ES6

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com


https://www.facebook.com/tutorialspointindia


https://twitter.com/tutorialspoint

About the Tutorial

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

This tutorial introduces you to ES6 implementation in JavaScript.

Audience

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

Prerequisites

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

Disclaimer & Copyright

Table of Contents

1. ES6 – Overview

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

This tutorial introduces you to ES6 implementation in JavaScript.

JavaScript

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

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

- Support for constants
- Block Scope
- Arrow Functions
- Extended Parameter Handling
- Template Literals
- Extended Literals
- Enhanced Object Properties
- De-structuring Assignment
- Modules
- Classes
- Iterators
- Generators
- Collections
- New built in methods for various classes
- Promises

2. ES6 – Environment

In this chapter, we will discuss the setting up of the environment for ES6.

Local Environment Setup

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

Text Editor

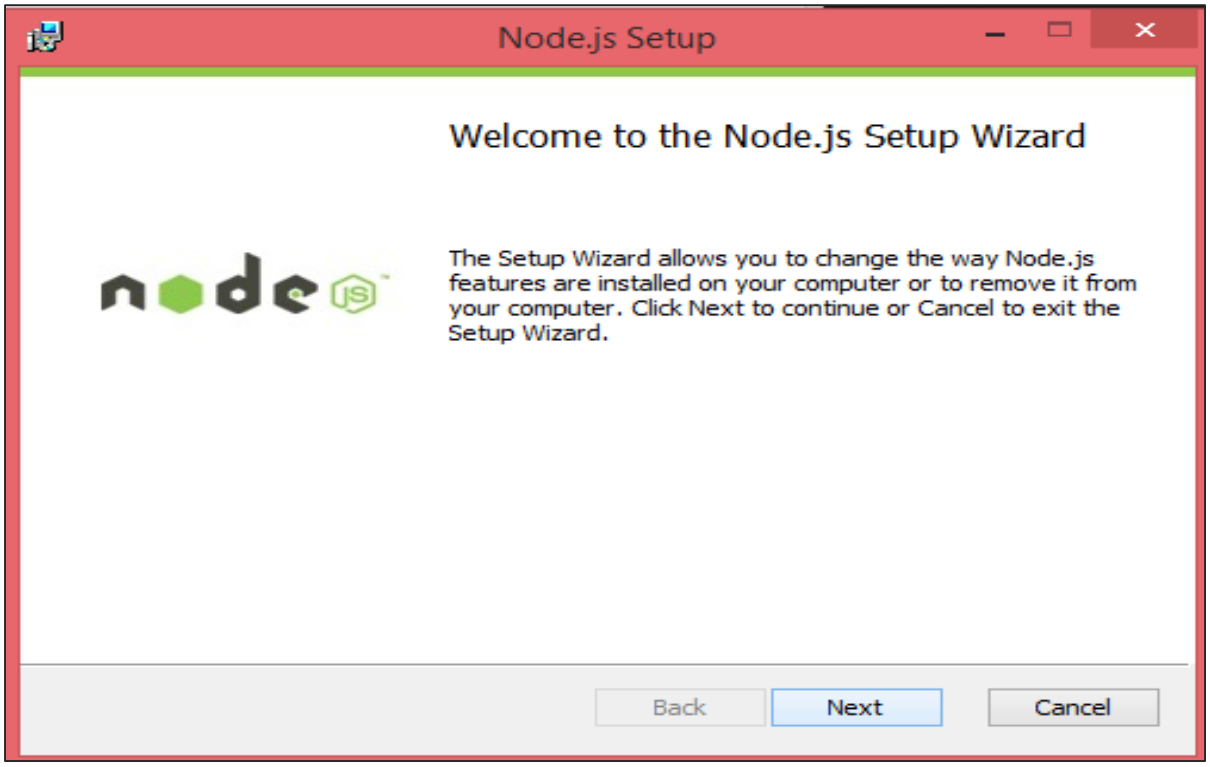
The text editor helps you to write your source code. Examples of few editors include Windows Notepad, Notepad++, Emacs, vim or vi etc. Editors used may vary with the operating systems. The source files are typically named with the **extension .js**.

Installing Node.js

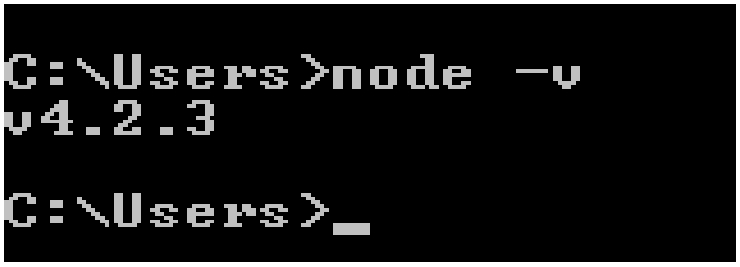
Node.js is an open source, cross-platform runtime environment for server-side JavaScript. Node.js is required to run JavaScript without a browser support. It uses Google V8 JavaScript engine to execute the code. You may download Node.js source code or a pre-built installer for your platform. Node is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

Installation on Windows

Download and run the **.msi installer** for Node.



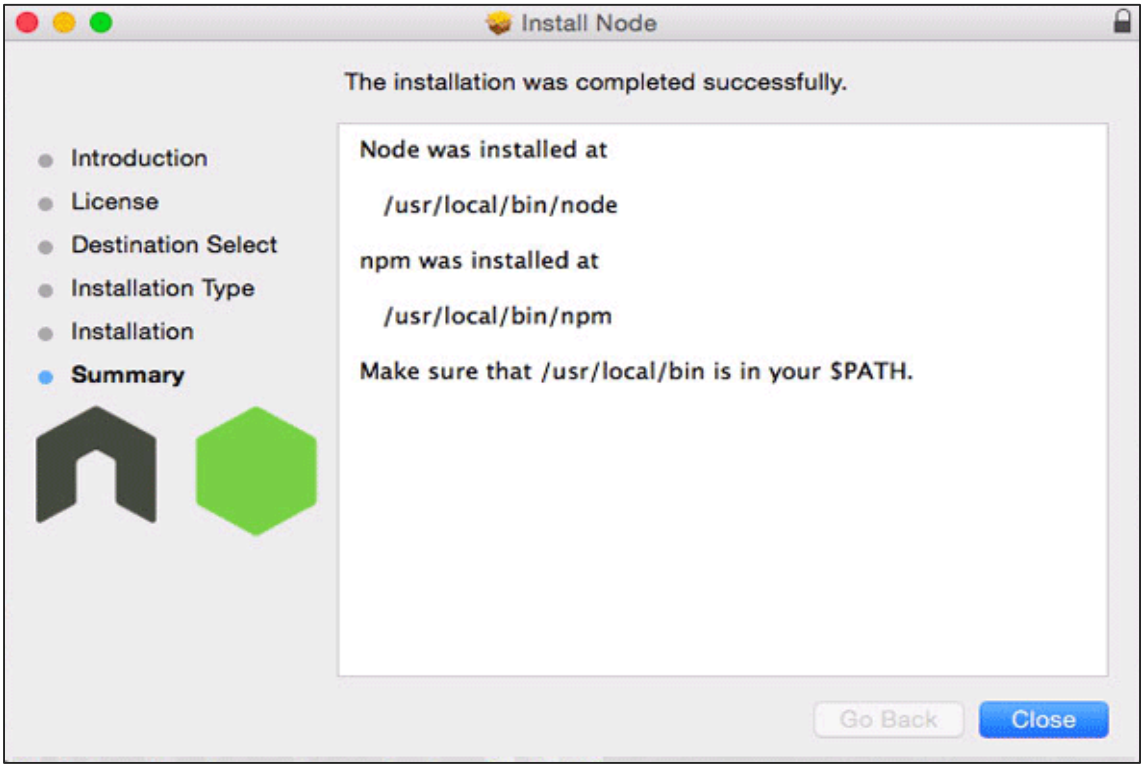
To verify if the installation was successful, enter the command ***node –v*** in the terminal window.



Installation on Mac OS X

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

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



Installation on Linux

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

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

Integrated Development Environment (IDE) Support

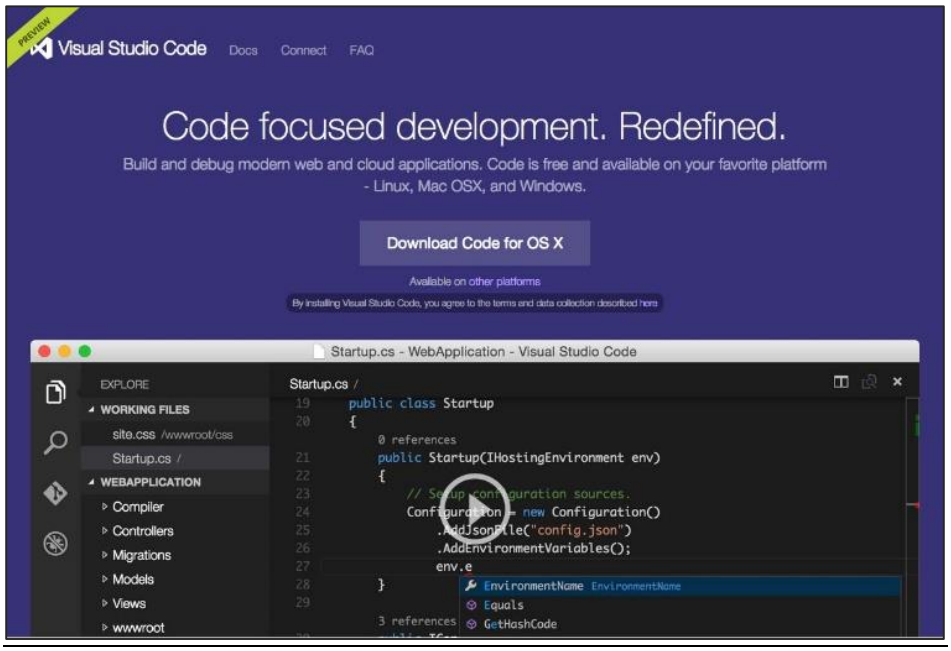
JavaScript can be built on a plethora of development environments like Visual Studio, Sublime Text 2, WebStorm/PHPStorm, Eclipse, Brackets, etc. The Visual Studio Code and Brackets IDE is discussed in this section. The development environment used here is Visual Studio Code (Windows platform).

Visual Studio Code

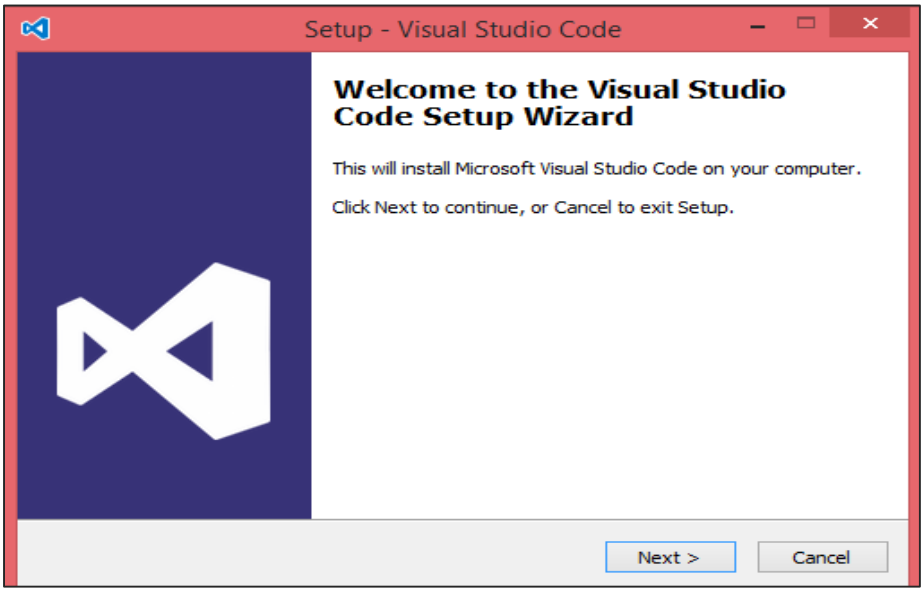
This is open source IDE from Visual Studio. It is available for Mac OS X, Linux, and Windows platforms. VScode is available athttps://code.visualstudio.com

Installation on Windows

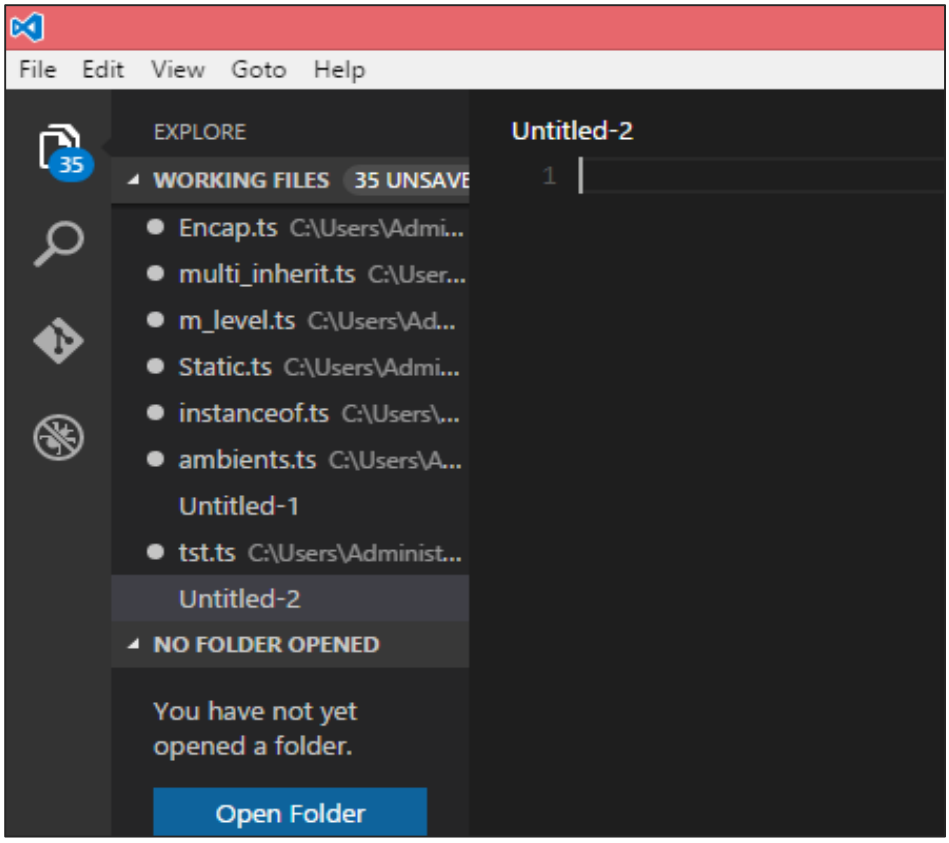
Download Visual Studio Code for Windows.



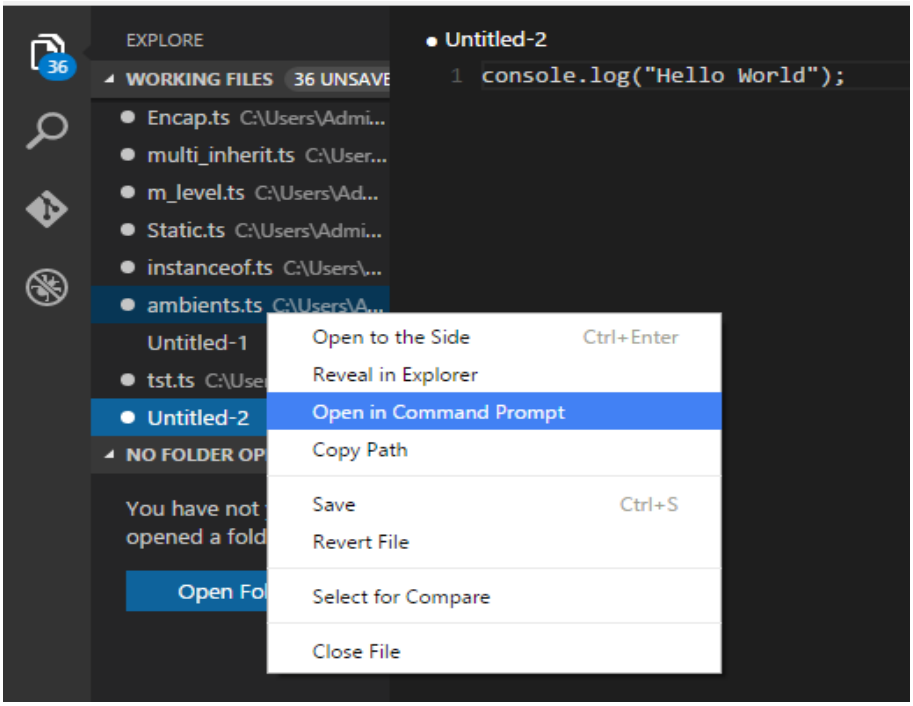
Double-click on VSCodeSetup.exe  to launch the setup process. This will only take a minute.



Following is the screenshot of the IDE.



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



Installation on Mac OS X

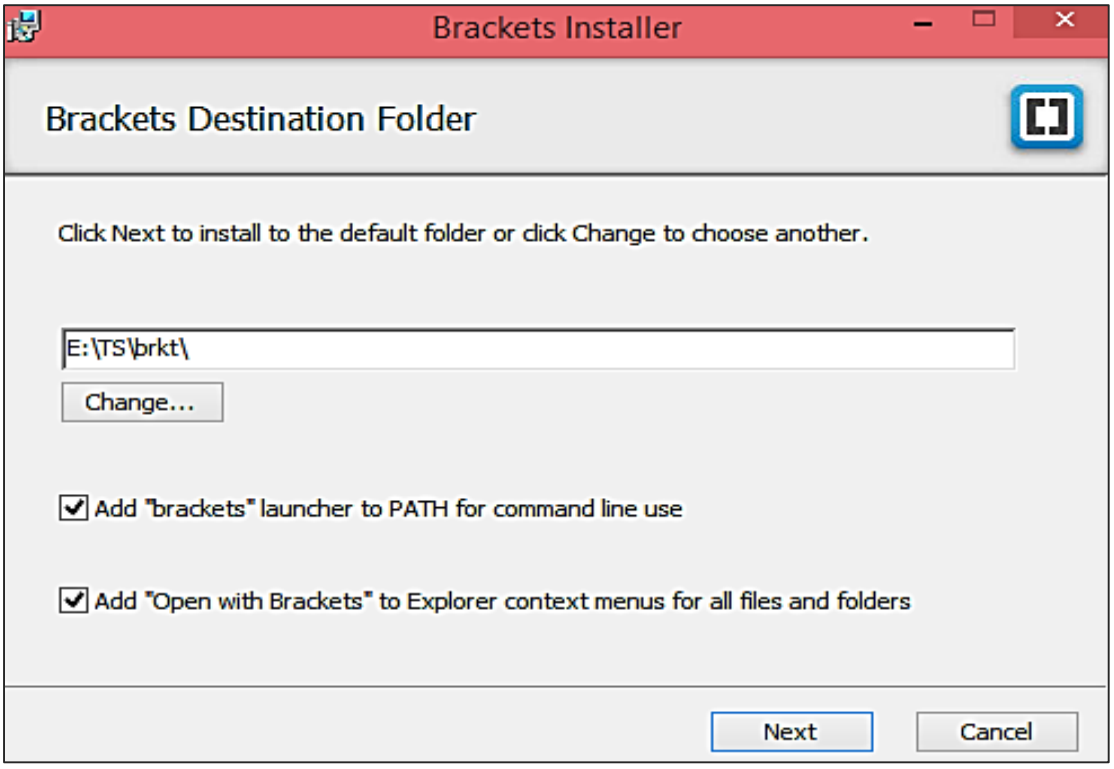
Visual Studio Code's Mac OS X specific installation guide can be found at https://code.visualstudio.com/Docs/editor/setup

Installation on Linux

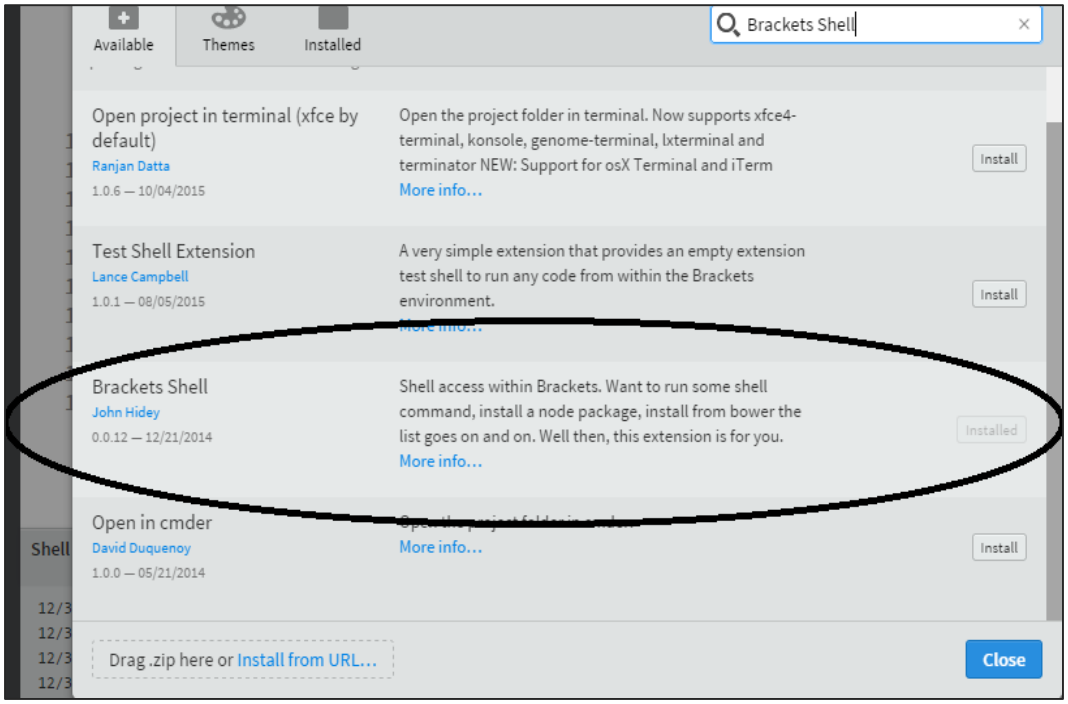
Linux specific installation guide for Visual Studio Code can be found at https://code.visualstudio.com/Docs/editor/setup

Brackets

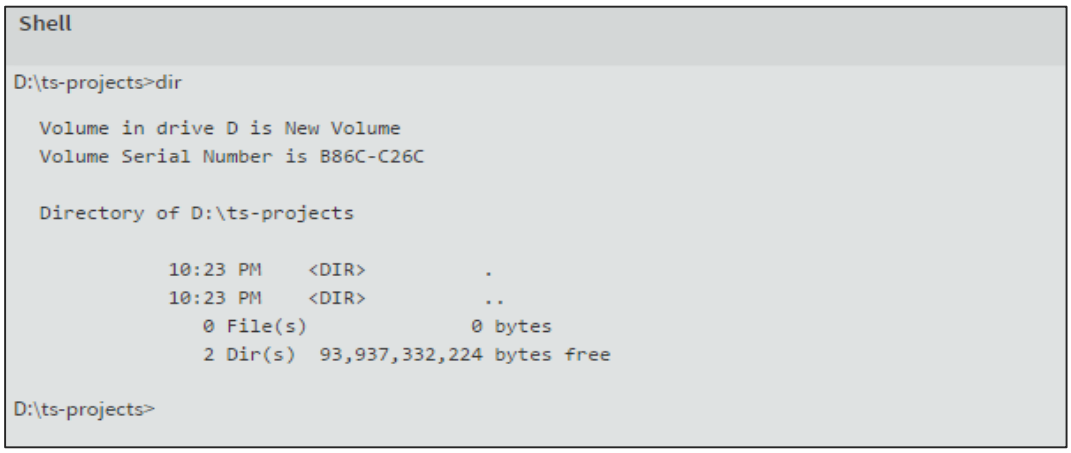
Brackets is a free open-source editor for web development, created by Adobe Systems. It is available for Linux, Windows and Mac OS X. Brackets is available at http://brackets.io



You can run DOS prompt/Shell within Brackets itself by adding one more extension Brackets Shell.



Upon installation, you will find an icon of shell on the right hand side of the editor  . Once you click on the icon, you will see the shell window as shown in the following screenshot.



You are all set!!!

3. ES6 – Syntax

Syntax defines the set of rules for writing programs. Every language specification defines its wn syntax.

A JavaScript program can be composed of:

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

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

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.

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.

Executing the Code

We shall use Node.js to execute our code.

Step 1: Save the file as Test.js

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

Step 3: Select Open in Command Prompt option.

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

```
node Test.js
```

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

```
Hello World
```

Node.js and JS/ES6

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

- **For Shipping**: These are features that V8 considers stable.
- **Staged Features**: These are almost completed features but not considered stable by the V8 team.
- **In Progress**: These features should be used only for testing purposes.

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

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

The Strict Mode

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

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

```
// 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-script mode. All statements within the function will be executed in the Strict Mode.

ES6 and Hoisting

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

Variable Hoisting and Function Hoisting are explained in the subsequent chapters.

4. ES6 – Variables

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 outside test() 100
```

ES6 defines a new variable scope - The Block scope.

The Let and Block Scope

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

```
"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.
- 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.

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.

5. ES6 – Operators

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.

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 2
++	Increments the value of the variable by one	a++ is 11
--	Decrements the value of the variable by one	A- - is 9

Example: Arithmetic Operators

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

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

```
Sum: 12
```

```
Difference: 8
Product: 20
Quotient : 5
Remainder: 0
Value of num1 after increment: 11
Value of num2 after decrement: 1
```

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.

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 True
!=	Not equal	(A!=B) is True

Example

```
var num1 = 5;
var num2 = 9;
console.log("Value of num1: " + num1);
console.log("Value of num2 :" + num2);
var res = num1 > num2;
console.log("num1 greater than num2: " + res);
res = num1 < num2;
console.log("num1 lesser than num2: " + res);
res = num1 >= num2;
console.log("num1 greater than or equal to  num2: " + res);
```

```
res = num1 <= num2;
console.log("num1 lesser than or equal to num2: " + res);
res = num1 == num2;
console.log("num1 is equal to num2: " + res);
res = num1 != num2;
console.log("num1 not equal to  num2: " + res);
```

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

```
Value of num1: 5
Value of num2 :9
num1 greater than num2: false
num1 lesser than num2: true
num1 greater than or equal to num2: false
num1 lesser than or equal to num2: true
14 num1 is equal to num2: false
16 num1 not equal to num2: 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.

Operator	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

Example

```
var avg = 20;
var percentage = 90;
console.log("Value of avg: " + avg + " ,value of percentage: " + percentage);
var res = ((avg > 50) && (percentage > 80));
console.log("(avg>50)&&(percentage>80): ", res);
var res = ((avg > 50) || (percentage > 80));
console.log("(avg>50)||(percentage>80): ", res);
var res = !((avg > 50) && (percentage > 80));
console.log("!((avg>50)&&(percentage>80)): ", res);
```

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

```
Value of avg: 20 ,value of percentage: 90
(avg>50)&&(percentage>80):  false
(avg>50)||(percentage>80):  true
!((avg>50)&&(percentage>80)):  true
```

Short-circuit Operators

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

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

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

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

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

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

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

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

Bitwise Operators

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

Operator	Usage	Description
Bitwise AND	a & b	Returns a one in each bit position for which the corresponding bits of both operands are ones
Bitwise OR	a | b	Returns a one in each bit position for which the corresponding bits of either or both operands are ones
Bitwise XOR	a ^ b	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones
Bitwise NOT	~ a	Inverts the bits of its operand
Left shift	a << b	Shifts a in binary representation b (< 32) bits to the left, shifting in zeroes from the right
Sign-propagating right shift	a >> b	Shifts a in binary representation b (< 32) bits to the right, discarding bits shifted off
Zero-fill right shift	a >>> b	Shifts a in binary representation b (< 32) bits to the right, discarding bits shifted off, and shifting in zeroes from the left

Example

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

```
result = (a ^ b);
console.log("(a ^ b) => ", result);
result = (~b);
console.log("(~b) => ", result);
result = (a << b);
console.log("(a << b) => ", result);
result = (a >> b);
console.log("(a >> b) => ", result);
```

Output

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

Assignment Operators

The following table summarizes Assignment operators.

Sr. No.	Operator and 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 ^=

Example

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

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

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

Miscellaneous Operators

Following are some of the miscellaneous operators.

The negation operator (-)

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

```
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
```

Type Operators

typeof operator

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

Type	String Returned by typeof
Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"

The following example code displays the number as the output.

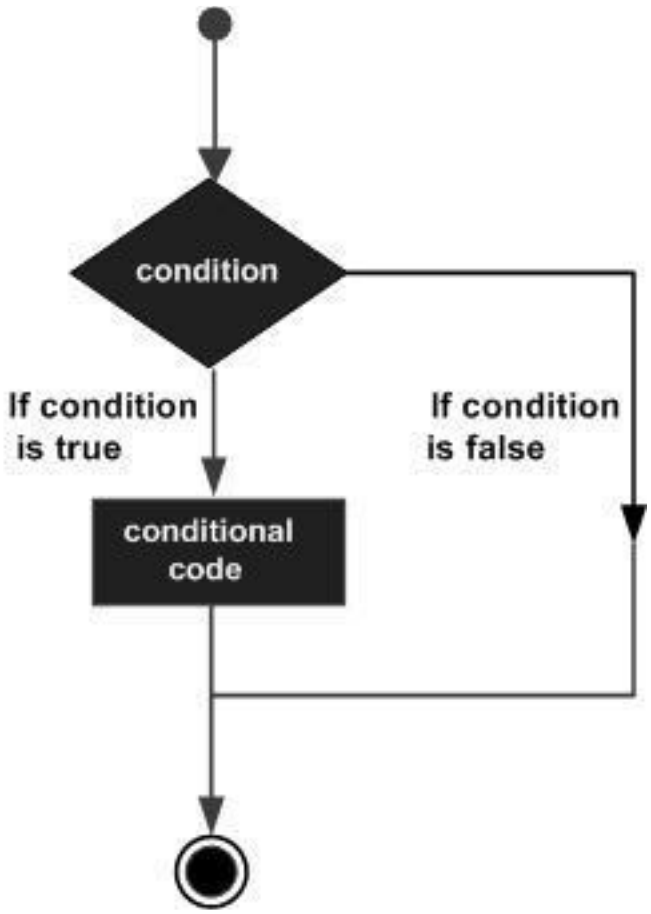
```
var num=12
console.log(typeof num); //output: number
```

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

```
number
```

6. ES6 – Decision Making

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



Conditional constructs in JavaScript are classified in the following table.

Statement	Description
if statement	An 'if' statement consists of a Boolean expression followed by one or more statements
if...else statement	An 'if' statement can be followed by an optional 'else' statement, which executes when the Boolean expression is false

The else.. if ladder / nested if statements	You can use one 'if' or 'else if' statement inside another 'if' or 'else if' statement(s)
switch statement	A 'switch' statement allows a variable to be tested for equality against a list of values

The if Statement

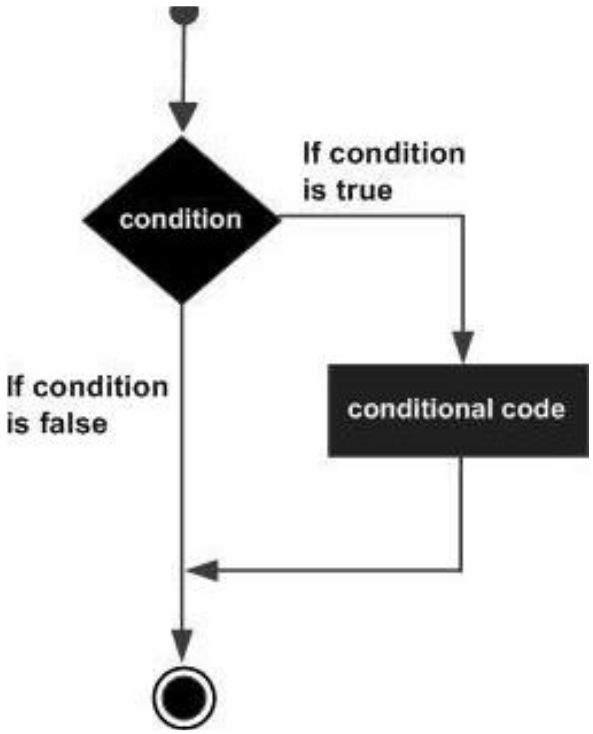
The 'if…else' construct evaluates a condition before a block of code is executed.

Following is the syntax.

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

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

Flowchart



Example

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

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

```
number is positive
```

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

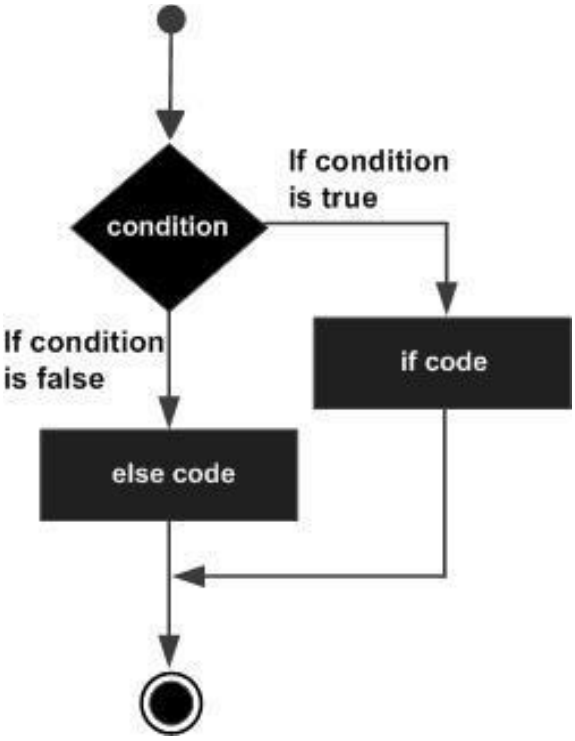
The if…else Statement

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

Following is the syntax.

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

Flowchart



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

Example: Simple if…else

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

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

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

```
Even
```

The else…if Ladder

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

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

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

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

Example: else…if ladder

```
var num=2
if(num > 0)
{
console.log(num+" is positive")
}
else if(num < 0)
```

```
{
console.log(num+" is negative")
}
else
{
console.log(num+" is neither positive nor negative")
}
```

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

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

```
2 is positive
```

The switch…case Statement

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

Following is the syntax.

```
switch(variable_expression)

   {
      case constant_expr1:
               {
            //statements;
            break;
               }
      case constant_expr2:
            {

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

```
            {
                  //statements;
                  break;
            }
}
```

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

The following rules apply to a switch statement:

- There can be any number of case statements within a switch.
- The case statements can include only constants. It cannot be a variable or an expression.
- The data type of the variable_expression and the constant expression must match.
- Unless you put a break after each block of code, the execution flows into the next block.
- The case expression must be unique.
- The default block is optional.

End of ebook preview

If you liked what you saw…

Buy it from our store @ **https://store.tutorialspoint.com**