ECMAScript (ES) is a scripting language specification standardized by ECMAScript International. It is used by applications to enable client-side scripting.

## 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](https://nodejs.org/en/download/)

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.

//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 −

//Declaration using var keyword

var variable\_name = value

### Example : Using Variables

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**.

## JavaScriptVariable 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).

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 in 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.

"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

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.

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.

### Example

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

var main = function() {

for(var 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();

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 −

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.

**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

class Class\_name {

}

### Syntax: Class Expressions

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

class Polygon {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

### Example: Class Expression

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.

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.

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

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.

//accessing a function

obj.function\_name()

### Example: Putting them together

'use strict'

class Polygon {

constructor(height, width) {

this.h = height;

this.w = width;

}

test() {

console.log("The height of the polygon: ", this.h)

console.log("The width of the polygon: ",this. w)

}

}

//creating an instance

var polyObj = new Polygon(10,20);

polyObj.test();

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.

Static methods are defined on the class itself, and not on the prototype.

That means you cannot call a static method on the object (mycar), but on the class (Car):

### Example

'use strict'

class StaticMem {

static disp() {

console.log("Static Function called")

}

}

StaticMem.disp() //invoke the static metho

**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 Circle 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.

'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();

obj.test()

The class Leaf derives the attributes from the Root and the Child classes by virtue of multilevel 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.

The super() method refers to the parent class.

By calling the super() method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

Inheritance is useful for code reusability: reuse properties and methods of an existing class when you create a new class.

Consider the following example −

'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 StringWriter, 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.

### Arrow Functions

Arrow functions are a great addition to the JavaScript language. They make for short and concise code. We are introducing arrow functions early in this article so that we can take advantage of them in other examples later on. The next code snippet shows an arrow function, with the same function written in the familiar ES5 style:

let books = [{title: 'X', price: 10}, {title: 'Y', price: 15}];

let titles = books.map( item => item.title );

// ES5 equivalent:

var titles = books.map(function(item) {

return item.title;

});

# Arrow Function

### With Arrow Function:

hello = () => {  
  return "Hello World!";  
}

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Arrow Function</h2>

<p>This example shows the syntax of an Arrow Function, and how to use it.</p>

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

<script>

var hello;

hello = () => {

return "Hello World!";

}

document.getElementById("demo").innerHTML = hello();

</script>

</body>

</html>

# Template literals (Template strings)

Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them. They were called "template strings" in prior editions of the ES2015 specification.

console.log('string text line 1\n' +

'string text line 2');

// "string text line 1

// string text line 2"