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.

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.

|  |  |  |  |
| --- | --- | --- | --- |
| Break | as | Any | Switch |
| Case | if | Throw | Else |
| Var | number | String | Get |
| Module | type | Instanceof | Typeof |
| Finally | for | Enum | Export |
| While | void | This | New |
| Null | super | Catch | let |
| Static | return | True | False |

* **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.

|  |  |
| --- | --- |
| **Examples of valid identifiers** | **Examples of invalid identifiers** |
| firstName  first\_name  num1  $result | Var#  first name  first-name  1number |

## 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

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.

AD

## 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

//this is single line comment

/\* This is a

Multi-line comment

\*/

## Your First JavaScript Code

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

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.

## 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 −

// 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 −

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-strict mode. All statements within the function will be executed in the Strict Mode.

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.

//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

AD

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 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.

AD

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.

let and multiple blocks

However, the same **let** variable can be used in different block level scopes without any syntax errors.

Example

<script>

let count = 100

for (let count = 1;count <= 10;count++){

//inside for loop brackets ,count value starts from 1

console.log("count value inside loop is ",count);

}

//outside for loop brackets ,count value is 100

console.log("count value after loop is",count);

if(count == 100){

//inside if brackets ,count value is 50

let count = 50;

console.log("count inside if block",count);

}

console.log(count);

</script>

The output of the above code will be as follows −

count value inside loop is 1

count value inside loop is 2

count value inside loop is 3

count value inside loop is 4

count value inside loop is 5

count value inside loop is 6

count value inside loop is 7

count value inside loop is 8

count value inside loop is 9

count value inside loop is 10

count value after loop is 100

count inside if block 50

100

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

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.

Constants are Immutable

Unlike variables declared using **let** keyword, **constants** are immutable. This means its value cannot be changed. For example, if we try to change value of the constant variable, an error will be displayed.

<script>

let income = 100000

const INTEREST\_RATE = 0.08

income += 50000 // mutable

console.log("changed income value is ",income)

INTEREST\_RATE += 0.01

console.log("changed rate is ",INTEREST\_RATE) //Error: not mutable

</script>

The output of the above code will be as follows −

changed income value is 150000

Uncaught TypeError: Assignment to constant variable

The var keyword

Prior to ES6, the **var** keyword was used to declare a variable in JavaScript. Variables declared using **var** do not support block level scope. This means if a variable is declared in a loop or **if block** it can be accessed outside the loop or the **if block**. This is because the variables declared using the **var** keyword support hoisting.

var and hoisting

**Variable hoisting** allows the use of a variable in a JavaScript program, even before it is declared. Such variables will be initialized to **undefined** by default. JavaScript runtime will scan for variable declarations and put them to the top of the function or script. Variables declared with **var** keyword get hoisted to the top. Consider the following example −

<script>

variable company is hoisted to top ,

var company = undefined

console.log(company); // using variable before declaring

var company = "ROGERSOFT"; // declare and initialized here

console.log(company);

</script>

The output of the above code will be as shown below −

Undefined  
ROGERSOFT

var 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. Variables declared using the **var** keyword do not have a block scope. Consider the following example −

<script>

//hoisted to top ; var i = undefined

for (var i = 1;i <= 5;i++){

console.log(i);

}

console.log("after the loop i value is "+i);

</script>

The output of the above code will be as follows −

1

2

3

4

5

after the loop i value is 6

The variable **i** is declared inside the for loop using the **var** keyword. The variable i is accessible outside the loop. However, at times, there might be a need to restrict a variable's access within a block. We cannot use the **var** keyword in this scenario. ES6 introduces the **let** keyword to overcome this limitation.

var and block level safety

If we declare the same **variable** twice using the **var keyword** within a block, the compiler will not throw an error. However, this may lead to unexpected logical errors at runtime.

<script>

var balance = 5000

console.log(typeof balance)

var balance = {message:"hello"}

console.log(typeof balance)

</script>

The output of the above code is as shown below −

number

object

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.

[Show Examples](https://www.tutorialspoint.com/es6/es6_arithmetic_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Function** | **Example** |
| + | **Addition**  Returns the sum of the operands. | a + b is 15 |
| - | **Subtraction**  Returns the difference of the values. | a-b is 5 |
| \* | **Multiplication**  Returns the product of the values. | a\*b is 50 |
| / | **Division**  Performs a division operation and returns the quotient. | a/b is 2 |
| % | **Modulus**  Performs a division and returns the remainder. | a%b is 0 |
| ++ | **Increment**  Increments the value of the variable by one. | a++ is 11 |
| -- | **Decrement**  Decrements the value of the variable by one. | a-- is 9 |

AD

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.

[Show Examples](https://www.tutorialspoint.com/es6/es6_relational_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operators** | **Description** | **Example** |
| > | Greater than | (A > B) is False |
| < | Lesser than | (A < B) is True |
| >= | Greater than or equal to | (A >= B) is False |
| <= | Lesser than or equal to | (A <= B) is True |
| == | Equality | (A == B) is False |
| != | Not Equal | (A!= B) is True |

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.

[Show Examples](https://www.tutorialspoint.com/es6/es6_logical_operators_examples.htm).

|  |  |  |
| --- | --- | --- |
| **Operators** | **Description** | **Example** |
| && | **And**  The operator returns true only if all the expressions specified return true. | (A > 10 && B > 10) is False |
| || | **Or**  The operator returns true if at least one of the expressions specified return true. | (A > 10 || B > 10) is True |
| ! | **Not**  The operator returns the inverse of the expression’s result. For E.g.: !(7>5) returns false. | !(A > 10) is True |

## Assignment Operators

The following table summarizes Assignment operators.

[Show Examples](https://www.tutorialspoint.com/es6/es6_assignment_operators_examples.htm).

|  |  |
| --- | --- |
| **Sr.No** | **Operator & Description** |
| 1 | **= (Simple Assignment)**  Assigns values from the right side operand to the left side operand.  **Example** − C = A + B will assign the value of A + B into C |
| 2 | **+= (Add and Assignment)**  It adds the right operand to the left operand and assigns the result to the left operand.  **Example** − C += A is equivalent to C = C + A |
| 3 | **-= (Subtract and Assignment)**  It subtracts the right operand from the left operand and assigns the result to the left operand.  **Example** C -= A is equivalent to C = C – A |
| 4 | **\*= (Multiply and Assignment)**  It multiplies the right operand with the left operand and assigns the result to the left operand.  **Example** C \*= A is equivalent to C = C \* A |
| 5 | **/= (Divide and Assignment)**  It divides the left operand with the right operand and assigns the result to the left operand. |

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

## 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.

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.

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**

var num = -2

var result = num > 0 ?"positive":"non-positive"

console.log(result)

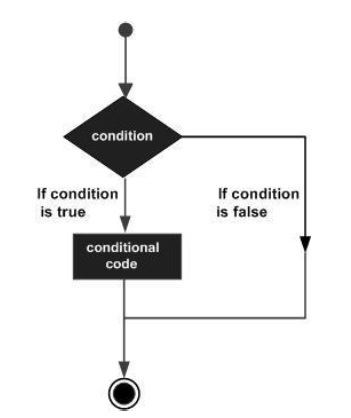
Line 2 checks whether the value in the variable num is greater than zero. If num is set to a value greater than zero, it returns the string “positive” else a “non-positive” string is returned.

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

non-positive

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

CONDITIONAL BRANCHING



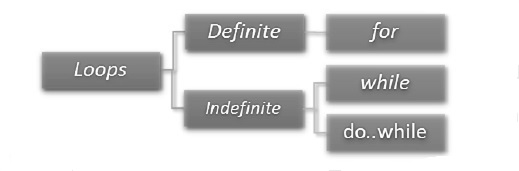
Conditional constructs in JavaScript are classified in the following table.

|  |  |
| --- | --- |
| **Sr.No** | **Statement & Description** |
| 1 | [if Statement](https://www.tutorialspoint.com/es6/es6_if_statement.htm)  An ‘if’ statement consists of a Boolean expression followed by one or more statements .  Following is the syntax.  if(boolean\_expression) {  // statement(s) will execute if the Boolean expression is true  }  var num = 5  if (num>0) {  console.log("number is positive")  } |
| 2 | [if…else Statement](https://www.tutorialspoint.com/es6/es6_if_else_statement.htm)  An ‘if’ statement can be followed by an optional ‘else’ statement, which executes when the Boolean expression is false.  Following is the syntax.  if(boolean\_expression) {  // statement(s) will execute if the Boolean expression is true  } else {  // statement(s) will execute if the Boolean expression is false  } Example: Simple if…else var num = 12;  if (num % 2 == 0) {  console.log("Even");  } else {  console.log("Odd");  } |
| 3 | [The else.. if ladder/nested if statements](https://www.tutorialspoint.com/es6/es6_else_if_ladder.htm)  The else…if ladder is useful to test multiple conditions. Following is the syntax of the same.  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  } Example: else…if ladder 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")  } |
| 4 | [switch…case Statement](https://www.tutorialspoint.com/es6/es6_switch_case_statement.htm)  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;  break;  }  default: {  //statements;  break;  }  } Example: switch…case 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;  }  } |

# ES6 - Loops

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 −



## 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**.

for (initial\_count\_value; termination-condition; step) {

//statements

}

|  |  |
| --- | --- |
| **Sr.No** | **Definite Loop & Description** |
| 1 | [The ‘for’ loop](https://www.tutorialspoint.com/es6/es6_for_loop.htm)  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.  var num = 5  var factorial=1;  for( let i = num ; i >= 1; i-- ) {  factorial \*= i ;  }  console.log(factorial); |
| 2 | [The for…in loop](https://www.tutorialspoint.com/es6/es6_for_in_loop.htm)  The for...in loop is used to loop through an object's properties.  Following is the syntax of ‘for…in’ loop.  for (variablename 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 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 |
| 3 | [The for…of loop](https://www.tutorialspoint.com/es6/es6_for_of_loop.htm)  The for…of loop is used to iterate iterables instead of object literals.  Following is the syntax of ‘for…of’ loop.  for (variablename of object){  statement or block to execute  } Example 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 |

AD

# ES6 - Functions

**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.

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.

function\_name()

### Example : Simple function definition

//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 −

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.

function func\_name( param1,param2 ,…..paramN) {

......

......

}

**Example − Parameterized 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.

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.

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.

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

For better understanding, let us consider the below example.

### Example 1

The following example shows a function which takes two parameters and returns their sum. The second parameter has a default value of 10. This means, if no value is passed to the second parameter, its value will be 10.

<script>

function addTwoNumbers(first,second = 10){

console.log('first parameter is :',first)

console.log('second parameter is :',second)

return first+second;

}

console.log("case 1 sum:",addTwoNumbers(20)) // no value

console.log("case 2 sum:",addTwoNumbers(2,3))

console.log("case 3 sum:",addTwoNumbers())

console.log("case 4 sum",addTwoNumbers(1,null))//null passed

console.log("case 5 sum",addTwoNumbers(3,undefined))

</script>

The output of the above code will be as mentioned below −

first parameter is : 20

second parameter is : 10

case 1 sum: 30

first parameter is : 2

second parameter is : 3

case 2 sum: 5

first parameter is : undefined

second parameter is : 10

case 3 sum: NaN

first parameter is : 1

second parameter is : null

case 4 sum 1

first parameter is : 3

second parameter is : 10

case 5 sum 13

### Example 2

<script>

let DEFAULT\_VAL = 30

function addTwoNumbers(first,second = DEFAULT\_VAL){

console.log('first parameter is :',first)

console.log('second parameter is :',second)

return first+second;

}

console.log("case 1 sum",addTwoNumbers(1))

console.log("case 2 sum",addTwoNumbers(3,undefined))

</script>

The output of the above code will be as shown below −

first parameter is : 1

second parameter is : 30

case 1 sum 31

first parameter is : 3

second parameter is : 30

case 2 sum 33

AD

## 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.

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.

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

**Example − Anonymous Function**

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**

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

# ES6 - Arrays

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.

## Declaring and Initializing Arrays

To declare and initialize an array in JavaScript use the following syntax −

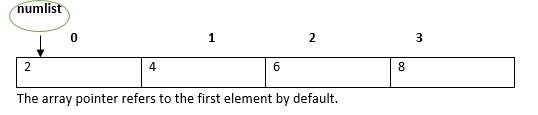
var array\_name; //declaration

array\_name = [val1,val2,valn..] //initialization

OR

var array\_name = [val1,val2…valn]

For example, a declaration like: **var numlist = [2,4,6,8]** will create an array as shown in the following figure.



## 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

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

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

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])

}

Array Methods

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

|  |  |
| --- | --- |
| **Sr.No** | **Method & Description** |
| 1 | concat()  Returns a new array comprised of this array joined with other array(s) and/or value(s)  concat() method returns a new array comprised of this array joined with two or more arrays.  Syntax  array.concat(value1, value2, ..., valueN);  **Example:**  var alpha = ["a", "b", "c"];  var numeric = [1, 2, 3];  var alphaNumeric = alpha.concat(numeric);  console.log("alphaNumeric : " + alphaNumeric );  Output  alphaNumeric : a,b,c,1,2,3 |
| 2 | **every()**  Returns true if every element in this array satisfies the provided testing function.  every method tests whether all the elements in an array passes the test implemented by the provided function.  Syntax  array.every(callback[, thisObject]);   * **callback** − Function to test for each element. * **thisObject** − Object to use as this when executing callback.   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 |
| 3 | **filter()**  Creates a new array with all of the elements of this array for which the provided filtering function returns true.  filter() method creates a new array with all elements that pass the test implemented by the provided function.  Syntax  array.filter(callback[, thisObject]);  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 |
| 4 | **forEach()**  forEach() method calls a function for each element in the array.  Syntax  array.forEach(callback[, thisObject]);  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)  })  Output  Printing Original Array….  12  13  14  15  Printing Reversed array…  15  14  13  1 |
| 5 | **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]);  Parameters   * **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  var index = [12, 5, 8, 130, 44].indexOf(8);  console.log("index is : " + index )  Output  index is : 2 |
| 6 | **join()**  join() method joins all the elements of an array into a string.  Syntax  array.join(separator);  Parameters   * **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  var arr = new Array("First","Second","Third");  var str = arr.join();  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 |
| 7 | **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  array.lastIndexOf(searchElement[, fromIndex]);  Parameters   * **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  var index = [12, 5, 8, 130, 44].lastIndexOf(8);  console.log("index is : " + index );  Output  index is : 3 |
| 8 | **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]);  Parameters   * **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  var numbers = [1, 4, 9];  var roots = numbers.map(Math.sqrt);  console.log("roots is : " + roots );  Output  roots is : 1,2,3 |
| 9 | **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  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 |
| 10 | **push()**  Adds one or more elements to the end of an array and returns the new length of the array.  push() method appends the given element(s) in the last of the array and returns the length of the new array.  Syntax  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  var numbers = new Array(1, 4, 9);  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 |
| 11 | **reduce()**  Applies a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value.  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  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  var total = [0, 1, 2, 3].reduce(function(a, b){ return a + b; });  console.log("total is : " + total );  Output  total is : 6 |
| 12 | **reduceRight()**  Applies a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value.  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  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  var total = [0, 1, 2, 3].reduceRight(function(a, b){ return a + b; });  console.log("total is : " + total );  Output  total is : 6 |
| 13 | **reverse()**  Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first.  Syntax  array.reverse();  Return Value  Returns the reversed single value of the array.  Example  var arr = [0, 1, 2, 3].reverse();  console.log("Reversed array is : " + arr );  Output  Reversed array is : 3,2,1,0 |
| 14 | **shift()**  Removes the first element from an array and returns that element slice.  Syntax  array.shift();  Return Value  Returns the removed single value of the array.  Example  var arr = [10, 1, 2, 3].shift();  console.log("Shifted value is : " + arr )  Output  Shifted value is : 10 |
| 15 | **slice()**  Extracts a section of an array and returns a new array.  Syntax  array.slice( begin [,end] );  Parameter Details   * **begin** − Zero-based index at which to begin extraction. As a negative index, start indicates an offset from the end of the sequence. * **end** − Zero-based index at which to end extraction.   Return Value  Returns the extracted array based on the passed parameters.  Example  var arr = ["orange", "mango", "banana", "sugar", "tea"];  console.log("arr.slice( 1, 2) : " + arr.slice( 1, 2) );  console.log("arr.slice( 1, 3) : " + arr.slice( 1, 3) );  Output  arr.slice( 1, 2) : mango  arr.slice( 1, 3) : mango,banana |
| 16 | **some()**  Returns true if at least one element in this array satisfies the provided testing function.  Syntax  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  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  Returned value is : true |
|  |  |
| 18 | **sort()**  Sorts the elements of an array.  array.sort();  Return Value  Returns a sorted array.  Example  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 |
| 19 | **splice()**  Adds and/or removes elements from an array.  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  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);  Output  After adding 1: orange,mango,water,banana,sugar,tea  removed is:  After adding 1: orange,mango,water,sugar,tea  removed is: banana |
| 20 | **toString()**  Returns a string representing the array and its elements.  Syntax  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 ); |
| 21 | **unshift()**  Adds one or more elements to the front of an array and returns the new length of the array.  Syntax  array.unshift( element1, ..., elementN );  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 |

**String Methods**

**slice() extracts a part of the string based on the given stating-index and ending-index and returns a new string.**

**JavaScript**

**// Define a string variable**

**let A = 'Rogersoft Training ';**

**// Use the slice() method to extract a substring**

**let b = A.slice(0, 5);**

**let c = A.slice(6, 9);**

**let d = A.slice(10);**

**// Output the value of variable**

**console.log(b);**

**console.log(c);**

**console.log(d);**

**Output**

**Roger**

**soft**

**Training**

**substring()**

**returns the part of the given string from the start index to the end index. Indexing starts from zero (0).**

**JavaScript**

**1**

**// Define a string variable**

**2**

**let str = "Mind, Power, Soul";**

**3**

**​**

**4**

**// Use the substring() method to extract a substring**

**5**

**let part = str.substring(6, 11);**

**6**

**​**

**7**

**// Output the value of variable**

**8**

**console.log(part);**

**Output**

**Power**

**substr()**

**substr() This method returns the specified number of characters from the specified index from the given string. It extracts a part of the original string.**

**JavaScript**

**1**

**// Define a string variable 'str'**

**2**

**let str = "Mind, Power, Soul";**

**3**

**​**

**4**

**// Use the substr() method to extract a substring f**

**5**

**let part = str.substr(6, 5);**

**6**

**​**

**7**

**// Output the value of variable**

**8**

**console.log(part);**

**Output**

**Power**

**replace()**

**replace() replaces a part of the given string with another string or a regular expression. The original string will remain unchanged.**

**JavaScript**

**1**

**// Define a string variable 'str'**

**2**

**let str = "Mind, Power, Soul";**

**3**

**​**

**4**

**// Use the replace() method to replace the substring**

**5**

**let part = str.replace("Power", "Space");**

**6**

**​**

**7**

**// Output the resulting string after replacement**

**8**

**console.log(part);**

**Output**

**Mind, Space, Soul**

**replaceAll()**

**replaceAll() returns a new string after replacing all the matches of a string with a specified string or a regular expression. The original string is left unchanged after this operation.**

**JavaScript**

**1**

**// Define a string variable 'str'**

**2**

**let str = "Mind, Power, Power, Soul";**

**3**

**​**

**4**

**// Use the replaceAll() method to replace all occurrences**

**5**

**//of "Power" with "Space" in the string 'str'**

**6**

**let part = str.replaceAll("Power", "Space");**

**7**

**​**

**8**

**// Output the resulting string after replacement**

**9**

**console.log(part);**

**Output**

**Mind, Space, Space, Soul**

**toUpperCase()**

**toUpperCase() converts all the characters present in the String to upper case and returns a new String with all characters in upper case. This method accepts single parameter stringVariable string that you want to convert in upper case.**

**JavaScript**

**1**

**// Define a string variable**

**2**

**let gfg = 'GFG ';**

**3**

**​**

**4**

**// Define another string variable**

**5**

**let geeks = 'stands-for-GeeksforGeeks';**

**6**

**​**

**7**

**// Convert the string 'geeks' to uppercase using the toUpperCase() method**

**8**

**console.log(geeks.toUpperCase());**

**Output**

**STANDS-FOR-GEEKSFORGEEKS**

**toLowerCase()**

**toLowerCase() converts all the characters present in the so lowercase and returns a new string with all the characters in lowercase.**

**JavaScript**

**1**

**// Define a string variable**

**2**

**let gfg = 'GFG ';**

**3**

**​**

**4**

**// Define a string variable**

**5**

**let geeks = 'stands-for-GeeksforGeeks';**

**6**

**​**

**7**

**// Convert the string 'geeks' to lowercase using the toLowerCase() method**

**8**

**console.log(geeks.toLowerCase());**

**Output**

**stands-for-geeksforgeeks**

**concat()**

**concat() combines the text of two strings and returns a new combined or joined string. To concatenate two strings, we use the concat() method on one object of string and send another object of string as a parameter. This method accepts one argument. The variable contains text in double quotes or single quotes.**

**JavaScript**

**1**

**let gfg = 'GFG ';**

**2**

**let geeks = 'stands for GeeksforGeeks';**

**3**

**​**

**4**

**// Accessing concat method on an object**

**5**

**// of String passing another object**

**6**

**// as a parameter**

**7**

**console.log(gfg.concat(geeks));**

**Output**

**GFG stands for GeeksforGeeks**

**trim()**

**trim() is used to remove either white spaces from the given string. This method returns a new string with removed white spaces. This method is called on a String object. This method doesn’t accept any parameter.**

**JavaScript**

**1**

**let gfg = 'GFG ';**

**2**

**let geeks = 'stands-for-GeeksforGeeks';**

**3**

**​**

**4**

**// Storing new object of string**

**5**

**// with removed white spaces**

**6**

**let newGfg = gfg.trim();**

**7**

**​**

**8**

**// Old length**

**9**

**console.log(gfg.length);**

**10**

**​**

**11**

**// New length**

**12**

**console.log(newGfg.length)**

**Output**

**7**

**3**

**trimStart()**

**trimStart() removes whitespace from the beginning of a string. The value of the string is not modified in any manner, including any whitespace present after the string.**

**JavaScript**

**1**

**// Define a string variable**

**2**

**let str = " Soul";**

**3**

**​**

**4**

**// Output the original value of the string**

**5**

**console.log(str);**

**6**

**​**

**7**

**// Use the trimStart() method to remove leading whitespace from the string 'str'**

**8**

**let part = str.trimStart();**

**9**

**​**

**10**

**// Output the resulting string after removing leading whitespace**

**11**

**console.log(part);**

**Output**

**Soul**

**Soul**

**trimEnd()**

**trimEnd() removes white space from the end of a string. The value of the string is not modified in any manner, including any white-space present before the string.**

**JavaScript**

**1**

**// Define a string variable**

**2**

**let str = "Soul ";**

**3**

**​**

**4**

**// Output the original value of the string 'str'**

**5**

**console.log(str);**

**6**

**​**

**7**

**// Use the trimEnd() method to remove trailing whitespace from the string 'str'**

**8**

**let part = str.trimEnd();**

**9**

**​**

**10**

**// Output the resulting string after removing trailing whitespace**

**11**

**console.log(part);**

**Output**

**Soul**

**Soul**

**padStart()**

**padStart() pad a string with another string until it reaches the given length. The padding is applied from the left end of the string.**

**JavaScript**

**1**

**// Define a string variable**

**2**

**let stone = "Soul";**

**3**

**​**

**4**

**// Use the padStart() method to add padding characters "Mind "**

**5**

**//to the beginning of the string 'stone'**

**6**

**stone = stone.padStart(9, "Mind ");**

**7**

**​**

**8**

**// Output the resulting string after padding**

**9**

**console.log(stone);**

**Output**

**Mind Soul**

**padEnd()**

**padEnd() pad a string with another string until it reaches the given length. The padding is applied from the right end of the string.**

**JavaScript**

**1**

**// Define a string variable**

**2**

**let stone = "Soul";**

**3**

**​**

**4**

**// Use the padEnd() method to add padding characters**

**5**

**//" Power" to the end of the string 'stone'**

**6**

**stone = stone.padEnd(10, " Power");**

**7**

**​**

**8**

**// Output the resulting string after padding**

**9**

**console.log(stone);**

**Output**

**Soul Power**

**charAt()**

**charAt() returns the character at the specified index. String in JavaScript has zero-based indexing.**

**JavaScript**

**1**

**let gfg = 'GeeksforGeeks';**

**2**

**let geeks = 'GfG is the best platform to learn and\n'+**

**3**

**'experience Computer Science.';**

**4**

**​**

**5**

**// Print the string as it is**

**6**

**console.log(gfg);**

**7**

**​**

**8**

**console.log(geeks);**

**9**

**​**

**10**

**// As string index starts from zero**

**11**

**// It will return first character of string**

**12**

**console.log(gfg.charAt(0));**

**13**

**​**

**14**

**console.log(geeks.charAt(5));**

**Output**

**GeeksforGeeks**

**GfG is the best platform to learn and**

**experience Computer Science.**

**G**

**s**

**charCodeAt()**

**charCodeAt() returns a number that represents the Unicode value of the character at the *specified index*. This method accepts one argument.**

**JavaScript**

**1**

**let gfg = 'GeeksforGeeks';**

**2**

**let geeks = 'GfG is the best platform\n\**

**3**

**to learn and experience\n\**

**4**

**Computer Science.';**

**5**

**​**

**6**

**// Return a number indicating Unicode**

**7**

**// value of character at index 0 ('G')**

**8**

**console.log(gfg.charCodeAt(0));**

**9**

**console.log(geeks.charCodeAt(5));**

**Output**

**71**

**115**

**split()**

**split() splits the string into an array of sub-strings. This method returns an array. This method accepts a single parameter character on which you want to split the string.**

**JavaScript**

**1**

**let gfg = 'GFG '**

**2**

**let geeks = 'stands-for-GeeksforGeeks'**

**3**

**​**

**4**

**// Split string on '-'.**

**5**

**console.log(geeks.split('-'))**

**Output**

**[ 'stands', 'for', 'GeeksforGeeks' ]**

**Add Event Listener**

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript addEventListener()</h2>

<p>addEventListener() method to attach a click event to a button.</p>

<button id="myBtn">Try it</button>

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

<script>

document.getElementById("myBtn").addEventListener("click", displayDate);

function displayDate() {

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

}

</script>

</body>

</html>

**Example 2**

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript addEventListener()</h2>

<p>This example uses the addEventListener() method to add many events on the same button.</p>

<button id="myBtn">Try it</button>

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

<script>

var x = document.getElementById("myBtn");

x.addEventListener("mouseover", myFunction);

x.addEventListener("click", mySecondFunction);

x.addEventListener("mouseout", myThirdFunction);

function myFunction() {

document.getElementById("demo").innerHTML += "Moused over!<br>";

}

function mySecondFunction() {

document.getElementById("demo").innerHTML += "Clicked!<br>";

}

function myThirdFunction() {

document.getElementById("demo").innerHTML += "Moused out!<br>";

}

</script>

</body>

</html>

AD

<!doctype html>

<html>

<body>

<h2>javascript addeventlistener()</h2>

<p>this example demonstrates how to pass parameter values when using the addeventlistener() method.</p>

<p>click the button to perform a calculation.</p>

<button id="mybtn">try it</button>

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

<script>

let p1 = 5;

let p2 = 7;

document.getElementById("mybtn"). addEventListener ("click", function() {

myfunction(p1, p2);

});

function myfunction(a, b) {

document.getElementById("demo").innerHTML = a \* b;

}

</script>

</body>

</html>

# ES6 - Objects

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 life time. 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 bracesv** ({and}). Following is the syntax for defining an object.

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.

objectName.propertyName

### Example: Object Initializers

var person = {

firstname:"Tom",

lastname:"Hanks",

func:function(){return "Hello!!"},

};

//access the object values

console.log(person.firstname)

console.log(person.lastname)

console.log(person.func())

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.

AD

## 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.

var obj\_name = new Object();

obj\_name.property = value;

OR

obj\_name["key"] = value

Following is the syntax for accessing a property.

Object\_name.property\_key

OR

Object\_name["property\_key"]

### Example

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

var myCar = new Object();

myCar.make = "Ford";

console.log(myCar["model"])

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

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**

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 −

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

# ES6 - Classes

**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.

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.)

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.

### 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.

AD

## 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

AD

s