

JavaScript & Node.js

JavaScript

What is JavaScript?

JavaScript is one of the most popular and widely used programming language in the world right now.

It's growing faster than any other programming languages and big companies like Netflix, Walmart, and PayPal build entire application around JavaScript.

What we can do with JavaScript?

For a long time, JavaScript was only used in browser to build interactive web pages.

Some developers refer to JavaScript as a toy language but those days are gone because of huge community support and investments by large companies like Facebook and Google.

Using JavaScript we can develop Web or Mobile applications as well as Real-time Networking applications like chats and video streaming services, Command-line tools or even gaming applications also.

Where does JavaScript code run?

JavaScript was originally designed to run only in browsers so every browser has what we call a JavaScript engine that can execute JavaScript code.

For example, the JavaScript engines in Firefox and Chrome are SpiderMonkey and V8.

In 2009, a very clever engineer called Ryan Dahl took the open-source JavaScript engine in chrome and embedded it inside a c++ program.

Node is a c++ program that includes Google's V8 JavaScript engine.

Now with this we can run JavaScript code outside of a browser.

So we can pass our JavaScript code to Node for execution and this means with JavaScript we can build the Back-end for our Web and Mobile applications.

Hence our JavaScript code can be run inside of browser or in a Node.

Browser and Node provide a runtime environment for our JavaScript code.

JavaScript v/s ECMAScript:

ECMAScript is just a specification.

JavaScript is a Programming Language that confirms to this specification.

So we have this organization called ECMA which is responsible for defining standards they take care of this ECMAScript specification.

The first version of ECMAScript was released in 1997. Then starting from 2015 ECMA has been working on annual releases of new specification.

In 2015, they released ECMAScript 2015 which is also called ECMAScript version 6 or ES6 for short.

This specification defined many new features for JavaScript.

Getting Started with JavaScript:

JavaScript is a scripting language used both on client-side and server-side that allows you to make web pages interactive.

Using JavaScript, we can develop Real-Time Networking Apps, Web/Mobile Apps, Command-Line Tools, Gaming Apps etc.

JavaScript is backward compatible that means the features which came before 20 years also going to work in the current version of JavaScript.

JavaScript is not forward compatible that means the features which we have in the current version will not work in the previous versions which released for JavaScript.

To make JavaScript work for all features irrespective of the versions we have a tool called Babel which is a JavaScript compiler can make it your code to work with any version.

Without checking whether the current version is going to support the features which you are using in the code.

Fundamental Concepts:

In the fundamental concepts of JavaScript we are going learn the following:

- Variable declaration
- Data Types
- Array and Objects
- Functions
- Conditional Statements
- Looping Concept

Variable Declaration:

in Programming, we use a variable to store data temporarily in a computer's memory.

So we store our data somewhere and give that memory location a name and with this name we can read data at the given location in the future.

Example:

```
1  var Name = "Tushar";  
2  console.log(Name);
```

Variables can store some information we can use that information later and also we can change that information later.

Rules for naming variables:

- Variable cannot be a reserved keyword.
- Variable should be meaning full.
- Variable cannot start with a number (1name).
- Variable cannot contain a space or hyphen (-).
- Variable are case-sensitive.

Data Types in JavaScript

We have two categories of types in JavaScript:

- 1) Primitives/Value Types.
- 2) Reference Types

In the category of Primitive Type we have,

1. String
2. Number
3. BigInt
4. Boolean
5. undefined
6. null

Types	Description	Remarks
Boolean	True or false 1 or 0	
Number	Any integer or a floating-point numeric value.	
BigInt	Large, integer even beyond the safe integer limit for number.	A BigInt is created by appending n to the end of an integer
String	The sequence of characters delimited by either single or double-quotes.	
Null	Special primitive-type that has only one value, null.	
undefined	A primitive type that has only once a value undefined.	The undefined keyword is the value assigned to a variable that wasn't initialized.

Example Program for all Primitive Data Types:

```
1  //boolean
2  var isApproved = true;
3
4  //number
5  var are = 20;
6
7  //BigInt
8  var bigInt = 123490597985685666367585n;
9
10 //string
11 var firstName = "Tushar";
12
13 //null
14 var selectedColor = null;
15
16 //undefined
17 var playerName = undefined;
```

How Do Primitive Types Behave?

Primitive types are always assigned, the value is copied. To practically see the behavior, let's look at the example below.


```
1  var name1 = "Tushar";
2  var name2 = name1;
3
4  console.log(name1); //Tushar
5  console.log(name2); //Tushar
6
7  name2 = "Mohit";
8
9  console.log(name1); //Tushar
10 console.log(name2); //Mohit
```

As you can see the variable name1 and name2 are completely separate from each other, and you can change the value in name2 without affecting name1 and vice versa.

Using the Typeof Operator:

```
1  console.log(typeof(true)); //boolean
2  console.log(typeof(30)); //number
3  console.log(typeof(BigInt)); //bigInt
4  console.log(typeof("Tushar")); //string
5  console.log(typeof(null)); //object
6  console.log(typeof(undefined)); //undefined
```

Why null returns object?

OK, you have probably run the code sample above and wondered why the null data-type returns object. This is the tricky part of the language, even I was confused at first but no worries, I will answer that here.

The truth, it is a bug and it has been acknowledged by TC39, the committee that maintains the JavaScript language. One reason this bug wasn't fixed is because the proposed fix broke a number of existing sites. Thus, the error remained.

Now, that we have an idea of why it behaves that way, what's the best way to determine if a value is null? To do this, you can compare the null directly against its value.

See the example:

```
1 var myObj = null;
2 console.log(myObj === null); //true
```

Difference Between =, == And === in JavaScript:

= is used for assigning value to a variable in JavaScript.

== is used for comparison between two variables irrespective of the data type of variable.

=== is used for comparison between two variables but this will check strict type, which means it checks data type and compares two values.

Reference Types:

If you are coming from another language like C#, Java, or C++, reference types are the closest thing to classes. Reference values are instances of reference types. Moreover, reference types do not store the object directly into the variable to which it is assigned. Thus, it holds a reference to the location in memory where the object exists.

Make Objects

There are various ways to create objects in JavaScript. And these are: using a new operator, object literal, and a constructor function.

Example:

```
1  var myObj = new Object(); //using the new operator
2  console.log(typeof(myObj)); //object
3
4  var myObj2 = {}; //using the object literal ({}))
5  console.log(typeof(myObj2));
6
7  //using the constructor function
8  function MyFun(property1, property2)
9  {
10     this.property1=property1;
11     this.property2=property2;
12 }
13
14 var myObj3 = new MyFun("JavaScript", 23);
15 console.log(typeof(myObj3)); //object
```

How Do Objects Behave?

Example:

```
1  var Obj1 = {  
2    |    favLanguage: "JavaScript"  
3  };  
4  Obj1.favLanguage = "JavaScript";  
5  
6  var Obj2 = Obj1;  
7  
8  console.log(Obj1);  
9  console.log(Obj2);  
10  
11  Obj1.favLanguage = "Python";  
12  
13  console.log(Obj1);  
14  console.log(Obj2);
```

As you can see, we have declared a new object using the variable Obj1 and assigned it to Obj2. As a result, both of them are referencing the same object. That's why any change to either one of them have the same effects.

Difference between Primitives types and Reference types:

The main difference between the two is that primitive types directly contain their values.

Meaning, when you assign a primitive value to a variable, the value is copied into that variable.

While reference types don't store the object directly into the variable to which it is assigned.

Instead, it holds the reference to the location in memory where the object exists.

Array and Object in JavaScript

An array in JavaScript is a collection of multiple values stored in a single variable.

Elements in array are enclosed within square brackets and each elements is separated using a comma.

Each element in an array is identified using an index number.

The index number is always a positive or a negative integer and starts with 0.

JavaScript array can also hold variables of different types.

Example:

Array containing same type of values:

```
1 const colors = ["blue","red","green","yellow","black","white"];
```

Accessing Array Items:

```
console.log(colors[0]);  
console.log(colors[1]);  
console.log(colors[2]);  
console.log(colors[3]);  
console.log(colors[4]);  
console.log(colors[5]);
```

Finding Array Length:

```
console.log(colors.length);
```

The length of an array is 1 number greater than the maximum index number. Since the index number starts with 0.

If the maximum index of colors array is 6 means then the length of colors array is 7.

Array containing different types of values:

Defining an Array called person:

```
1 const person = [1,"Tushar",true,["3130, Sector 40-D, Chandigarh"],92.67];
```

Accessing Array Items:

```
console.log(person[0]);  
console.log(person[1]);  
console.log(person[2]);  
console.log(person[3]);  
console.log(person[4]);  
console.log(person[5]);
```

Finding Array Length:

```
console.log(person.length);
```

Array Function:

1) Push function:

Push function adds a new element to the end of the array.

```
const Bikes=['Bajaj','Honda','Royal_Enfield','Yamaha','Ducati'];  
Bikes.push('BMW');  
console.log(Bikes);
```

2) Pop function:

```
const Bikes=['Bajaj','Honda','Royal_Enfield','Yamaha','Ducati'];  
Bikes.pop();  
console.log(Bikes);
```


3)Unshift function:

Unshift function is similar to push function, but add a new element to the start of an array.

```
const Bikes=['Bajaj','Honda','Royal_Enfield','Yamaha','Ducati'];  
Bikes.unshift('BMW');  
console.log(Bikes);
```

4)Shift function:

Shift function is similar to pop function, but is removes an element from the start of an array.

```
const Bikes=['Bajaj','Honda','Royal_Enfield','Yamaha','Ducati'];  
Bikes.shift();  
console.log(Bikes);
```

5)Splice function:

Splice function add new element to an array at specific location.

```
const Bikes=['Bajaj','Honda','Royal_Enfield','Yamaha','Ducati'];  
Bikes.splice(3,0,"BMW","HJHG");  
console.log(Bikes);
```

6)Slice function:

Slice function slices out a part from the original array and creates a new array. It doesn't remove elements from the original array.

```
const Bikes=['Bajaj','Honda','Royal_Enfield','Yamaha','Ducati'];
Bikes1 = Bikes.slice(1,4);
console.log(Bikes1);
console.log(Bikes);
```

7)forEach function:

forEach function is used to loop through an array.

```
const Bikes=['Bajaj','Honda','Royal_Enfield','Yamaha','Ducati'];
Bikes.forEach(function(item,index){console.log(item,index)});
```

Here the function is called as callback function.

Callback function is a function passed an argument to another function.

8)Filter function:

Filter function creates a new array from an existing array with elements that qualifies the given condition.

```
const Bikes=['Bajaj','Honda','Royal_Enfield','Yamaha','Ducati'];
console.log(Bikes.filter(function(item)
{
    return item.startsWith("R");
})));
```

Functions in JavaScript

Functions are one of the fundamental building blocks in JavaScript.

A function is basically a set of statements that performs a task or calculates a value.

Advantages of JavaScript function:

There are mainly two advantages of JavaScript function.

- **Code reusability:** We can call a function several times so it saves coding.
- **Less coding:** It makes our program compact. We don't need to write many lines of code each time to perform a common task.

Example:

```
function greetings()  
{  
  console.log("Hi, Good Morning");  
}  
greetings();
```

```
function greetings(name)
{
  console.log("Hi " + name + " Good Morning");
}
greetings("Tushar");
greetings("Mohit");
```

Conditional Statements in JavaScript

Typically, you want to perform an action when a condition is True and another action when the condition is False.

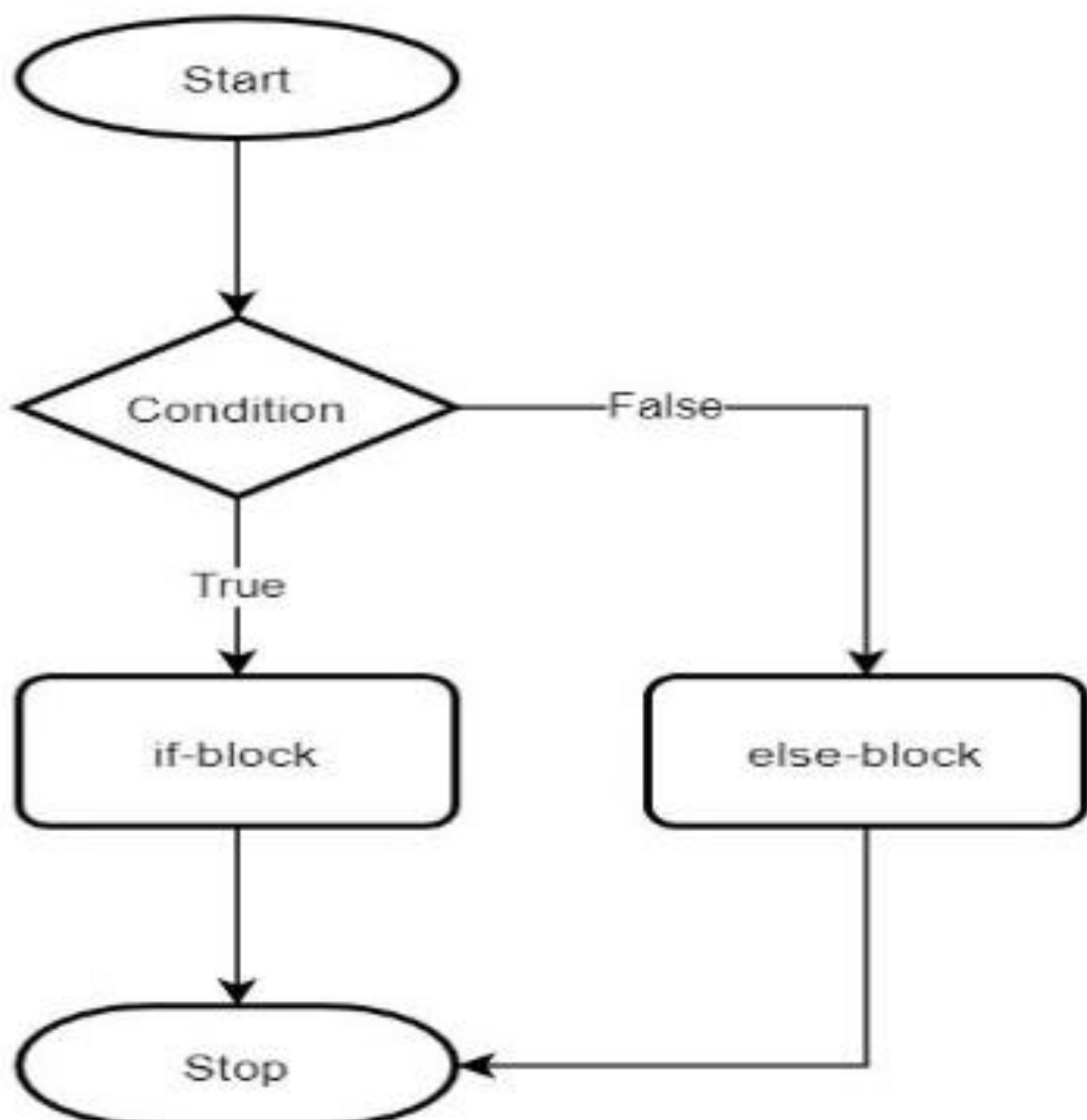
To do so, you use the if....else statement.

Syntax of if... else statement:

```
if(condition)
{
  if-block
}
else
{
  else-block
}
```

In this syntax, the if... else will execute the if-block if the condition evaluates to True, Otherwise, it'll execute the else-block.

The following flowchart illustrates the if... else statement:



Example:

```
age = 19;
if(age >= 18)
{
    console.log("You are eligible to vote");
}
else
{
    console.log("you are nor eligible to vote")
}
```

```
age = prompt("Enter the age of voter: " );
if(age >= 18)
{
    console.log("You are eligible to vote");
}
else
{
    console.log("you are nor eligible to vote")
}
```

In this example, if you enter the age of voter with a number less than 18 then you'll see the message that "You're not eligible to vote".

If you want to check multiple conditions and perform an action according, you can use the if... else if... else statement.

Syntax for if... else if... else statements:

```
if(if-condition)
```

```
{
```

```
    if-block
```

```
}
```

```
else if(elseif-condition1)
```

```
{
```

```
    else-if-block1
```

```
}
```

```
else if(elseif-condition2)
```

```
{
```

```
    else if-block2
```

```
}
```

```
....
```

```
else
```

```
{
```

```
    else-block
```

```
}
```

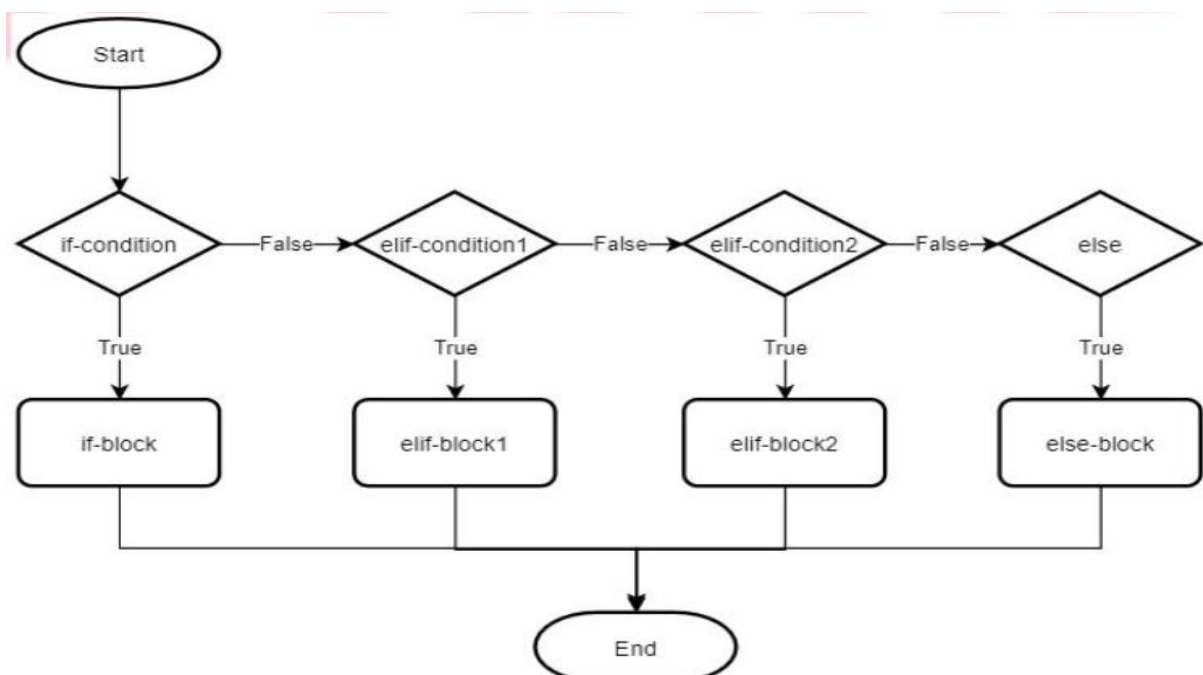

The if...else if...else statement checks each condition (if-condition, else if-condition1, else if-condition2,...) in the order that they appear in the statements until it finds the one that evaluates to True.

When the if...else if...else statement finds one, it executes the statement that follows the condition and skips testing the remaining conditions.

If no condition evaluates to True, the if...else if...else statement executes the statement in the else branch.

Note that the else block is optional. If you omit it and no condition is True, the statement does nothing.

The following flowchart illustrates the if... else if... else statement:



Example:

```
age = 15;
if(age < 5)
{
    ticket_price = 5;
}
else if(age < 16)
{
    ticket_price = 10;
}
else
{
    ticket_price = 18;
}
console.log("You have to pay "+ ticket_price + " rupees for the ticket");
```

```
age = prompt("Enter the age for checking the ticket price");
if(age < 5)
{
    ticket_price = 5;
}
else if(age < 16)
{
    ticket_price = 10;
}
else
{
    ticket_price = 18;
}
console.log("You have to pay "+ ticket_price + " rupees for the ticket");
```

Ternary Operator in JavaScript

A ternary operator evaluates a condition and executes a block of code based on the condition.

The conditional (ternary) operator is the only JavaScript operator that takes three operands:

- a condition followed by a question mark (?),
- then an expression to execute if the condition is truthy followed by a colon (:), and
- finally the expression to execute if the condition is falsy.

As the ternary operator takes three operands, hence, the name called as ternary operator. It is also known as a conditional operator.

Example:

Without ternary operator:

```
let age = 4;
let drink;
if(age >= 5)
{
    drink = "coffee";
}
else
{
    drink = "milk";
}
console.log(drink);
```

With using a ternary operator:

```
let age = 3;  
let drink = age >= 5 ? "coffee" : "milk";  
console.log(drink);
```

The syntax of ternary operator in JavaScript:

condition ? expression1 : expression2

The ternary operator evaluates the test condition.

- If the condition is true, expression1 is executed.
- If the condition is false, the expression2 is executed.

Strings in JavaScript

A string is a sequence of one or more characters that may consist of letters, numbers, or symbols.

Each character in JavaScript string can be accessed by an index number, and all strings have methods and properties available to them.

Each of the characters in a string corresponds to an index number, starting with 0.

To demonstrate, we will create a string with the value "Tushar".

T u s h a r

0 1 2 3 4 5 6

The first character in the string is S, which corresponds to the index 0. The last character is r, which corresponds to 6. Note that in JavaScript white space characters also have an index.

Being able to access every character in a string gives us a number of ways to work with and manipulate strings.

Example:

```
// String indexing

let firstName = "Shastri";

console.log(firstName.length);
console.log(firstName[0]);
console.log(firstName[1]);
console.log(firstName[2]);
console.log(firstName[3]);
console.log(firstName[4]);
console.log(firstName[5]);
console.log(firstName[6]);
```

Converting to Upper or Lower Case:

Two built-in methods in JavaScript for strings are: `toUpperCase()` and `toLowerCase()` are helpful ways to format text and make textual comparisons in JS.

toUpperCase() will convert all character to uppercase characters.

```
console.log("i love javascript".toUpperCase());
```

```
PS E:\Work\ts> node test.js  
I LOVE JAVASCRIPT
```

toLowerCase() will convert all characters to lowercase characters.

```
js  
console.log("I LOVE JAVASCRIPT".toLowerCase());
```

```
PS E:\Work\ts> node test.js  
i love javascript  
PS E:\Work\ts> node test.js
```

Splitting Strings:

JavaScript has a very useful method for splitting a string by a character and creating a new array out of the sections.

We will use split() method to separate the array by a whitespace character, represented by “ ”.

```
tjs > ...  
const originalString = "I Love JavaScript";  
const splitString = originalString.split(" ");  
console.log(splitString);
```

```
PS E:\Work\ts> node test.js  
[ 'I', 'Love', 'JavaScript' ]  
PS E:\Work\ts>
```

Trimming Whitespace:

The JavaScript trim() method removes white space from both ends of a string, but not anywhere in between. Whitespace can be tabs or space.

The trim() method is a simple way to perform the common task of removing excess whitespace.

```
ts> ...  
const abc = "    I Love JavaScript    ";  
const trimmed = abc.trim();  
console.log(trimmed);
```

```
PS node test.js  
I Love JavaScript  
PS E:\Work\ts>
```

Finding and Replace String Values:

We can search a string for a value, and replace it with a new value using the replace() method.

The first parameter will be the value to be found, and the second parameter will be the value to replace it with.

```
ts> ...  
const abc = "I Love JavaScript";  
const newStr = abc.replace("JavaScript", "Python");  
console.log(newStr);
```

```
PS E:\Work\ts> node test.js  
I Love Python  
PS E:\Work\ts>
```

Nested if Statement in JavaScript

The JavaScript if statement itself is a statement and it can be used within another JavaScript if structure.

When a JavaScript statement may also contain an 'else' part.

The JavaScript if statement that contains another statement is referred to as outer JavaScript is statement while the JavaScript if statement used inside if (nested if statement) is referred to as inner if statement.

The nested if statement will be executed only if the outer if statement (that contains the nested if statement) is true.

For example, to check whether the value of variable 'a' is greater than the value of variable 'b' and 'c'.

Example:

```
let num = 19;
let num2 = +prompt("Guess a number");

if(num2 === num)
{
    console.log("Your guess is right");
}
else
{
    if(num2 < num)
    {
        console.log("too low !!!");
    }
    else{
        console.log("too high !!!");
    }
}
```

Switch Statement in JavaScript

Although the nested if-else statement is used for multiple selection but it becomes complicated for large number of choices.

The switch statement is used as a substitute of nested if-else statement.

It is used when multiple choices are given and one choice is to be selected.

For example, this statement is used for menu selection.

The switch statement contains only one expression at its beginning and multiple case (each case contains a set of statements) with its body.

One of the case (or block of statements) is selected for execution depending upon the value returned by this expression.

The general syntax of switch statement is:

switch (expression)

{

Case label-1:

Set of statement-1

Break;

Case label-2:

Set of statement-2

Break;

.

.

.

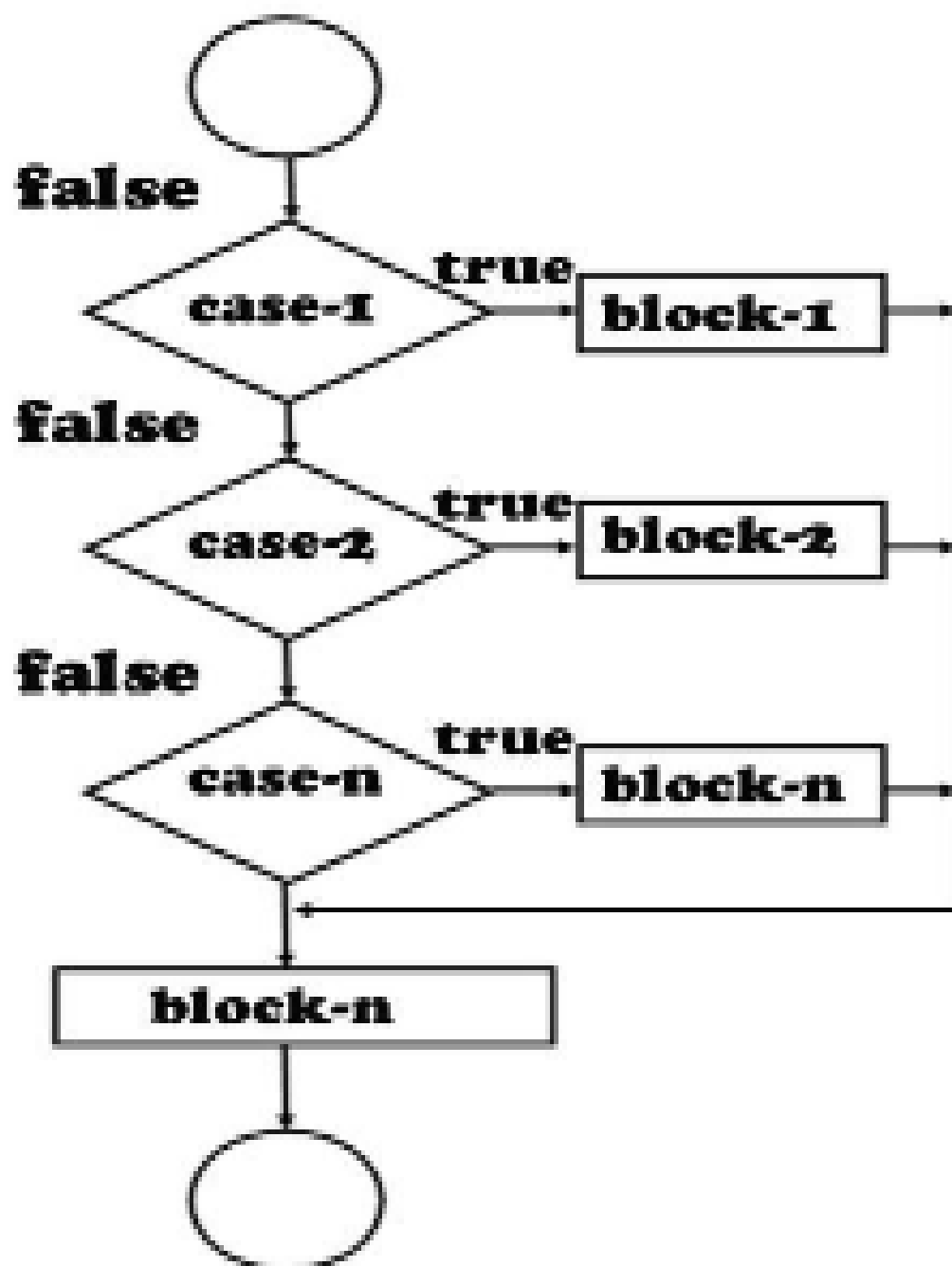
.

Default:

Set of statements-n

}

Flowchart of Switch Statement:



The switch statement can be used within another switch selection statement, it is called the nested switch statement.

The case constant value of the inner and outer switch statement may contain values.

Break Statement:

The 'break' statement is used to exit from the body the switch statement (or loop structure).

After executing this statement, execution control is transferred to the statement that comes immediately after the switch statement (or loop structure).

In the switch statement, the break statement is normally used at the end of the statement in each case.

If all break statements are omitted from the switch statement, then the statement of all the remaining cases that comes after the matched case are also executed in sequential order.

Example:

```
st.js > ...
let day = 2;
switch(day)
{
  case 0:
    console.log("Sunday");
    break;
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  case 4:
    console.log("Thursday");
    break;
  case 5:
    console.log("Friday");
    break;
  case 6:
    console.log("Saturday");
    break;
  default:
    console.log("Invalid Day");
}
```

- PS E:\Work\ts> node test.js
Tuesday
- PS E:\Work\ts>

Iterative Statement in JavaScript

In JavaScript, there are three types of loops:

- The while loop.
- The for loop.
- The do while loop.

JavaScript while loop:

The while loop executes a group of statements as long as the given expression is True.

The while Keyword is the while loop in JavaScript.

JavaScript While Loop is used to execute a block of statements repeatedly until a given condition is satisfied.

While loop is just like condition based loop.

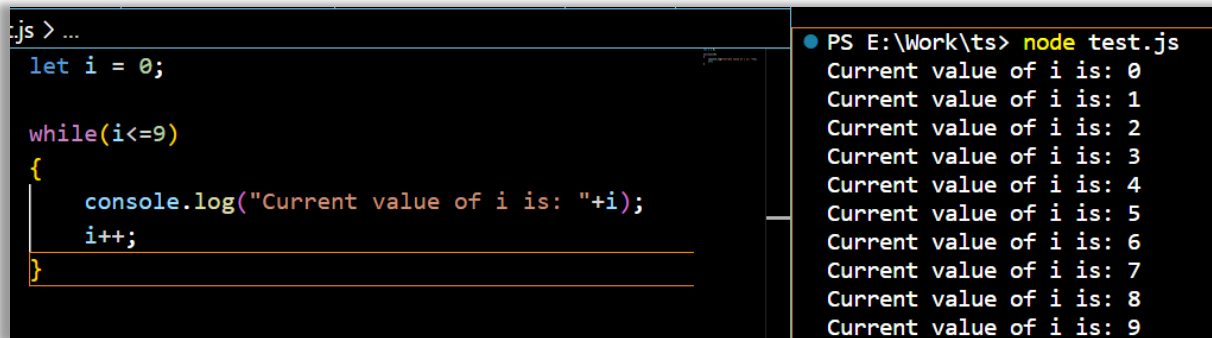
Syntax:

```
while (condition)
{
    // code block to executed
}
```


Important Note:

If you don't increment the counter, the while loop not stop until forever.

Example:



The image shows a code editor on the left and a terminal on the right. The code editor contains the following JavaScript code:

```
js > ...  
let i = 0;  
  
while(i<=9)  
{  
    console.log("Current value of i is: "+i);  
    i++;  
}
```

The terminal on the right shows the output of running the code:

```
PS E:\Work\ts> node test.js  
Current value of i is: 0  
Current value of i is: 1  
Current value of i is: 2  
Current value of i is: 3  
Current value of i is: 4  
Current value of i is: 5  
Current value of i is: 6  
Current value of i is: 7  
Current value of i is: 8  
Current value of i is: 9
```

JavaScript for loop:

The for loop is used to loop through or iterate over a sequence.

The for loop through a block of code a number of times.

The for loop has the following syntax:

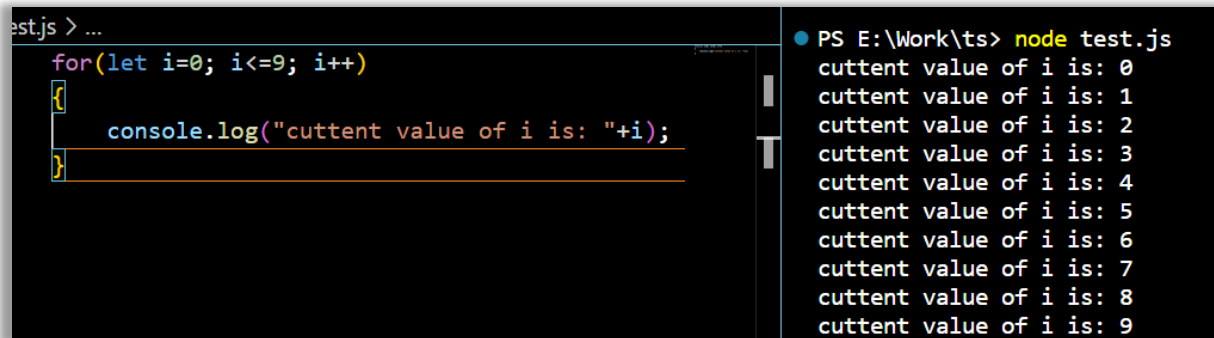
```
for(statement 1; statement 2; statement 3)  
{  
    // code block to be executed  
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 define the condition for execution the code block.

Statement 3 is executed (every time) after the code block has been executed.

Example:

A screenshot of a code editor with a dark background. On the left, a code block is shown with a for loop:

```
for(let i=0; i<=9; i++)  
{  
  console.log("cuttent value of i is: "+i);  
}
```

 On the right, the output of the code is displayed in a terminal window. It shows the command `PS E:\Work\ts> node test.js` followed by ten lines of output: `cuttent value of i is: 0` through `cuttent value of i is: 9`.

```
est.js > ...  
for(let i=0; i<=9; i++)  
{  
  console.log("cuttent value of i is: "+i);  
}  
  
PS E:\Work\ts> node test.js  
cuttent value of i is: 0  
cuttent value of i is: 1  
cuttent value of i is: 2  
cuttent value of i is: 3  
cuttent value of i is: 4  
cuttent value of i is: 5  
cuttent value of i is: 6  
cuttent value of i is: 7  
cuttent value of i is: 8  
cuttent value of i is: 9
```

JavaScript do while loop:

The do while loop is a variant of while loop.

This loop will executed the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

The syntax of do while is as follow:

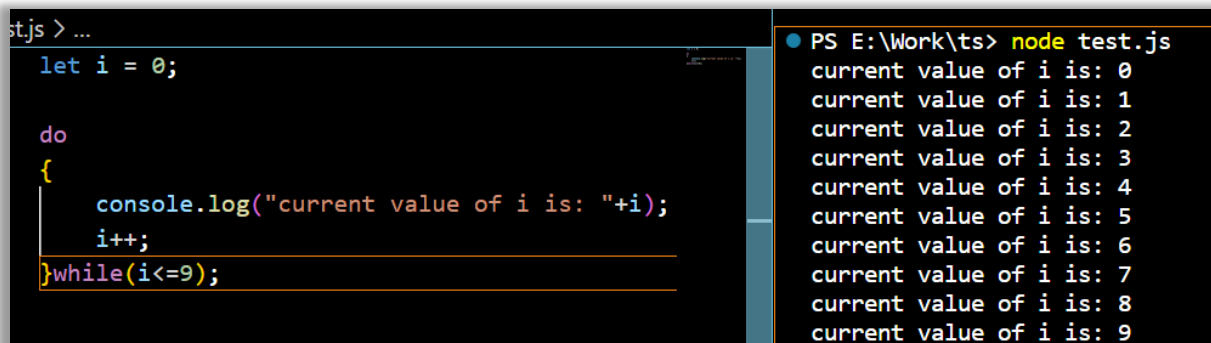
```
do  
{  
  // code block to be executed  
}  
while (condition);
```

Important Note:

If you don't increment the counter, the while loop will not stop until forever.

Example:

The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:



The image shows a code editor on the left and a terminal window on the right. The code editor contains the following JavaScript code:

```
st.js > ...  
let i = 0;  
  
do  
{  
    console.log("current value of i is: "+i);  
    i++;  
}while(i<=9);
```

The terminal window shows the output of running the code:

```
PS E:\Work\ts> node test.js  
current value of i is: 0  
current value of i is: 1  
current value of i is: 2  
current value of i is: 3  
current value of i is: 4  
current value of i is: 5  
current value of i is: 6  
current value of i is: 7  
current value of i is: 8  
current value of i is: 9
```

Break and Continue Statement in JavaScript

The break and continue statement are commonly used in loops.

The Break statement:

The break statement stops the execution of the current loop.

The Continue statement:

The continue statement terminates the execution of the current iteration of the loop.

And continue the execution of the loop with the next iteration.

Example for Break Statement:

```
stjs > ...
for(let i = 1; i<=10; i++)
{
  if(i==4)
  {
    console.log("When the value of i is 4 the loop
    will stops it's iteration.");
    break;
  }
  console.log("current value of i is: "+i);
}
```

- PS E:\Work\ts> node test.js
- current value of i is: 1
- current value of i is: 2
- current value of i is: 3
- When the value of i is 4 the loop will stops it's iteration.
- PS E:\Work\ts> █

Example for Continue Statement:

```
stjs > ...
for(let i = 1; i<=10; i++)
{
  if(i==4)
  {
    console.log("When the value of i is 4 the loop
    will stops it's iteration.");
    continue;
  }
  console.log("current value of i is: "+i);
}
```

- PS E:\Work\ts> node test.js
- current value of i is: 1
- current value of i is: 2
- current value of i is: 3
- When the value of i is 4 the loop will stops it's iteration.
- current value of i is: 5
- current value of i is: 6
- current value of i is: 7
- current value of i is: 8
- current value of i is: 9
- current value of i is: 10
- PS E:\Work\ts> █

Arrow Function in JavaScript

Arrow function expressions were introduced in ES6.

These expressions are a clean and concise alternative to the traditional function syntax.

The syntax for an arrow function expression does not require the function keyword and uses a fat arrow => to separate the parameter(s) from the body.

However, they are limited and can't be used in all situations.

There are several variations of arrow functions:

- Arrow function with a single parameter does not require () around the parameter list.
- Arrow functions with a single expression can use the expression without the return keyword.

Basic Syntax of Arrow Function:

One parameter. With simple expression return is not needed:

Parameter => expression

Multiple parameters require parentheses. With simple expression return is not needed:

(parameter1, parameter) => expression

Multi line statements require body braces and return:

```
Parameter =>
{
  Let a = 1;
  Return a + parameter;
}
```

Multiple parameter require parentheses Multi line statements require body braces and return:

```
(parameter1, parameterN) =>
{
  Let a = 1;
  Return a + parameter1 + parameterN;
}
```

Example:

Arrow function program with no arguments:



The screenshot shows a code editor on the left and a terminal on the right. The code editor contains the following JavaScript code:

```
st.js > ...
const Hello={()=> {
  console.log("Hello World");
}}
Hello();
```

The terminal on the right shows the command prompt output:

```
PS E:\Work\ts> node test.js
Hello World
PS E:\Work\ts>
```

Arrow function program with a single argument:

```
st.js > ...  
const Hello=(weight)=> {  
  | console.log("Baggage weight is "+weight+" Kilograms");  
  | }  
  | Hello(40);
```

● PS E:\Work\ts> node test.js
Baggage weight is 40 Kilograms
○ PS E:\Work\ts> |

Arrow function Program with two arguments:

```
st.js > ...  
const sum=(num1, num2)=> {  
  | return num1 + num2;  
  | }  
  | console.log("The sum of numbers is "+sum(2,5));
```

● PS E:\Work\ts> node test.js
The sum of numbers is 7
○ PS E:\Work\ts> |

Concise arrow function Program example:

```
st.js > ...  
const multiply = (num1, num2)=> num1 * num2;  
console.log("The multiplication of number is : ",multiply(2,30));
```

● PS E:\Work\ts> node test.js
The multiplication of number is : 60
○ PS E:\Work\ts> |

Classes in JavaScript

Classes are a template for creating objects.

They encapsulate data with code to work on that data.

Classes in JavaScript are built on prototypes but also have some syntax and semantics that are not shared with ES5 class-like semantics.

Defining Classes:

Classes are in fact “special functions”, and just as you can define function expressions and function declarations.

The class syntax has two components:

Class expressions and class declarations.

Class declarations:

One way to define a class is using a class declaration. To declare a class, you use the class keyword with the name of the class.

Example:

```
class Rectangle{  
  constructor(height, weight){  
    this.height=height;  
    this.weight=weight  
  }  
}
```

Hoisting:

JavaScript Hoisting refers to the process whereby the interpreter appears to move the declaration of functions, variables or classes to the top of their scope, prior to execution of the code.

Hoisting allows functions to be safely used in code before they are declared.

Variable and class declarations are also hoisted, so they too can be referenced before they are declared.

Note that doing so can lead to unexpected errors, and is not generally recommended.

Function hoisting:

One of the advantages of hoisting is that lets you use a function before you declare it in your code.

Variable hoisting:

Hoisting works with variables too, so you can use a variable in code before it is declared and/or initialized.

Var hoisting:

Here we declare then initialized the value of a var after using it. The default initialization of var is undefined.

Let and Const hoisting:

Variable declared with let and const are also hoisted but, unlike var, are not initialized with a default value.

An exception will be thrown if a variable declaration with let or const is read before it is initialized.

Class hoisting:

Classes defined using a class declaration are hoisted, which means that JavaScript has a reference to the class.

However the class is not initialized by default, so any code that uses it before the line in which it is initialized execution will throw a `ReferenceError`.

An important difference between function declarations and class declarations is that while functions classes must be defined before they can be constructed.

Code like following will throw a `ReferenceError`:

```
const p = new Rectangle();  
class Rectangle{}
```

This occurs because while the class is hoisted its values are not initialized.

Class Expressions:

A class expression is another way to define a class.

Class expressions can be named or unnamed.

The name given to a named class expression is local to the class's body. However, it can be accessed via the `name` property.

```
let Rectangle = class{  
  constructor(height, weight){  
    this.height=height;  
    this.weight=weight;  
  }  
};  
console.log(Rectangle.name);
```

output is “Rectangle”.

```
let Rectangle = class Rectangle2 {  
  constructor(height, weight){  
    this.height=height;  
    this.weight=weight;  
  }  
};  
console.log(Rectangle.name);
```

output is “Rectangle2”

Class body and Methods definitions:

The body of a class is the part that is in curly brackets{ }.

This is where you define class members, such as methods or constructor.

Constructor:

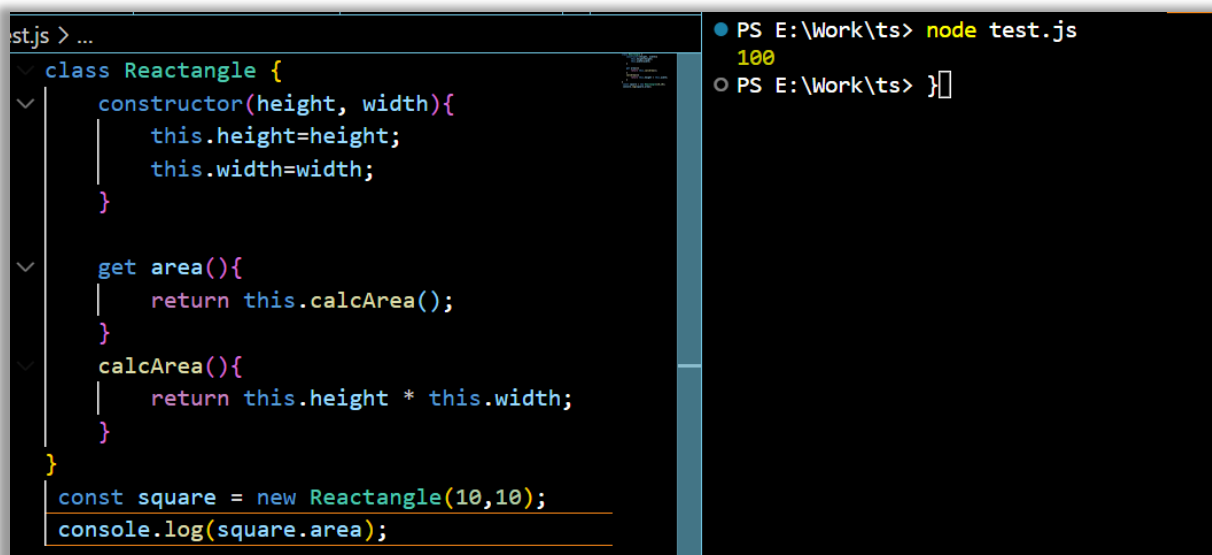
The constructor method is a special method for creating and initializing an object creation with a class.

There can only be one special method with the name “constructor” in a class.

A `SyntaxError` will be thrown if the class contains more than one occurrence of a constructor method.

A constructor can use the `super` keyword to call the constructor of the super class.

Example:



```
st.js > ...  
class Rectangle {  
  constructor(height, width){  
    this.height=height;  
    this.width=width;  
  }  
  
  get area(){  
    return this.calcArea();  
  }  
  
  calcArea(){  
    return this.height * this.width;  
  }  
}  
  
const square = new Rectangle(10,10);  
console.log(square.area);
```

```
PS E:\Work\ts> node test.js  
100  
PS E:\Work\ts> }
```

Sub Classing in JavaScript

The extends keyword is used in class declarations or class expression to create a class as a child of another class.

Example:

```
test.js > ...
class Animal{
  constructor(name, age){
    this.name=name;
    this.age=age;
  }
  eat(){
    return this.name + "is eating";
  }
  isSuperCute(){
    return this.age <=1;
  }
  isCute(){
    return true;
  }
}

class Dog extends Animal{}
const tommy = new Dog("tommy",5);
console.log(tommy);
console.log(tommy.eat());
console.log(tommy.isSuperCute());
console.log(tommy.isCute());
```

```
● PS E:\Work\ts> node test.js
Dog { name: 'tommy', age: 5 }
tommyis eating
false
true
○ PS E:\Work\ts> |
```

Super class calls with super:

The super keyword is used to call corresponding methods of super class.

If there is a constructor present in the subclass, it needs to first call super() before using “this”.

Example:

```
stjs > ...
class Animal{
  constructor(name, age){
    this.name=name;
    this.age=age;
  }
  eat(){
    return this.name + "is eating";
  }
  isSuperCute(){
    return this.age <=1;
  }
  isCute(){
    return true;
  }
}

class Dog extends Animal{
  constructor(name, age, speed){
    super(name,age);
    this.speed=speed;
  }
  run(){
    return this.name + " is running at "+this.speed+" kmph";
  }
}

const tommy = new Dog("tommy",5,20);
console.log(tommy);
console.log(tommy.eat());
console.log(tommy.isSuperCute());
console.log(tommy.isCute());
console.log(tommy.run());
```

```
PS E:\Work\ts> node test.js
Dog { name: 'tommy', age: 5, speed: 20 }
tommyis eating
false
true
tommy is running at 20 kmph
PS E:\Work\ts>
```

Concepts of OOPs in JavaScript

The various concepts of Object Oriented Programming in JavaScript Programming Language are as follow:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

1) Abstraction

An abstraction is a way of hiding the implementation details and showing only the functionality to the users.

In other words, it ignores the irrelevant details and shows only the required one.

The main purpose of abstraction is hiding the unnecessary details from the users.

Real time example of Abstraction is ATM.

In ATM, we will be having an internal process for withdrawing and depositing the amount.

That internal process will not be revealed to the users.

The operation of ATM will be completely based on our action to the input whatever the external buttons which are shown to us like withdraw and deposit.

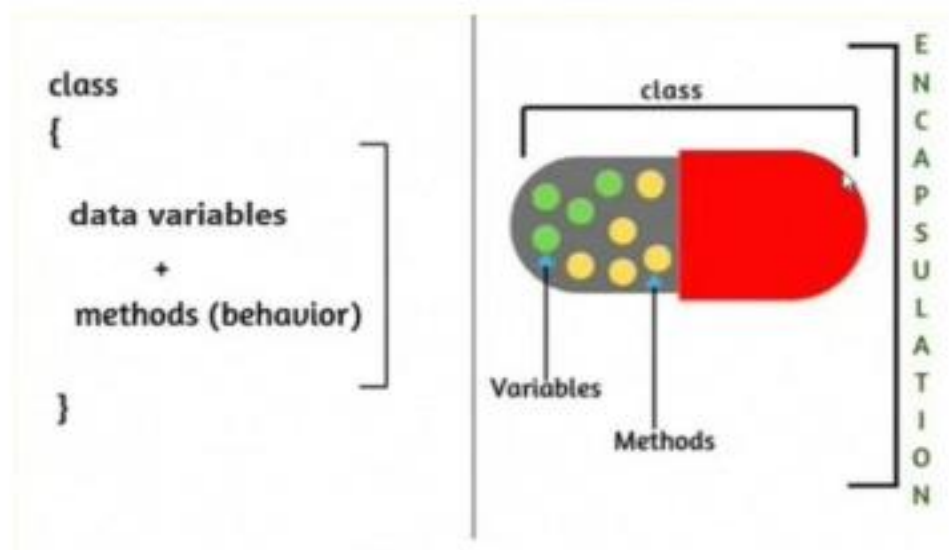
Example:

```
ts> ...
class ATM
{
  constructor(withdraw)
  {
    this.balance = 10000;
    this.withdraw = withdraw;
  }
  getAmount()
  {
    this.minimum = 5000;
    if((this.balance-this.withdraw)>=this.minimum)
    {
      console.log("Withdrwa Successful");
    }
    else
    {
      console.log("Withdrwa Failed");
    }
  }
}

let obj_ATM = new ATM(5000);
console.log("The Minimum balance to be maintained is: ",obj_ATM.minimum);
console.log("The aount user tried to withdwat is: ",obj_ATM.withdraw);
obj_ATM.getAmount();
```

```
PS E:\Work\ts> node test.js
The Minimum balance to be maintained is: undefined
The aount user tried to withdwat is: 5000
Withdrwa Successful
PS E:\Work\ts>
```

2) Encapsulation:



Encapsulation is mechanism of bundling data variables and methods together and hides them from other classes.

Definition of Encapsulation:

Encapsulation can be defined as “the packing of data and functions into one component”.

Packing, which is also known as bundling, grouping and binding, simply means to bring together data and the methods which operate on data.

The component can be a function, a class or an object.

Packing enables “controlling access to that component”. When we have the data and related methods in a single unit, we can control how is it accessed outside the unit.

This process is called as Encapsulation.

Example:

```
estjs > ...  
  
class Person  
{  
  constructor(name,age,salary)  
  {  
    this.name=name;  
    this.age=age;  
    this.salary=salary;  
  }  
  getName()  
  {  
    console.log(this.name);  
  }  
  getAge()  
  {  
    console.log(this.age);  
  }  
  getSalary()  
  {  
    console.log(this.salary);  
  }  
}  
  
let obj_Person = new Person("Tushar Devtwal",21,35000);  
obj_Person.getName();  
obj_Person.getAge();  
obj_Person.getSalary();
```

```
PS E:\Work\ts> node test.js  
Tushar Devtwal  
21  
35000  
PS E:\Work\ts>
```

3) Inheritance:

Inheritance is a mechanism in which one class acquires the property of another class.

Example:

```
estjs > ...  
  
class Parent  
{  
  getMobile()  
  {  
    console.log("iPhone 14");  
  }  
}  
  
class Child extends Parent  
{  
}  
  
let obj_child = new Child();  
console.log("As a Child of your Parent you can acquire  
your Parent's Mobile");  
obj_child.getMobile();
```

```
PS E:\Work\ts> node test.js  
As a Child of your Parent you can acquire your Parent's Mobile  
iPhone 14  
PS E:\Work\ts>
```

4) Polymorphism:

Poly means many and morph means form so all together we call Polymorphism as many forms.

Polymorphism allows us to perform a single action in different ways.

Example:

```
est.js > ...
class Grand_Parent
{
  getMobile()
  {
    console.log("iPhone 14");
  }
}
class Parent extends Grand_Parent
{
  getMobile()
  {
    console.log("iPhone 13");
  }
}
class Child extends Parent
{
  getMobile()
  {
    console.log("iPhone 12");
  }
}
let obj_gp = new Grand_Parent();
console.log("***Grand Parent Mobile is***");
obj_gp.getMobile();
let obj_parent = new Parent();
console.log("***Parent Mobile is***");
obj_parent.getMobile();
let obj_child = new Child();
console.log("***Chile Mobile is***");
obj_child.getMobile();
```

```
PS E:\Work\ts> node test.js
***Grand Parent Mobile is***
iPhone 14
***Parent Mobile is***
iPhone 13
***Chile Mobile is***
iPhone 12
PS E:\Work\ts>
```

Benefits of OOPs:

- Reuse of code through inheritance.
- Flexibility through Polymorphism.
- Easier to troubleshoot.

- Code maintainability.
- Code readability.

Callback, Promises, Async and Await

JavaScript Callback Function:

A callback function can be defined as a function passed into another function as a parameter.

Don't relate the callback with the keyword, as the callback is just a name of a argument that is passed to a function.

In other words, we can say that a function passed to another function as argument is referred to as a callback function.

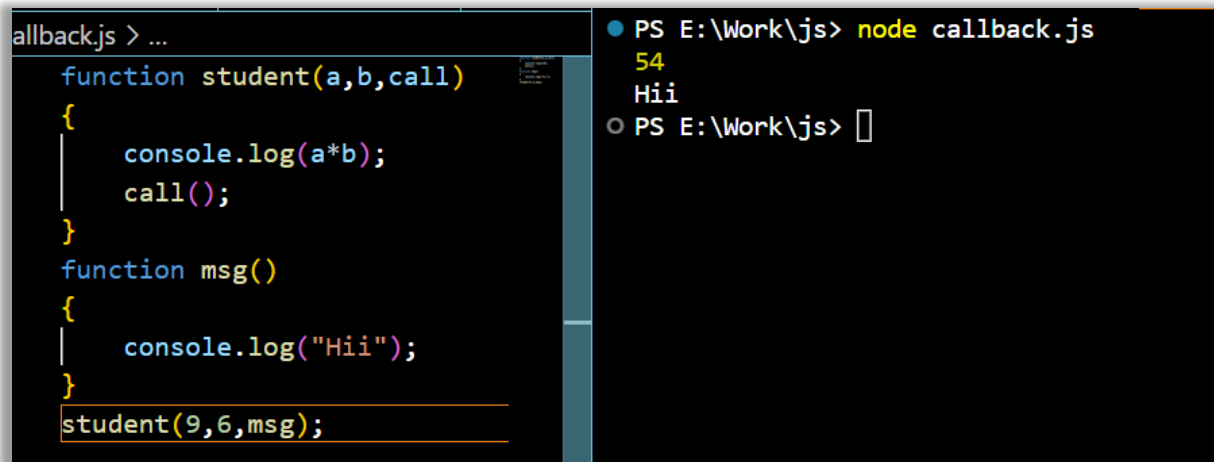
A callback function runs after the completion of the outer function.

It is useful to develop an asynchronous JavaScript code.

In JavaScript, a callback is easier to create. That is, we simply have to pass the callback function as a parameter to another function and call it right after the completion of the task.

Callbacks are mainly used to handle the asynchronous operations such as the registering event listeners, fetching or inserting some data into/from the file.

Example:



```
callback.js > ...
function student(a,b,call)
{
  console.log(a*b);
  call();
}
function msg()
{
  console.log("Hii");
}
student(9,6,msg);
```

```
PS E:\Work\js> node callback.js
54
Hii
PS E:\Work\js>
```

Problem with Callback function:

The main problem with using callback function is nesting of many function leads to a callback hell.

So what is callback hell?

This is big issue caused by coding complex nested callbacks. Here, each and every callback takes an argument that is result of the previous callbacks.

In this manner, the code structure looks like a Pyramid, making it difficult to read and maintain.

Also, if there is an error in one function, then all other function gets affected.

How to escape from a callback hell?

JavaScript provides an easy way of escaping from a callback hell. This is done by event queue and promises.

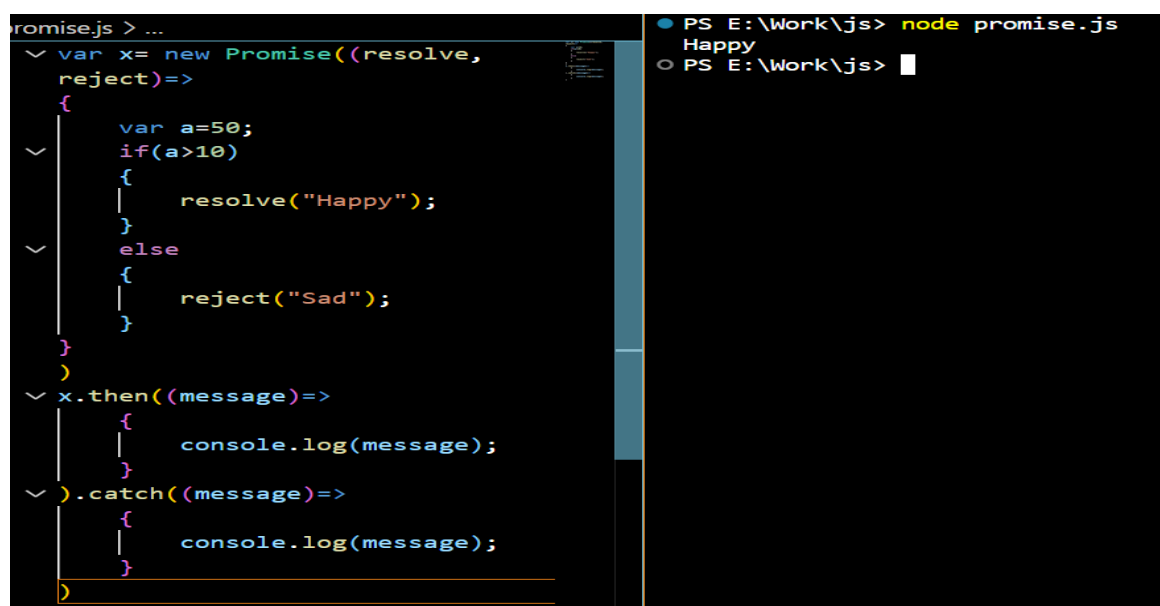
So what are Promises?

A promise is a returned object from any asynchronous function, to which callback methods can be added based on the previous function's result.

Promises use `.then()` method to call Async callback. We can chain as many callback as we want and order is also strictly maintained.

Promise use `.fetch()` method to fetch an object from the network. It also uses `.catch()` method to catch any exception when any block fails.

Example:



```
promise.js > ...
var x= new Promise((resolve,
reject)=>
{
  var a=50;
  if(a>10)
  {
    resolve("Happy");
  }
  else
  {
    reject("Sad");
  }
})
x.then((message)=>
{
  console.log(message);
})
).catch((message)=>
{
  console.log(message);
})
}
```

```
PS E:\Work\js> node promise.js
Happy
PS E:\Work\js> 
```

To simplify and make better use of callbacks using Promises we can use `async` and `await` functions in JavaScript.

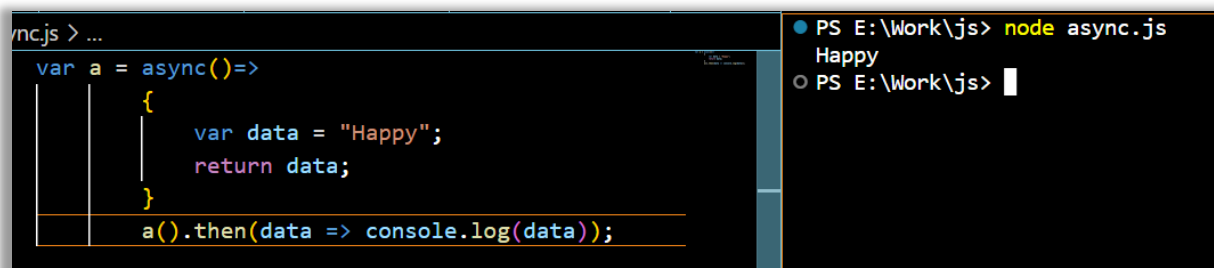
JavaScript Async:

An Async function that is declared with the `async` keyword and allows the `await` keyword inside it.

The `async` and `await` keywords allows asynchronous, promise-based behaviour to be written easily and avoid configured promise chains.

The `async` keyword may be used with any of the methods for creating a function.

Example:



```
nc.js > ...
var a = async()=>
{
  var data = "Happy";
  return data;
}
a().then(data => console.log(data));
```

```
PS E:\Work\js> node async.js
Happy
PS E:\Work\js> 
```

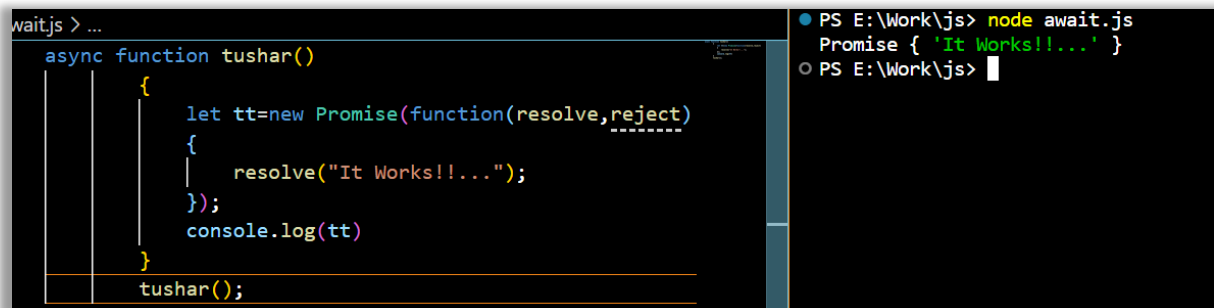
JavaScript Await:

JavaScript Await function is used to wait for the promise. It could only be used inside the `async` block. It instructs the code to wait until the promise returns a response. It only delays the `async` block. Await is a simple command that instructs JavaScript to wait for an asynchronous action to complete before

continuing with the feature. It's similar to a "pause until done" keyword.

The await keyword is used to retrieve a value from a function where we will usually be the then() method. Instead of calling after the asynchronous function, we would use await to allocate a variable to the result and then use the result in the code as we will in the synchronous code.

Example:



```
waitjs > ...  
async function tushar()  
{  
  let tt=new Promise(function(resolve, reject)  
  {  
    resolve("It Works!!...");  
  });  
  console.log(tt)  
}  
tushar();
```

```
PS E:\Work\js> node await.js  
Promise { 'It Works!!...' }  
PS E:\Work\js>
```


Node.js

Node JS or Node is an Open Source and cross platform runtime environment for executing JavaScript Code outside of a browser.

We often use Node to build back-end services also called API's or Application Programming Interfaces.

There are the services that power our Client application like a web app running inside of a web browser or mobile app running on a mobile device.

These client apps are simply what the user sees and interact with.

Client Apps are just surface, they need to take to some services sitting under on the server or in the cloud to store data send emails and push notifications kick off work flow and so on.

Node is ideal for building Highly scalable, data-intensive and real-time back-end services that power our client application.

What's so Special about Node?

- Node is easy to get started and can be used for prototyping super-fast and highly scalable services.
- Node is used in production by large companies such as PayPal, Uber, Netflix, Walmart and so on.
- PayPal is rebuilt one of their Java and Spring based application using Node and it found that the Node application was built twice as fast with fewer people in 33% fewer line of code and 40% fewer files.
- Node is more importantly double the number of requests times by 35% so node is an excellent choice for building highly scalable services.
- Another reason for using Node is that in old applications we use JavaScript so if you are front-end developer and know JavaScript you can reuse your JavaScript skills and transition into a full-stack developer.
- Using Node, you don't need to learn a new programming language also because you can use JavaScript both on the front-end and back-end your source code will be cleaner and more consistent so you will use the same best practices.

- Node is having largest ecosystem of open source libraries available to you so for pretty much any features or building blocks you want to add your application there are some free open source library out there that you can use so you don't have to build this building blocks from scratch and instead you can focus on the core of your application.

File System Module

File system module in Node JS Is used for managing files.

In real time, if you want to read a file and display the content of file on browser or to update the file or to delete the file simply we can do all those activity using File System Module.

For using File System Module in Node JS, we need to import it using the (require) method.

```
var fs = require('fs');
```

Here fs stands for File System.

For implementing fs module we need to write:

```
fs.readFile()
```

Here using fs module, we are reading the file on browser using fs.readFile() method. This method to be called in http.createServer() method only.

For reading the File:

```
var http=require('http');
var fs=require('fs');

http.createServer(function(req,res)
{
    fs.readFile('text.txt',(error,data)=>
    {
        res.write(data);
        res.end();
    })
}).listen(8081);
console.log("http://127.0.0.1:8081/");
```

For appending the File:

```
var http=require('http');
var fs=require('fs');
http.createServer(function(req,res)
{
    fs.appendFile('text.txt',' Hii Students',(error,data)=>
    {
        res.write(data);
        res.end();
    })
}).listen(8081);
console.log("http://127.0.0.1:8081/");
```

For writing the File:

```
var http=require('http');
var fs=require('fs');
http.createServer(function(req,res)
{
    fs.writeFile('abc.txt','this is my text',(error,data)=>
    {
        if(error) throw error;
        fs.readFile('1.html',(error,data)=>
        {
            if(error) throw error;
            res.write(data);
            res.end();
            console.log("done");
        })
    })
}).listen(8081);
console.log("http://127.0.0.1:8081");
```

For deleting File:

```
var http=require('http');
var fs=require('fs');
http.createServer(function(req,res)
{
    fs.unlink('text.txt',(error,data)=>
    {
        res.write(data);
        res.end();
    })
}).listen(8081);
console.log("http://127.0.0.1:8081/");
```

What is Path Parameter?

Path parameter is a part of the URL and takes you to end point/resources and give you the result of query from that resources.

What is Query Parameter?

Query parameter is NOT a part of URL and they are added to the URL after the question mark (?), as key and value it is filtering the result of query.

The end point/resource is same but it acts like a search button and filter the result and returns the response.

Additionally, query parameter can be null but path parameter can't be null. If you don't append the path parameter, you will get 404 error. So you can use path parameter if you want to send the data as mandatory.

Difference between Path parameter & Query parameter:

The first difference between query and path parameters is their position in URL. While the query parameters appear on the right side of the '?' in the URL, path parameters come before the question mark sign.

As part of implementing URL Module, we need to create a temporary URL with Query parameter:

```
var Address = 'http://localhost:8080/home.html?year=2022&month=june';
```

Node JS Email

For sending Email using Node JS we require node mailer module.

For using mailer module, we need to import it first.

First open the terminal or Git Bash or command prompt and try to import the module called “nodemailer”.

npm install nodemailer

By using the command we can install nodemailer in Node JS.

Using this nodemailer module, we can send Email using different platform services like google, yahoo, rediff etc.