set by `setTimeout()` function and to set the object at its initial position.

Example

Try the following example code.

```html
<html>
<head>
<title>JavaScript Animation</title>
<script type="text/javascript">
<!--
var imgObj = null; var animate ; function init(){
    imgObj = document.getElementById('myImage');    imgObj.style.position= 'relative';    imgObj.style.left = '0px';
}
function moveRight(){
    imgObj.style.left = parseInt(imgObj.style.left) + 10 + 'px';    animate = setTimeout(moveRight,20); // call moveRight in 20msec
}
function stop(){    clearTimeout(animate);    imgObj.style.left = '0px';
}
window.onload =init;
//-->
</script>
</head>
<body>
<form>
<img id="myImage" src="/images/html.gif" />
<p>Click the buttons below to handle animation</p>
<input type="button" value="Start" onclick="moveRight();" />
<input type="button" value="Stop" onclick="stop();" />
</form>
```
Rollover with a Mouse Event

Here is a simple example showing the image rollover with a mouse event.

Let's see what we are using in the following example:

- At the time of loading this page, the `if` statement checks for the existence of the image object. If the image object is unavailable, this block will not be executed.

- The `Image()` constructor creates and preloads a new image object called `image1`.

- The `src` property is assigned the name of the external image file called `/images/html.gif`.

- Similarly, we have created `image2` object and assigned `/images/http.gif` in this object.

- The `#` (hash mark) disables the link so that the browser does not try to go to a URL when clicked. This link is an image.

- The `onMouseOver` event handler is triggered when the user's mouse moves onto the link, and the `onMouseOut` event handler is triggered when the user's mouse moves away from the link (image).

- When the mouse moves over the image, the HTTP image changes from the first image to the second one. When the mouse is moved away from the image, the original image is displayed.

- When the mouse is moved away from the link, the initial image `html.gif` will reappear on the screen.

```javascript
if(document.images){
    var image1 = new Image();
    // Preload an image image1.src = "/images/html.gif";
    var image2 = new Image();
    // Preload second image image2.src = "/images/http.gif";
}
//-->
```
<script>
</script>
</head>
<body>
<p>Move your mouse over the image to see the result</p>
<a href="#" onMouseOver="document.myImage.src=image2.src;"
   onMouseOut="document.myImage.src=image1.src;">
<img name="myImage" src="/images/html.gif" />
</a>
</body>
</html>
The JavaScript navigator object includes a child object called **plugins**. This object is an array, with one entry for each plug-in installed on the browser. The `navigator.plugins` object is supported only by Netscape, Firefox, and Mozilla.

**Example**

The following example shows how to list down all the plug-ins installed in your browser.

```html
<html>
<head>
<title>List of Plug-Ins</title>
</head>
<body>
<table border="1">
<tr><th>Plug-in Name</th><th>Filename</th><th>Description</th></tr>
<script LANGUAGE="JavaScript" type="text/javascript"> for (i=0; i<navigator.plugins.length; i++) {    document.write("<tr><td>");    document.write(navigator.plugins[i].name);    document.write("</td><td>");    document.write(navigator.plugins[i].filename);    document.write("</td><td>");    document.write(navigator.plugins[i].description);    document.write("</td></tr>"};
</script>
</table>
</body>
</html>
```
Output

The following output is displayed on successful execution of the above code.

<table>
<thead>
<tr>
<th>Plugin Name</th>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Citrix Online Web Deployment Plugin 1.0.0.104</td>
<td>npapppdetector.dll</td>
<td>Citrix Online App Detector Plugin</td>
</tr>
<tr>
<td>Google Update</td>
<td>npGoogleUpdate51.dll</td>
<td>Google Update</td>
</tr>
<tr>
<td>Intel® Identity Protection Technology</td>
<td>npIntelWebAPIUpdater.dll</td>
<td>Intel® web components updater - Installs and updates the Intel web components</td>
</tr>
<tr>
<td>Intel® Identity Protection Technology</td>
<td>npIntelWebAPIUPIT.dll</td>
<td>Intel® web components for Intel® Identity Protection Technology</td>
</tr>
<tr>
<td>McAfee SecurityCenter</td>
<td>npMcSfaFPPL.dll</td>
<td>McAfee MSC FF plugin DLL</td>
</tr>
<tr>
<td>Microsoft Office 2013</td>
<td>npSWWRAP.DLL</td>
<td>The plugin allows you to have a better experience with Microsoft SharePoint</td>
</tr>
<tr>
<td>Microsoft Office 2013</td>
<td>npMeeting/outPluginOC.dll</td>
<td>The plugin allows you to have a better experience with Microsoft Lync</td>
</tr>
<tr>
<td>Picasa</td>
<td>npPicasa3.dll</td>
<td>Picasa plugin</td>
</tr>
<tr>
<td>Shockwave Flash</td>
<td>NPWF32_16_0_0_305.dll</td>
<td>Shockwave Flash 16.0.0</td>
</tr>
</tbody>
</table>

Checking for Plugins

Each plug-in has an entry in the array. Each entry has the following properties:

- **name** - The name of the plug-in.
- **filename** - The executable file that was loaded to install the plug-in.
- **description** - A description of the plug-in, supplied by the developer.
- **mimeTypes** - An array with one entry for each MIME type supported by the plug-in.

You can use these properties in a script to find out the installed plug-ins, and then using JavaScript, you can play the appropriate multimedia file. Take a look at the following code.

```html
<html>
<head>
<title>Using Plug-Ins</title>
</head>
<body>
<script language="JavaScript" type="text/javascript"> media = navigator.mimeTypes["video/quicktime"]; if (media){
    document.write("<embed src='quick.mov' height=100 width=100>");
} else{
    document.write("<img src='quick.gif' height=100 width=100>");
}
</script>
</body>
```
Controlling Multimedia

Let us take a real example which works in almost all the browsers.

```html
<html>
<head>
<title>Using Embedded Object</title>
<script type="text/javascript">
<!--
function play()
{
    if (!document.demo.IsPlaying())
    {
        document.demo.Play();
    }
}
function stop()
{
    if (document.demo.IsPlaying()){
        document.demo.StopPlay();
    }
}
function rewind()
    if (document.demo.IsPlaying()){
        document.demo.StopPlay();
        document.demo.Rewind();
    }
//-->
</script>
</head>
<body>
<embed id="demo" name="demo"
```
<embed src="http://www.amrood.com/games/kumite.swf" width="318" height="300"
play="false" loop="false"
pluginspage="http://www.macromedia.com/go/getflashplayer"
swliveconnect="true">
</embed>
<form name="form" id="form" action="#" method="get">
<input type="button" value="Start" onclick="play();" />
<input type="button" value="Stop" onclick="stop();" />
<input type="button" value="Rewind" onclick="rewind();" />
</form></body>
</html>
Every now and then, developers commit mistakes while coding. A mistake in a program or a script is referred to as a **bug**.

The process of finding and fixing bugs is called **debugging** and is a normal part of the development process. This chapter covers the tools and techniques that can help you with debugging tasks.

### Error Messages in IE

The most basic way to track down errors is by turning on the error information in your browser. By default, the Internet Explorer shows an error icon in the status bar when an error occurs on the page.

Double-clicking this icon takes you to a dialog box showing information about the specific error that has occurred.

Since this icon is easy to overlook, Internet Explorer gives you the option to automatically show the Error dialog box whenever an error occurs.

To enable this option, select **Tools --> Internet Options --> Advanced tab** and then finally check the “**Display a Notification about Every Script Error**” box option as shown in the following screenshot.
Error Messages in Firefox or Mozilla

Other browsers like Firefox, Netscape, and Mozilla send error messages to a special window called the JavaScript Console or Error Console. To view the console, select Tools --> Error Console or Web Development.

Unfortunately, since these browsers give no visual indication when an error occurs, you must keep the Console open and watch for errors as your script executes.

Error Notifications

Error notifications that show up on the Console or through Internet Explorer dialog boxes are the result of both syntax and runtime errors. These error notifications include the line number at which the error occurred.

If you are using Firefox, then you can click on the error available in the error console to go to the exact line in the script having the error.

Debugging a Script

There are various ways to debug your JavaScript. Following are some of the methods.

Use a JavaScript Validator

One way to check your JavaScript code for strange bugs is to run it through a program that checks it to make sure it is valid and that it follows the official syntax rules of the language. These programs are called validating parsers or just validators for short, and often come with commercial HTML and JavaScript editors.
The most convenient validator for JavaScript is Douglas Crockford's JavaScript Lint, which is available for free at Douglas Crockford's JavaScript Lint.

Simply visit the web page, paste your JavaScript (Only JavaScript) code into the text area provided, and click the **jshint** button. This program will parse through your JavaScript code, ensuring that all the variable and function definitions follow the correct syntax. It will also check JavaScript statements, such as if and while, to ensure they too follow the correct format.

### Add Debugging Code to Your Programs

You can use the `alert()` or `document.write()` methods in your program to debug your code. For example, you might write something as follows:

```javascript
var debugging = true; var whichImage = "widget"; if( debugging )
    alert( "Calls swapImage() with argument: " + whichImage );
var swapStatus = swapImage( whichImage ); if( debugging )
    alert( "Exits swapImage() with swapStatus=" + swapStatus );
```

By examining the content and order of the `alert()` as they appear, you can examine the health of your program very easily.

### Use a JavaScript Debugger

A **debugger** is an application that places all aspects of script execution under the control of the programmer. Debuggers provide fine-grained control over the state of the script through an interface that allows you to examine and set values as well as control the flow of execution.

Once a script has been loaded into a debugger, it can be run one line at a time or instructed to halt at certain breakpoints. Once the execution is halted, the programmer can examine the state of the script and its variables in order to determine if something is amiss. You can also watch variables for changes in their values.

The latest version of the Mozilla JavaScript Debugger (code-named Venkman) for both Mozilla and Netscape browsers can be downloaded from: [http://www.hacksrus.com/~ginda/venkman](http://www.hacksrus.com/~ginda/venkman).

### Useful Tips for Developers

You can keep the following tips in mind to reduce the number of errors in your scripts and simplify the debugging process:

- Use plenty of comments. Comments enable you to explain why you wrote the script the way you did and to explain particularly the difficult sections of the code.

- Always use indentation to make your code easy to read. Indenting statements also makes it easier for you to match up the beginning and ending tags, curly braces, and other HTML and script elements.
• Write modular code. Whenever possible, group your statements into functions. Functions let you group related statements, and test as well as reuse portions of the code with minimal effort.

• Be consistent in the way you name your variables and functions. Try using names that are long enough to be meaningful and that describe the contents of the variable or the purpose of the function.

• Use consistent syntax when naming variables and functions. In other words, keep them all lowercase or all uppercase; if you prefer Camel-Back notation, use it consistently.

• Test long scripts in a modular fashion. In other words, do not try to write the entire script before testing any portion of it. Write a piece and get it to work before adding the next portion of the code.

• Use descriptive variable and function names and avoid using single character names.

• Watch your quotation marks. Remember that quotation marks are used in pairs around strings and that both quotation marks must be of the same style (either single or double).

• Watch your equal signs. You should not use a single = for comparison purpose.

• Declare variables explicitly using the var keyword.

### Debugging with Node.js

Node.js includes a full-featured debugging utility. To use it, start Node.js with the debug argument followed by the path to the script to debug.

```
node debug test.js
```

A prompt indicating that the debugger has started successfully will be launched.

To apply a breakpoint at a specified location, call the debugger in the source code as shown in the following code.

```javascript
// myscript.js
x = 5;
setTimeout(() => {
    debugger;
    console.log('world');
}, 1000);
console.log('hello');
```
Following is a set of stepping commands that one can use with Node.

<table>
<thead>
<tr>
<th>Stepping Commands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cont,c</td>
<td>Continue</td>
</tr>
<tr>
<td>next,n</td>
<td>Next</td>
</tr>
<tr>
<td>step,s</td>
<td>Step in</td>
</tr>
<tr>
<td>out,o</td>
<td>Step out</td>
</tr>
<tr>
<td>pause</td>
<td>Pause the code. Similar to pause in the developer tools</td>
</tr>
</tbody>
</table>

A complete list of Node’s debugging commands can be found here: https://nodejs.org/api/debugger.html

**Visual Studio Code and Debugging**

One of the key features of Visual Studio Code is its great in-built debugging support for Node.js Runtime. For debugging code in other languages, it provides debugger extensions. The debugger provides a plethora of features that allow us to launch configuration files, apply/remove/disable and enable breakpoints, variable, or enable data inspection, etc.

A detailed guide on debugging using VS Code can be found here: https://code.visualstudio.com/docs/editor/debugging
You can use JavaScript to create a client-side image map. Client-side image maps are enabled by the usemap attribute for the `<img />` tag and defined by special `<map>` and `<area>` extension tags.

The image that is going to form the map is inserted into the page using the `<img />` element as normal, except that it carries an extra attribute called `usemap`. The value of the usemap attribute is the value of the name attribute on the `<map>` element, which you are about to meet, preceded by a pound or a hash sign.

The `<map>` element actually creates the map for the image and usually follows directly after the `<img />` element. It acts as a container for the `<area />` elements that actually define the clickable hotspots. The `<map>` element carries only one attribute, the name attribute, which is the name that identifies the map. This is how the `<img />` element knows which `<map>` element to use.

The `<area>` element specifies the shape and the coordinates that define the boundaries of each clickable hotspot.

The following code combines imagemaps and JavaScript to produce a message in a text box when the mouse is moved over different parts of an image.

```html
<html>
<head>
<title>Using JavaScript Image Map</title>
<script type="text/javascript">
!---
function showTutorial(name){
    document.myform.stage.value = name
}
//-->
</script>
</head>
<body>
<form name="myform">
<input type="text" name="stage" size="20" />
</form>
<!-- Create Mappings -->
<img src="/images/usemap.gif" alt="HTML Map" border="0" usemap="#tutorials" />
<map name="tutorials">
    <area shape="poly" coords="74,0,113,29,98,72,52,72,38,27"
        href="/perl/index.htm" alt="Perl Tutorial" target="_self"
        onMouseOver="showTutorial('perl')" onMouseOut="showTutorial('')"/>
</map>
</body>
</html>
```
The following output is displayed on successful execution of the above code. You can feel the map concept by placing the mouse cursor on the image object.
It is important to understand the differences between different browsers in order to handle each in the way it is expected. So it is important to know which browser your web page is running in. To get information about the browser your webpage is currently running in, use the built-in navigator object.

**Navigator Properties**

There are several Navigator related properties that you can use in your webpage. The following is a list of the names and its description.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Property and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>appCodeName</strong>&lt;br&gt;This property is a string that contains the code name of the browser, Netscape for Netscape and Microsoft Internet Explorer for Internet Explorer</td>
</tr>
<tr>
<td>2</td>
<td><strong>appVersion</strong>&lt;br&gt;This property is a string that contains the version of the browser as well as other useful information such as its language and compatibility</td>
</tr>
<tr>
<td>3</td>
<td><strong>language</strong>&lt;br&gt;This property contains the two-letter abbreviation for the language that is used by the browser. Netscape only</td>
</tr>
<tr>
<td>4</td>
<td><strong>mimTypes[]</strong>&lt;br&gt;This property is an array that contains all MIME types supported by the client. Netscape only</td>
</tr>
<tr>
<td>5</td>
<td><strong>platform[]</strong>&lt;br&gt;This property is a string that contains the platform for which the browser was compiled. &quot;Win32&quot; for 32-bit Windows operating systems</td>
</tr>
<tr>
<td>6</td>
<td><strong>plugins[]</strong>&lt;br&gt;This property is an array containing all the plug-ins that have been installed on the client. Netscape only</td>
</tr>
<tr>
<td>7</td>
<td><strong>userAgent[]</strong>&lt;br&gt;This property is a string that contains the code name and version of the browser. This value is sent to the originating server to identify the client</td>
</tr>
</tbody>
</table>
Navigator Methods

There are several Navigator-specific methods. Here is a list of their names and descriptions.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>javaEnabled()</td>
</tr>
<tr>
<td></td>
<td>This method determines if JavaScript is enabled in the client. If JavaScript is enabled, this method returns true; otherwise, it returns false</td>
</tr>
<tr>
<td>2</td>
<td>plugings.refresh</td>
</tr>
<tr>
<td></td>
<td>This method makes newly installed plug-ins available and populates the plugins array with all new plug-in names. Netscape only</td>
</tr>
<tr>
<td>3</td>
<td>preference(name,value)</td>
</tr>
<tr>
<td></td>
<td>This method allows a signed script to get and set some Netscape preferences. If the second parameter is omitted, this method will return the value of the specified preference; otherwise, it sets the value. Netscape only</td>
</tr>
<tr>
<td>4</td>
<td>taintEnabled()</td>
</tr>
<tr>
<td></td>
<td>This method returns true if data tainting is enabled; false otherwise</td>
</tr>
</tbody>
</table>

Browser Detection

The following JavaScript code can be used to find out the name of a browser and then accordingly an HTML page can be served to the user.

```html
<html>
<head>
<title>Browser Detection Example</title>
</head>
<body>
<script type="text/javascript">
!---
var userAgent   = navigator.userAgent;
var opera       = (userAgent.indexOf('Opera') != -1); var ie          =
(userAgent.indexOf('MSIE') != -1); var gecko       =
(userAgent.indexOf('Gecko') != -1); var netscape    =
(userAgent.indexOf('Mozilla') != -1); var version     = navigator.appVersion;
if (opera){
    document.write("Opera based browser"); // Keep your opera specific URL here.
}else if (gecko){
```
document.write("Mozilla based browser"); // Keep your gecko specific URL here.
}
else if (ie){
    document.write("IE based browser"); // Keep your IE specific URL here.
}
else if (netscape){
    document.write("Netscape based browser");
    // Keep your Netscape specific URL here.
}
else{
    document.write("Unknown browser");
}

// You can include version to along with any above condition.
document.write("<br /> Browser version info : " + version);
//-->
</script>
</body>
</html>

The following output is displayed on successful execution of the above code.

Mozilla based browser
Browser version info : 5.0

(Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.101 Safari/537.36