|  |
| --- |
|  |
| JAVASCRIPT |
| Nagarjun Sanji |
|  |
|  |
|  |

|  |
| --- |
|  |

# Javascript

JavaScript is a lightweight, interpreted programming language. It is designed for creating network-centric applications. It is complimentary to and integrated with Java. JavaScript is very easy to implement because it is integrated with HTML. It is open and cross-platform.

<html>

<body>

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

<!--

document.write("Hello World!")

//-->

</script>

</body>

</html>

## Statements

JavaScript statements are composed of - Values, Operators, Expressions, Keywords, and Comments.

This statement tells the browser to write "Hello Dolly." inside an HTML element with id="demo":

document.getElementById("demo").innerHTML = "Hello Dolly.";

Most JavaScript programs contain many JavaScript statements.

The statements are executed, one by one, in the same order as they are written.

**JavaScript Keywords**

JavaScript statements often start with a keyword to identify the JavaScript action to be performed.

Here is a list of some of the keywords you will learn about in this tutorial:

Keyword Description

break Terminates a switch or a loop

continue Jumps out of a loop and starts at the top

debugger Stops the execution of JavaScript, and calls (if available) the debugging function

do ... while Executes a block of statements, and repeats the block, while a condition is true

for Marks a block of statements to be executed, as long as a condition is true

function Declares a function

if ... else Marks a block of statements to be executed, depending on a condition

return Exits a function

switch Marks a block of statements to be executed, depending on different cases

try ... catch Implements error handling to a block of statements

var Declares a variable

## Comments

**Single Line Comments**

Single line comments start with //.

Any text between // and the end of the line will be ignored by JavaScript (will not be executed).

This example uses a single-line comment before each code line:

// Change heading:  
document.getElementById("myH").innerHTML = "My First Page";  
// Change paragraph:  
document.getElementById("myP").innerHTML = "My first paragraph.";

var x = 5;      // Declare x, give it the value of 5  
var y = x + 2;  // Declare y, give it the value of x + 2

**Multi-line Comments**

Multi-line comments start with /\* and end with \*/.

Any text between /\* and \*/ will be ignored by JavaScript.

This example uses a multi-line comment (a comment block) to explain the code:

/\*  
The code below will change  
the heading with id = "myH"  
and the paragraph with id = "myP"  
in my web page:  
\*/  
document.getElementById("myH").innerHTML = "My First Page";  
document.getElementById("myP").innerHTML = "My first paragraph.";

## Variables

All JavaScript variables must be identified with unique names.

These unique names are called identifiers.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

The general rules for constructing names for variables (unique identifiers) are:

* Names can contain letters, digits, underscores, and dollar signs.
* Names must begin with a letter
* Names can also begin with $ and \_ (but we will not use it in this tutorial)
* Names are case sensitive (y and Y are different variables)
* Reserved words (like JavaScript keywords) cannot be used as names

<p id="demo"></p>  
  
<script>  
var carName = "Volvo";  
document.getElementById("demo").innerHTML = carName;   
</script>

## Data Types

JavaScript variables can hold many data types: numbers, strings, objects and more:

var length = 16;                               // Number  
var lastName = "Johnson";                      // String  
var x = {firstName:"John", lastName:"Doe"};    // Object var y = false;     // Boolean

var cars = ["Saab", "Volvo", "BMW"]; // Array

JavaScript Types are Dynamic.

JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

var x;           // Now x is undefined  
x = 5;           // Now x is a Number  
x = "John";      // Now x is a String

## Function

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is executed when "something" invokes it (calls it).

var x = myFunction(4, 3);// Function is called, return value will end up in x  
  
function myFunction(a, b) {  
    return a \* b;            // Function returns the product of a and b  
}

## Window Location

The window.location object can be written without the window prefix.

Some examples:

* window.location.href returns the href (URL) of the current page
* window.location.hostname returns the domain name of the web host
* window.location.pathname returns the path and filename of the current page
* window.location.protocol returns the web protocol used (http: or https:)
* window.location.assign loads a new document

Display the href (URL) of the current page:

document.getElementById("demo").innerHTML = "Page location is " + window.location.href;

## History

The window.history object can be written without the window prefix.

To protect the privacy of the users, there are limitations to how JavaScript can access this object.

Some methods:

* history.back() - same as clicking back in the browser
* history.forward() - same as clicking forward in the browser

**Window History Back**

The history.back() method loads the previous URL in the history list.

This is the same as clicking the Back button in the browser.

<html>  
<head>  
<script>  
function goBack() {  
    window.history.back()  
}  
</script>  
</head>  
<body>  
  
<input type="button" value="Back" onclick="goBack()">  
  
</body>  
</html>

**Window History Forward**

The history forward() method loads the next URL in the history list.

This is the same as clicking the Forward button in the browser.

<html>  
<head>  
<script>  
function goForward() {  
    window.history.forward()  
}  
</script>  
</head>  
<body>  
  
<input type="button" value="Forward" onclick="goForward()">  
  
</body>  
</html>

## Prototypes

All JavaScript objects inherit properties and methods from a prototype.

function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
}  
var myFather = new Person("John", "Doe", 50, "blue");  
var myMother = new Person("Sally", "Rally", 48, "green");

Adding Properties and Methods to Objects

Sometimes you want to add new properties (or methods) to all existing objects of a given type.

Sometimes you want to add new properties (or methods) to an object constructor.

Using the prototype Property

The JavaScript prototype property allows you to add new properties to object constructors:

function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
}  
Person.prototype.nationality = "English";

The JavaScript prototype property also allows you to add new methods objects constructors:

function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
}  
Person.prototype.name = function() {  
    return this.firstName + " " + this.lastName;  
};

## Design Patterns

**Module Design Pattern**

JavaScript modules are the most prevalently used design patterns for keeping particular pieces of code independent of other components. This provides loose coupling to support well-structured code.

For those that are familiar with object-oriented languages, modules are JavaScript "classes". One of the many advantages of classes is encapsulation - protecting states and behaviors from being accessed from other classes. The module pattern allows for public and private (plus the lesser-know protected and privileged) access levels.

Modules should be Immediately-Invoked-Function-Expressions (IIFE) to allow for private scopes - that is, a closure that protect variables and methods (however, it will return an object instead of a function). This is what it looks like:

(function() {

// declare private variables and/or functions

return {

// declare public variables and/or functions

}

})();

Here we instantiate the private variables and/or functions before returning our object that we want to return. Code outside of our closure is unable to access these private variables since it is not in the same scope. Let's take a more concrete implementation:

var Exposer = (function() {

var privateVariable = 10;

var privateMethod = function() {

console.log('Inside a private method!');

privateVariable++;

}

var methodToExpose = function() {

console.log('This is a method I want to expose!');

}

var otherMethodIWantToExpose = function() {

privateMethod();

}

return {

first: methodToExpose,

second: otherMethodIWantToExpose

};

})();

Exposer.first(); // Output: This is a method I want to expose!

Exposer.second(); // Output: Inside a private method!

Exposer.methodToExpose; // undefined

**Prototype Design Pattern**

Any JavaScript developer has either seen the keyword prototype, confused by the prototypical inheritance, or implemented prototypes in their code. The Prototype design pattern relies on the JavaScript prototypical inheritance. The prototype model is used mainly for creating objects in performance-intensive situations.

The objects created are clones (shallow clones) of the original object that are passed around. One use case of the prototype pattern is performing an extensive database operation to create an object used for other parts of the application. If another process needs to use this object, instead of having to perform this substantial database operation, it would be advantageous to clone the previously created object.

var TeslaModelS = function() {

this.numWheels = 4;

this.manufacturer = 'Tesla';

this.make = 'Model S';

}

TeslaModelS.prototype.go = function() {

// Rotate wheels

}

TeslaModelS.prototype.stop = function() {

// Apply brake pads

}

**Singleton**

A Singleton only allows for a single instantiation, but many instances of the same object. The Singleton restricts clients from creating multiple objects, after the first object created, it will return instances of itself.

Finding use cases for Singletons is difficult for most who have not yet used it prior. One example is using an office printer. If there are ten people in an office, and they all use one printer, ten computers share one printer (instance). By sharing one printer, they share the same resources.

var printer = (function () {

var printerInstance;

function create () {

function print() {

// underlying printer mechanics

}

function turnOn() {

// warm up

// check for paper

}

return {

// public + private states and behaviors

print: print,

turnOn: turnOn

};

}

return {

getInstance: function() {

if(!printerInstance) {

printerInstance = create();

}

return printerInstance;

}

};

function Singleton () {

if(!printerInstance) {

printerInstance = intialize();

}

};

})();

## AJAX

AJAX = Asynchronous JavaScript And XML.

AJAX is not a programming language.

AJAX just uses a combination of:

* A browser built-in XMLHttpRequest object (to request data from a web server)
* JavaScript and HTML DOM (to display or use the data)

AJAX allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.



* An event occurs in a web page (the page is loaded, a button is clicked)
* An XMLHttpRequest object is created by JavaScript
* The XMLHttpRequest object sends a request to a web server
* The server processes the request
* The server sends a response back to the web page
* The response is read by JavaScript
* Proper action (like page update) is performed by JavaScript