JavaScript

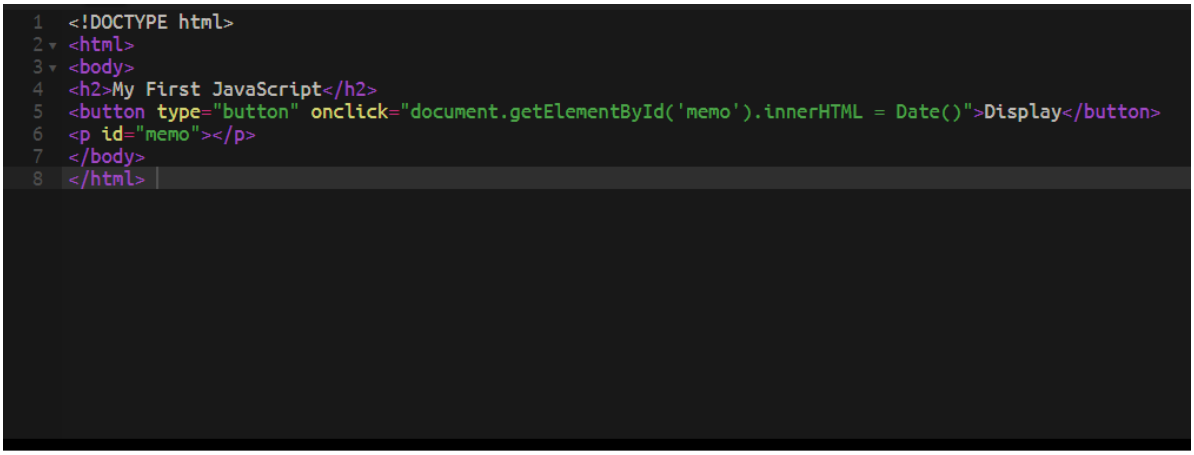
JavaScript is the world's most popular programming language. It is the programming language of the Web. `Null` is of type object.

C,C++,Java are statically data types but Javascript is dynamically typed.
Change of variable datatype is possible at runtime.

Coding Playground

First Code

```
<!DOCTYPE html>
<html>
<body>
<h2>My First JavaScript</h2>
<button type="button" onclick="document.getElementById('memo').innerHTML =
Date()">Display</button>
<p id="memo"></p>
</body>
</html>
```



My First JavaScript

Display

Tue May 18 2021 23:06:09 GMT+0545 (Nepal Time)

Why JavaScript ?

JavaScript is one of the **3 languages** all web developers **must** learn:

1. **HTML** to define the content of web pages
2. **CSS** to specify the layout of web pages
3. **JavaScript** to program the behavior of web pages

Introduction

JavaScript Can Change HTML Content

- getElementById()

  > JavaScript accepts both double and single quotes:

```
onclick='document.getElementById("demo").innerHTML = "Hello JavaScript!"'

onclick="document.getElementById('demo').innerHTML = 'Hello JavaScript!'"
```

JavaScript Can Change HTML Attribute Values

In this example JavaScript changes the value of the `src` (source) attribute of an `<img>` tag:

```
<button onclick="document.getElementById('myImage').src='BulbOnImage'">Turn on
the light</button>

<img id="myImage" src="BulboffImage" style="width:100px">

<button onclick="document.getElementById('myImage').src='BulboffImage'">Turn off
the light</button>
```

JavaScript Can Change HTML Styles (CSS)

```
<p id="demo">JavaScript can change the style of an HTML element.</p>

<button type="button"
onclick="document.getElementById('demo').style.fontSize='35px'">Click Me!
</button>
```

JavaScript Can Hide & Unhide HTML Elements

```
<p id="demo">JavaScript can hide and unhide HTML elements.</p>

<button type="button"
onclick="document.getElementById('demo').style.display='block'">UnHide Me!
</button>

<button type="button"
onclick="document.getElementById('demo').style.display='none'">Hide Me!</button>
```

Where To

The Tag

In HTML, JavaScript code is inserted between tags.

JavaScript in

```
<!DOCTYPE html>
<html>
<title>Color</title>
<head>
<script>
    function myFunction(){ document.getElementById('demo').innerHTML="Function
call";
    }
</script>
</head>
<body>
<h2>JavaScript in Body</h2>
<button type="button" onclick="myFunction()">Try It</button>
<p id="demo"></p>
</body>
</html>
```

JavaScript in

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript in Body</h2>
<p id="demo"></p>
<script>    document.getElementById('demo').innerHTML="Printing From Script";
</script>
</body>
</html>
```

> Placing scripts at the bottom of the element improves the display speed, because script interpretation slows down the display.

External JavaScript

External scripts are practical when the same code is used in many different web pages.

JavaScript files have the file extension **.js**.

To use an external script, put the name of the script file in the `src` (source) attribute of a `<script>` tag:

```
<script src="myScript.js"></script>
```

> You can place an external script reference in `<head>` or `<body>` as you like.

JavaScript Output

JavaScript Display Possibilities

JavaScript can "display" data in different ways:

- Writing into an HTML element, using `innerHTML`.
- Writing into the HTML output using `document.write()`.
- Writing into an alert box, using `window.alert()`.
- Writing into the browser console, using `console.log()`.

Using innerHTML

To access an HTML element, JavaScript can use the `document.getElementById(id)` method.

```
<html>
<body>
<h1>My First Web Page</h1>
<p>My First Paragraph</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>
</body>
</html>
```

Using document.write()

For testing purposes, it is convenient to use `document.write()`:

```
<html>
<body>

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<script>
document.write(5 + 6);
</script>

</body>
</html>
```

Using window.alert()

You can use an alert box to display data:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<script>
window.alert(5 + 6);
</script>

</body>
</html>
```

> You can skip the `window` keyword. It is optional

Using console.log()

For debugging purposes, you can call the `console.log()` method in the browser to display data.

```
<html>
<body>

<script>
console.log(5 + 6);
</script>

</body>
</html>
```

JavaScript Print

`window.print()` method in the browser to print the content of the current window.

```
<html>
<head>
</head>
<title>Print Page</title>
<body>
<button type="button" onclick="window.print()" >Print Page</button>
</body>
</html>
```

JavaScript KeyWords

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

> JavaScript keywords are reserved words. Reserved words cannot be used as names for variables.

Keyword	Description
break	Terminates a switch or a loop
continue	Jumps out of a loop and starts at the top
debugger	Stops the execution of JavaScript, and calls (if available) the debugging function
do ... while	Executes a block of statements, and repeats the block, while a condition is true
for	Marks a block of statements to be executed, as long as a condition is true
function	Declares a function
if ... else	Marks a block of statements to be executed, depending on a condition
return	Exits a function
switch	Marks a block of statements to be executed, depending on different cases
try ... catch	Implements error handling to a block of statements
var	Declares a variable

JavaScript Identifiers

All JavaScript **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

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

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

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

JavaScript identifiers are case-sensitive.

JavaScript Variables

Before 2015, using the `var` keyword was the only way to declare a JavaScript variable.

The 2015 version of JavaScript (ES6) allows the use of the `const` keyword to define a variable that cannot be reassigned, and the `let` keyword to define a variable with restricted scope.

One Statement, Many Variables

You can declare many variables in one statement.

Start the statement with `var` and separate the variables by **comma**:

Redeclaring

Redeclaring a JavaScript variable with `var` is allowed anywhere in a program:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript let</h2>

<p>Redeclaring a JavaScript variable with <b>var</b> is allowed anywhere in a
program:</p>

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

<script>
var x = 2;
// Now x is 2

var x = 3;
// Now x is 3

document.getElementById("demo").innerHTML = x;
</script>

</body>
</html>
```

Redeclaring a `var` variable with `let`, in the same scope, or in the same block, is not allowed. Redeclaring a `let` variable with `let`, in the same scope, or in the same block, is not allowed. but in another scope, or in another block is allowed.

Variables defined with `const` behave like `let` variables, except they cannot be reassigned. JavaScript `const` variables must be assigned a value when they are declared.

JavaScript Data Types

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

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

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

JavaScript Functions

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

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

JavaScript Function Syntax

A JavaScript function is defined with the `function` keyword, followed by a **name**, followed by parentheses **()**.

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas:
(*parameter1, parameter2, ...*)

The code to be executed, by the function, is placed inside curly brackets: **{}**

JavaScript Objects

Objects are variables too. But objects can contain many values. You define (and create) a JavaScript object with an object literal:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Objects</h2>

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

<script>
// Create an object:
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

// Display some data from the object:
document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>

</body>
</html>
```

If you access a method **without** the () parentheses, it will return the **function definition**:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Objects</h2>

<p>If you access an object method without (), it will return the function
definition:</p>

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

<script>
// Create an object:
var person = {
  firstName: "John",
  lastName : "Doe",
  id     : 5566,
  fullName : function() {
```

```
    return this.firstName + " " + this.lastName;
  }
};

// Display data from the object:
document.getElementById("demo").innerHTML = person.fullName;
</script>

</body>
</html>
```

When a JavaScript variable is declared with the keyword "`new`", the variable is created as an object:

```
var x = new String();        // Declares x as a String object
var y = new Number();        // Declares y as a Number object
var z = new Boolean();       // Declares z as a Boolean object
```

Avoid `String`, `Number`, and `Boolean` objects. They complicate your code and slow down execution speed.

JavaScript Events

HTML events are **"things"** that happen to HTML elements.

When JavaScript is used in HTML pages, JavaScript can **"react"** on these events.

HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.

Here is a list of some common HTML events:

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

JavaScript Strings

JavaScript strings are used for storing and manipulating text.

Strings Can be Objects

Normally, JavaScript strings are primitive values, created from literals:

```
var firstName = "John";
```

But strings can also be defined as objects with the keyword `new`:

```
var firstName = new String("John");
```

When using the `==` operator, equal strings are equal. But using the `===` operator, equal values may not be equal, because the `===` operator expects equality in both data type and value Or even worse. Objects cannot be compared.

Comparing two JavaScript objects will **always** return `false`.

Methods & Properties

Primitive values, cannot have properties or methods (because they are not objects).

But with JavaScript, methods and properties are also available to primitive values, because JavaScript treats primitive values as objects when executing methods and properties.

indexOf() ： returns the index of (the position of) the `first` occurrence of a specified text in a string

lastIndexOf() : method returns the index of the `last` occurrence of a specified text in a string.

search() : method searches a string for a specified value and returns the position of the match.

Both `indexOf()`, and `lastIndexOf()` return -1 if the text is not found.

Both `indexOf()`, and `lastIndexOf()` accept a second parameter as the starting position for the search.

The two methods, `indexOf()` and `search()`, are **equal?**

They accept the same arguments (parameters), and return the same value?

The two methods are **NOT** equal. These are the differences:

- The `search()` method cannot take a second start position argument.
- The `indexOf()` method cannot take powerful search values.

Extracting String Parts

There are 3 methods for extracting a part of a string:

- `slice(*start*, *end*)`

  `slice()` extracts a part of a string and returns the extracted part in a new string.

  The method takes 2 parameters: the start position, and the end position (end not included). If a parameter is negative, the position is counted from the end of the string.

- `substring(*start*, *end*)`

  `substring()` is similar to `slice()`.

  The difference is that `substring()` cannot accept negative indexes. If you omit the second parameter, `substring()` will slice out the rest of the string.

- `substr(*start*, *length*)`

  `substr()` is similar to `slice()`.

  The difference is that the second parameter specifies the **length** of the extracted part. If the first parameter is negative, the position counts from the end of the string.

Converting to Upper and Lower Case

A string is converted to upper case with `toUpperCase()`

A string is converted to lower case with `toLowerCase()`

JavaScript Numbers

They are always 64-bit Floating Point

The toString() Method

The `toString()` method returns a number as a string.

All number methods can be used on any type of numbers (literals, variables, or expressions):

```
var x = 123;
x.toString();            // returns 123 from variable x
(123).toString();        // returns 123 from literal 123
(100 + 23).toString();   // returns 123 from expression 100 + 23
```

The toExponential() Method

`toExponential()` returns a string, with a number rounded and written using exponential notation.

A parameter defines the number of characters behind the decimal point:

```
var x = 9.656;
x.toExponential(2);     // returns 9.66e+0
x.toExponential(4);     // returns 9.6560e+0
x.toExponential(6);     // returns 9.656000e+0
```

The toFixed() Method

`toFixed()` returns a string, with the number written with a specified number of decimals:

```
var x = 9.656;
x.toFixed(0);           // returns 10
x.toFixed(2);           // returns 9.66
x.toFixed(4);           // returns 9.6560
x.toFixed(6);           // returns 9.656000
```

The toPrecision() Method

`toPrecision()` returns a string, with a number written with a specified length:

```
var x = 9.656;
x.toPrecision();        // returns 9.656
x.toPrecision(2);       // returns 9.7
x.toPrecision(4);       // returns 9.656
x.toPrecision(6);       // returns 9.65600
```

JavaScript Arrays

JavaScript arrays are used to store multiple values in a single variable.

An array is a special variable, which can hold more than one value at a time. Arrays are a special type of objects. The `typeof` operator in JavaScript returns "object" for arrays.

But, JavaScript arrays are best described as arrays.

How to Recognize an Array

The problem is that the JavaScript operator `typeof` returns "`object`":

Solution 1:

```
Array.isArray(Array Name);  // returns true
```

Solution 2:

The `instanceof` operator returns true if an object is created by a given constructor:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits instanceof Array;  // returns true
```

Sorting an Array

The `sort()` method sorts an array alphabetically:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();        // Sorts the elements of fruits
```

Reversing an Array

The `reverse()` method reverses the elements in an array. You can use it to sort an array in descending order:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();        // First sort the elements of fruits
fruits.reverse();     // Then reverse the order of the elements
```

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Array Sort Reverse</h2>

<p>The reverse() method reverses the elements in an array.</p>
<p>By combining sort() and reverse() you can sort an array in descending order.
</p>
<button onclick="myFunction()">Try it</button>

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

<script>
// Create and display an array:
var fruits = ["Banana", "Orange", "Apple", "Mango"];
```

```
document.getElementById("demo").innerHTML = fruits;

function myFunction() {
  // First sort the array
  fruits.sort();
  // Then reverse it:
  fruits.reverse();
  document.getElementById("demo").innerHTML = fruits;
}
</script>

</body>
</html>
```

Numeric Sort

By default, the `sort()` function sorts values as **strings**. This works well for strings ("Apple" comes before "Banana"). However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1". Because of this, the `sort()` method will produce incorrect result when sorting numbers.

You can fix this by providing a **compare function**:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Array Sort</h2>

<p>Click the button to sort the array in ascending order.</p>

<button onclick="myFunction()">Try it</button>

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

<script>
var points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML = points;

function myFunction() {
  points.sort(function(a, b){return a - b});
  document.getElementById("demo").innerHTML = points;
}
</script>

</body>
</html>
```

JavaScript Math

The JavaScript Math object allows you to perform mathematical tasks on numbers.

Unlike other objects, the Math object has no constructor.

JavaScript provides 8 mathematical constants that can be accessed as Math properties:

```
Math.E        // returns Euler's number
Math.PI       // returns PI
Math.SQRT2    // returns the square root of 2
Math.SQRT1_2  // returns the square root of 1/2
Math.LN2      // returns the natural logarithm of 2
Math.LN10     // returns the natural logarithm of 10
Math.LOG2E    // returns base 2 logarithm of E
Math.LOG10E   // returns base 10 logarithm of E
```

Math Methods

The syntax for Math any methods is : `Math.method.(number)`

Number to Integer

There are 4 common methods to round a number to an integer:

Math.round(x)	Returns x rounded to its nearest integer
Math.ceil(x)	Returns x rounded up to its nearest integer
Math.floor(x)	Returns x rounded down to its nearest integer
Math.trunc(x)	Returns the integer part of x
Math.sign(x)	Returns if x is negative, null or positive
Math.pow(x, y)	Returns the value of x to the power of y
Math.sqrt(x)	Returns the square root of x
Math.abs(x)	Returns the absolute (positive) value of x
Math.random()	Returns a random number between 0 (inclusive), and 1 (exclusive)
Math.log(x)	Returns the natural logarithm of x

JavaScript ReGex

A regular expression is a sequence of characters that forms a search pattern. It can be used for text search and text replace operations.

In JavaScript, regular expressions are often used with the two **string methods**: `search()` and `replace()`.

The `search()` method uses an expression to search for a match, and returns the position of the match.

The `replace()` method returns a modified string where the pattern is replaced.

Modifier	Description
i	Perform case-insensitive matching
g	Perform a global match (find all matches rather than stopping after the first match)
m	Perform multiline matching

JavaScript Hoisting

Hoisting is JavaScript's default behavior of moving declarations to the top.

In JavaScript a variable can be used before it has been declared.

But for let and const keywords block of code is aware of the variable, but it cannot be used until it has been declared. Using a `let` variable before it is declared will result in a `ReferenceError` .

Example

This will result in a `ReferenceError` :

```
carName = "Volvo";
let carName;
```

Example

This code will not run.

```
carName = "Volvo";
const carName;
```

> JavaScript only hoists declarations, not initializations.

The "use strict" Directive. The purpose of `"use strict"` is to indicate that the code should be executed in "strict mode". With strict mode, you can not, for example, use undeclared variables.

Strict mode is declared by adding "use strict"; to the beginning of a script or a function.

Strict mode makes it easier to write "secure" JavaScript.

Strict mode changes previously accepted "bad syntax" into real errors.

JavaScript This Keyword

The JavaScript `this` keyword refers to the object it belongs to.

```
<script>
// Create an object:
var person = {
  firstName: "John",
  lastName : "Doe",
  id     : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};

// Display data from the object:
document.getElementById("demo").innerHTML = person.fullName();
</script>
```

JavaScript Arrow Function

Arrow functions were introduced in ES6. In 99% of cases we use arrow function but in 1% case we dont use it when we have to use **this keywords**.

Arrow functions allow us to write shorter function syntax:

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

With Arrow Function

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

JavaScript Classes

Use the keyword `class` to create a class. A JavaScript class is **not** an object. It is a **template** for JavaScript objects.

Always add a method named `constructor()`:

```
<script>
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}

myCar = new Car("Ford", 2014);
document.getElementById("demo").innerHTML =
myCar.name + " " + myCar.year;
</script>
```

```
<script>
class Car {
  constructor(name, year) {
```

```
    this.name = name;
    this.year = year;

  }
  age() {
    let date = new Date();
    return date.getFullYear() - this.year;
  }
}

let myCar = new Car("Ford", 2014);
document.getElementById("demo").innerHTML =
"My car is " + myCar.age() + " years old.";
</script>
```

Classes also allows you to use getters and setters. To add getters and setters in the class, use the `get` and `set` keywords.

```
<html>
<body>
<h2>JavaScript Class Gettter/Setter</h2>
<p id="demo"></p>
<script>
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  get cnam() {
    return this.carname;
  }
  set cnam(x) {
    this.carname = x;
  }
}
let myCar = new Car("Ford");
document.getElementById("demo").innerHTML = myCar.cnam;
</script>
</body>
</html>
```

Unlike functions, and other JavaScript declarations, class declarations are not hoisted.

That means that you must declare a class before you can use it.

Static class methods are defined on the class itself.

You cannot call a `static` method on an object, only on an object class.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello() {
    return "Hello!!";
  }
}
```

```
let myCar = new Car("Ford");

// You can calll 'hello()' on the Car Class:
document.getElementById("demo").innerHTML = Car.hello();

// But NOT on a Car Object:
// document.getElementById("demo").innerHTML = myCar.hello();
// this will raise an error.
```

If you want to use the myCar object inside the `static` method, you can send it as a parameter:

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Hello " + x.name;
  }
}
let myCar = new Car("Ford");
document.getElementById("demo").innerHTML = Car.hello(myCar);
```

JavaScript JSON

JSON is a format for storing and transporting data. JSON is often used when data is sent from a server to a web page.

JSON Syntax Rules

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

JSON Data - A Name and a Value

JSON data is written as name/value pairs, just like JavaScript object properties.

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

```
"firstName":"John"
```

JSON Objects

JSON objects are written inside curly braces.

Just like in JavaScript, objects can contain multiple name/value pairs:

```
{"firstName":"John", "lastName":"Doe"}
```

Converting a JSON Text to a JavaScript Object

A common use of JSON is to read data from a web server, and display the data in a web page

```
<html>
<body>

<h2>Create Object from JSON String</h2>

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

<script>
var text = '{"employees":[' +
'{"firstName":"John","lastName":"Doe" },' +
'{"firstName":"Anna","lastName":"Smith" },' +
'{"firstName":"Peter","lastName":"Jones" }]}';

obj = JSON.parse(text);
document.getElementById("demo").innerHTML =
obj.employees[1].firstName + " " + obj.employees[1].lastName;
</script>

</body>
</html>
```

JavaScript Scope

Global Scope

Variables declared **Globally** (outside any function) have **Global Scope**.

```
var carName = "Volvo";

// code here can use carName

function myFunction() {
  // code here can also use carName
}
```

Global variables can be accessed from anywhere in a JavaScript program.

Function Scope

Variables declared **Locally** (inside a function) have **Function Scope**.

```
// code here can NOT use carName

function myFunction() {
  var carName = "Volvo";
  // code here CAN use carName
}

// code here can NOT use carName
```

Local variables can only be accessed from inside the function where they are declared.

JavaScript Block Scope

Variables declared with the `var` keyword cannot have **Block Scope**.

Variables declared inside a block **{}** can be accessed from outside the block.

```
{
  var x = 2;
}
// x CAN be used here
```

Variables declared with the `let` keyword can have Block Scope.

Variables declared inside a block **{}** cannot be accessed from outside the block:

```
{
  let x = 2;
}
// x can NOT be used here
```

Redeclaring Variables

Redeclaring a variable using the `var` keyword can impose problems.

Redeclaring a variable inside a block will also redeclare the variable outside the block:

```
var x = 10;
// Here x is 10
{
  var x = 2;
  // Here x is 2
}
// Here x is 2
```

Redeclaring a variable using the `let` keyword can solve this problem.

Redeclaring a variable inside a block will not redeclare the variable outside the block:

```
var x = 10;
// Here x is 10
{
  let x = 2;
  // Here x is 2
}
// Here x is 10
```

JavaScript Callbacks

A callback is a function passed as an argument to another function.

This technique allows a function to call another function. A callback function can run after another function has finished.

```
<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}

myCalculator(5, 5, myDisplayer);
</script>
```

Callbacks really shine are in asynchronous functions, where one function has to wait for another function (like waiting for a file to load).

Asynchronous JavaScript

What is **synchronous** ? Wait for the task to finish which are placed before it .

```
posts = loadPostsSync();
// ...wait til posts are fetched
// ...do something with posts

doTheNextThing(); // Has to wait until posts load
```

So in synchronous it block all next function until all the post are fetched and loaded which slows things down.

In Asynchronous

```
loadPostsAsync(function () {
  // ...wait til posts are fetched
  // ...do something with posts
});

doTheNextThing(); // Doesn't have to wait until posts load
```

In this case instead of just pulling the post out of a synchronous function we're passing in callback function. This callback will run and fetch the post. But here all other function aren't blocked . doTheNextThing() function don't have to wait until all the post are fetched and loaded. So even if it takes some time to load the post it doesn't stop rest of code.

Waiting for a Timeout

When using the JavaScript function `setTimeout()`, you can specify a callback function to be executed on time-out:

```
<html>
```

```
<body>

<h2>JavaScript Callback</h2>

<p>Wait 3 seconds for this page to change.</p>

<h1 id="demo"></h1>

<script>
setTimeout(myFunction, 3000);

function myFunction() {
  document.getElementById("demo").innerHTML = "Your 3 seconds are over !!";
}
</script>

</body>
</html>
```

> When you pass a function as an argument, remember not to use parenthesis.
>
> Right: setTimeout(myFunction, 3000);
>
> Wrong: ~~setTimeout(myFunction(), 3000);~~

Waiting for Intervals:

When using the JavaScript function `setInterval()`, you can specify a callback function to be executed for each interval:

```
<html>
<body>

<h2>JavaScript setInterval()</h2>

<p>Using setInterval() to display the time every second.</p>

<h1 id="demo"></h1>

<script>
setInterval(myFunction, 1000);

function myFunction() {
  let d = new Date();
  document.getElementById("demo").innerHTML=

  d.getHours() + ":" +
  d.getMinutes() + ":" +
  d.getSeconds();
}
</script>

</body>
</html>
```

Waiting for Files

If you create a function to load an external resource (like a script or a file), you cannot use the content before it is fully loaded.

This is the perfect time to use a callback.

This example loads a HTML file (`mycar.html`), and displays the HTML file in a web page, after the file is fully loaded:

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

function getFile(myCallback) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.html");
  req.onload = function() {
    if (req.status == 200) {
      myCallback(this.responseText);
    } else {
      myCallback("Error: " + req.status);
    }
  }
  req.send();
}

getFile(myDisplayer);
```

In the example above, `myDisplayer` is used as a callback.

The function (the function name) is passed to `getFile()` as an argument.

JavaScript Promises

"Producing code" is code that can take some time

"Consuming code" is code that must wait for the result

A Promise is a JavaScript object that links producing code and consuming code

JavaScript Promise Object

A JavaScript Promise object contains both the producing code and calls to the consuming code:

Syntax :

```
let myPromise = new Promise(function(myResolve, myReject) {
// "Producing Code" (May take some time)

  myResolve(); // when successful
  myReject();  // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

The Promise object supports two properties :

- State
- Result

myPromise.state	myPromise.result
"pending"	undefined
"fulfilled"	a result value
"rejected"	an error object

How to use Promise ?

```
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Promise.then() takes two arguments, a callback for success and another for failure.

Both are optional, so you can add a callback for success or failure only.

Example

```
<html>
<body>

<h2>JavaScript Promise</h2>

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

<script>
function myDisplayer(some) { // --5
  document.getElementById("demo").innerHTML = some;
}

//My Promise Object --- 1
let myPromise = new Promise(function(myResolve, myReject) {
//Producing Code  --2
let x = 0;
```

```
//When Sucessful --3
  if (x == 0) {
  //Passing Argument To Function
    myResolve("OK");
  }
//When Failure
  else {
  //Passing Argument To Function
    myReject("Error");
  }
});

//Consuming Code --4
myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
</script>

</body>
</html>
```

Waiting For Timeout

As seen in asynchronous here we will use promise instead of callbacks.

```
<html>
<body>

<h2>JavaScript Promise</h2>

<p>Wait 3 seconds for this page to change.</p>

<h1 id="demo"></h1>

<script>
let myPromise = new Promise(function(myResolve, myReject) {
  setTimeout(function(){ myResolve(document.getElementById("demo").innerHTML =
"Your 3 seconds are over !!"); }, 3000);
});

myPromise.then(function(value) {
  document.getElementById("demo").innerHTML = value;
});
</script>

</body>
</html>
```

Waiting For a File

```
<html>
<body>

<h2>JavaScript Promise</h2>

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

<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

let myPromise = new Promise(function(myResolve, myReject) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.html");
  req.onload = function() {
    if (req.status == 200) {
      myResolve(req.response);
    } else {
      myReject("File not Found");
    }
  };
  req.send();
});

myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
</script>

</body>
</html>
```

JavaScript Async

"async and await make promises easier to write"

async makes a function return a Promise

await makes a function wait for a Promise

Async Syntax

The keyword `async` before a function makes the function return a promise:

```
<html>
<body>

<h2>JavaScript async / await</h2>

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

<script>
```

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

async function myFunction() {
return Promise.resolve("Hello");
//Can be written as
//  return "Hello";

}

myFunction().then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
</script>
</body>
</html>
```

Await Syntax

The keyword `await` before a function makes the function wait for a promise:

The `await` keyword can only be used inside an `async` function.

```
<html>
<body>

<h2>JavaScript async / await</h2>

<h1 id="demo"></h1>

<script>
async function myDisplay() {
  let myPromise = new Promise(function(myResolve, myReject) {
    myResolve("Hello Buddy !!");
  });
  document.getElementById("demo").innerHTML = await myPromise;
}

myDisplay();
</script>

</body>
</html>
```