Javascript Notes

Adding js file to html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <!-- script tags can be used with internal js scripts in both the header or body-->
    <!-- putting the script tag in the body is a bit faster-->
    <!-- Link to external js file is shown below and can also be placed in header or body-->
    <p id = 'demo'></p>
    <!--<script src = 'script.js'></script>-->
    <!-- to add several script files to one page, use several script tags-->
    <!-- external scripts can be referenced with a full URL  -->
    <script src="https://www.w3schools.com/js/myScript.js"></script>
    <script src = 'script2.js'></script>
  </body>
</html>
```

Comments

Single Line Comments // and Multi-Line Comments /**/

```
//single line comment

/*
multi line comment
*/
```

Variables

var

var declarations are globally scoped or function/locally scoped. This means that any variable that is declared with var outside a function block is available for use in the whole window.

```
var greeter = "hey hi";
function newFunction() {
    var hello = "hello";
    var greeter = 'yo'
    console.log(greeter);//prints yo
}
newFunction();
console.log(greeter);//prints 'hey hi'
//console.log(hello); // error: hello is not defined
```

var variables can be re-declared and updated

```
var greeter = "hey hi";
var greeter = "say Hello instead";
greeter = "say yo instead";
```

Hoisting of var. Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution

```
console.log (var1); // prints undefined
var var1 = "say hello"
//the above 2 lines is the same a the below two lines
var var2;
console.log(var2);  //prints undefined
var2 = 'say hello'
```

```
var var3 = 'yes'
if (true){
  var3 = 'no' //changes the value of the var from yes to no, not declare a new local variable
}
console.log(var3);  //will print 'no'
```

let

let is now preferred for variable declaration. let is blocked scope. a block is anything with {}

```
let var3 = 'sayHi';
if(True){
  let hello = 'hello1';
  console.log(hello); //prints hello1
}
console.log(hello); //returns error since hello is not defined
```

let can be updated but not re-declared

```
let var4 = "say Hi";
var4 = "say Hello instead"; //no error

let varr = 'yo';
let varr = 'no' //error since you cannot redeclare
```

When using let, you don't have to bother if you have used a name for a variable before as a variable exists only within its scope. This fact makes let a better choice than var

```
let var5 = "say Hi";
if (true) {
    let var5 = "say Hello instead";
    console.log(var5); // "say Hello instead"
}
console.log(var5); // "say Hi"
```

Hoisting of let. Just like var, let declarations are hoisted to the top. Unlike var which is initialized as undefined, the let keyword is not initialized. So if you try to use a let variable before declaration, you'll get a Reference Error

const

It does not define a constant value. It defines a constant reference to a value.
Because of this you can NOT:

- Reassign a constant value
- Reassign a constant array
- Reassign a constant object

```
const PI = 3.14159265359;
```

But you CAN:

- Change a constant array

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];
// You can change an element:
cars[0] = "Toyota";
// You can add an element:
cars.push("Audi");

const cars2 = ["Saab", "Volvo", "BMW"];
cars2 = ["Toyota", "Volvo", "Audi"];    // ERROR
```

- Change a constant object

Declaring a variable with const is similar to let when it comes to Block Scope. The x declared in the block, in this example, is not the same as the x declared outside the block. Redeclaring a variable with const, in another scope, or in another block, is allowed

```
const x = 2;       // Allowed
x = 2;             // Not allowed
var x = 2;         // Not allowed
let x = 2;         // Not allowed
const x = 2;       // Not allowed

{
  const x = 2;   // Allowed
  x = 2;         // Not allowed
  var x = 2;     // Not allowed
  let x = 2;     // Not allowed
  const x = 2;   // Not allowed
}
```

Hoisting, using a const variable before it is declared will result in a ReferenceError

```
alert (carName); //error
const carName = "Volvo";
```

Operators

JavaScript Arithmetic Operators

Arithmetic operators are used to perform arithmetic on numbers:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

```
console.log(3**2);  //prints 9

let text1 = "What a very ";
text1 += "nice day";
text1 += 3;
console.log(text1); //prints What a very nice day3
```

JavaScript evaluates expressions from left to right. Different sequences can produce different results:

```
let x = 16 + 4 + "Volvo";   // x is 20Volvo, JavaScript treats 16 and 4 as numbers, until it reaches "Volvo"
let y = "Volvo" + 16 + 4;   // y is Volvo164,since the first operand is a string, all operands are treated as strings.
```

JavaScript Comparison Operators

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

JavaScript Logical Operators

Operator	Description
&&	logical and
||	logical or
!	logical not

JavaScript Type Operators

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

Data Types

Declaring Variables

Separated by semicolons, multiple statements on one line are allowed. The two codes below are the same.

```
let a, b, c;  // Declare 3 variables
a = 5;        // Assign the value 5 to a
b = 6;        // Assign the value 6 to b
c = a + b;    // Assign the sum of a and b to c
```

```
a = 5; b = 6; c = a + b;
```

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

```
let x;           // Now x is undefined
x = 5;           // Now x is a Number
x = "John";      // Now x is a String
```

Undefined

```
let x;           // Now x is undefined
```

Strings/Empty

```
let carName1 = "Volvo XC60";   // Using double quotes
let carName2 = 'Volvo XC60';   // Using single quotes
```

An empty value has nothing to do with undefined. An empty string has both a legal value and a type

```
let car = "";    // The value is "", the typeof is "string"
```

Numbers

```
let x1 = 34.00;
let x2 = 34;
let x3 = 3.14;
```

The above 3 variables will print as the following 3

```
34
34
3.14
```

```
let y = 123e5;      // 12300000
let z = 123e-5;     // 0.00123
```

Booleans

```
let x = true;
let y = true;
```

Arrays

```
const cars = ["Saab",3,"BMW"];
```

Objects

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
```

typeOf Operator

```
typeof "John Doe"     // Returns "string"
```

Arrays

Creating an Array

you can have variables of different types in the same Array such as objects, functions, arrays

```
let str = 'string';
const cars = [3,2,str];
console.log(cars);
```

```
[3, 2, "string"]
```

Accessing Array Elements

```
const cars = ["Saab", "Volvo", "BMW"];
let x = cars[0];    // x = "Saab"
```

Changing Array Elements

```
const cars = ["Saab", "Volvo", "BMW"];
cars[0] = "Opel";
```

Access the Full Array

```
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
```

This will change demo to

Saab,Volvo,BMW

Adding Array Elements

The push() method add an element to the end of an the array

```
const fruits = ["Banana", "Orange", "Apple"];
fruits.push("Lemon");  // Adds a new element (Lemon) to fruits
```

Adding large indexes can create holes

```
const fruits = ["Banana", "Orange", "Apple"];
fruits[6] = "Lemon";
```

This will print the following

Banana
Orange
Apple
undefined
undefined
undefined
Lemon

The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon"); //adds "Lemon" to fruits
console.log(fruits); //prints ["Lemon", "Banana", "Orange", "Apple", "Mango"]
console.log(fruits.unshift("yo"));  //prints 6
console.log(fruits);  //prints ["yo", "Lemon", "Banana", "Orange", "Apple", "Mango"]
```

Remove Elements

The pop() method removes the last element from an array

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.pop();  // Removes "Mango" from fruits
let x = fruits.pop() //Removes Apple from fruits and stores it in x
console.log(x); //prints Apple
console.log(fruits); //prints ["Banana", "Orange"]
```

The shift() method removes the first array element and "shifts" all other elements to a lower index

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.shift();  // Removes "Banana" from fruits
let x = fruits.shift() //Removes Orange from fruits and stores it in x
console.log(x); //prints Orange
console.log(fruits); //prints ["Apple", "Mango"]
```

Length

```
cars.length   // Returns the number of elements
```

typeOf and Array.isArray() and instanceof

The problem is that the JavaScript operator typeof returns "object" for arrays

```
const fruits = ["Banana", "Orange", "Apple"];
typeof fruits;    // returns object
```

To solve this issue, we can use Array.isArray or instanceof

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

fruits instanceof Array;   // returns true
```

Convert Arrays to Strings

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
console.log(fruits.toString());
// prints Banana,Orange,Apple,Mango
```

Splicing Arrays

The splice() method can be used to add new items to an array

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi");
console.log(fruits);
//prints ["Banana", "Orange", "Lemon", "Kiwi", "Apple", "Mango"]
```

The first parameter (2) defines the position where new elements should be added (spliced in).
The second parameter (0) defines how many elements should be removed.
The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be added.

Merging (Concatenating) Arrays

The concat() method creates a new array by merging (concatenating) existing arrays.
The concat() method does not change the existing arrays. It always returns a new array.
The concat() method can take any number of array arguments.

```
const arr1 = ["Cecilie", "Lone"];
const arr2 = ["Emil", "Tobias", "Linus"];
const arr3 = ["Robin", "Morgan"];
const myChildren = arr1.concat(arr2, arr3);
console.log(myChildren);
//prints ["Cecilie", "Lone", "Emil", "Tobias", "Linus", "Robin", "Morgan"]
```

Slicing Arrays

The slice() method slices out a piece of an array into a new array.
The slice() method creates a new array. It does not remove any elements from the source array.
The slice() method can take two arguments and selects elements from the start argument, and up to (but not including) the end argument.
If the end argument is omitted, the slice() method slices out the rest of the array.

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
let citrus = fruits.slice(3);
console.log(fruits);//prints ["Banana", "Orange", "Lemon", "Apple", "Mango"]
console.log(citrus);//prints ["Apple", "Mango"]
citrus = fruits.slice(1, 3);
console.log(citrus);//prints ["Orange", "Lemon"]
```

Sorting and Reversing Arrays

The sort() method sorts an array alphabetically for both strings and number elements
The reverse() method reverses the elements in an array by sorting in descending order:

Sorting Strings

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();          // Sorts the elements of fruits
console.log(fruits); //prints ["Apple", "Banana", "Mango", "Orange"]
fruits.reverse();
console.log(fruits); //prints ["Orange", "Mango", "Banana", "Apple"]
```

Sorting Numbers

By default, the sort() function sorts values as strings
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 results when sorting numbers.
You can fix this by providing a compare function
Sort ascending: array_name.sort(function(a, b){return a - b})
Sort descending: array_name.sort(function(a, b){return b - a})

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
console.log(points); // prints [1, 5, 10, 25, 40, 100]
points.sort(function(a, b){return b - a});
console.log(points); // [100, 40, 25, 10, 5, 1]
```

indexOf

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];
//gets the index of the first instnace of orange, or -1 otherwise
let position = fruits.indexOf("Orange");
console.log(position); //prints 1
```

Array Iteration

map()

The map() method creates a new array by performing a function on each array element.
The map() method does not execute the function for array elements without values.
The map() method does not change the original array

```
const numbers1 = [45, 4, 9, 16, 25];
const numbers2 = numbers1.map(myFunction);
/*
line 2 is the same as the following:
const numbers2 = numbers1.map(function(item, index, array){
  return item*2;
})
*/
console.log(numbers1); //prints [45, 4, 9, 16, 25]
console.log(numbers2); //prints [90, 8, 18, 32, 50]

function myFunction(item, index, array) {
  return item * 2;
}
```

filter()

The filter() method creates a new array with array elements that passes a test.

```
const numbers1 = [45, 4, 9, 16, 25];
const over18 = numbers1.filter(myFunction);
/*
line 2 is the same as the following:
const over18 = numbers1.map(function(item, index, array){
  return item >18;
})
*/
console.log(numbers1); //prints [45, 4, 9, 16, 25]
console.log(over18); //prints [45, 25]

function myFunction(item, index, array) {
  return item > 18;
  //only items that are greater than 18 will be kept while items <= 18 will be removed
}
```

reduce()

The reduce() method runs a function on each array element to return a single value.
The reduce() method works from left-to-right in the array.

```
const numbers = [1,2,3];
let sum = numbers.reduce(myFunction, 0); //0 is the inital value of sum
/*
line 2 is the same as the following:
let sum = numbers.reduce(function(sum, item){
  return sum + item;
})
*/
console.log(numbers); //prints [1,2,3]
console.log(sum); //prints 6
function myFunction(sum, item) {
  return sum + item;
}
/*
explantion for output

myFunction for item index 0 is ran
sum = 0  since its the inital value
item = 1 since its the value of the item at index 0
myFunction(0,1) returns 1 which is the new value of sum

myFunction for item index 1 is ran
sum = 1  since it was the return value of myFunction for item index 0
item = 2 since its the value of the item at index 1
myFunction(1,2) returns 3 which is the new value of sum

myFunction for item index 2 is ran
sum = 3  since it was the return value of myFunction for item index 1
item = 3 since its the value of the item at index 2
myFunction(3,3) returns 3 which is the final return
*/
```

Objects

Object Definition

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue",
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
console.log(person);
//prints {firstName: "John", lastName: "Doe", age: 50, eyeColor: "blue", fullName: f}
```

This Keyword

In a function definition, this refers to the "owner" of the function.
In the example above, this is the person object that "owns" the fullName function.
In other words, this.firstName means the firstName property of this object.

Accessing Object Properties

let name1 = objectName.propertyName
let name2 = objectName["propertyName"]

Accessing Object Methods

let fullname = objectName.methodName()

Property Shorthand

```
let cat = 'Miaow';
let dog = 'Woof';
let bird = 'Peet peet';

let someObject = {
  cat,
  dog,
  bird
}

console.log(someObject);
//prints {cat: "Miaow", dog: "Woof", bird: "Peet peet"}
```

Destructuring

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};

/*
let firstName = person.firstName;
let lastName = person.lastName;
instead of doing the two lines above, we can do the line below
*/
//fname and lname are new variables that
//take on the value of person.firstName and person.lastName respectively
let { firstName: fname, lastName: lname } = person;
console.log(fname); // 'John'
console.log(lname); // 'Doe'
```

Declaring variables that have the same names as the properties of the object

```
//If the variables have the same names as the properties of the object,
//you can make the code more concise as follows
let { firstName, lastName } = person;
console.log(firstName); // 'John'
console.log(lastName); // 'Doe'
//When you assign a property that does not exist to a variable using the object destructuring,
//the variable is set to undefined. For example:
let { firstName, lastName, middleName } = person;
console.log(middleName); // undefined
```

Default values

```
//assign a default value to the variable when the property of an object doesn't exist.
//if a default value is assigned to a variable and the property of an object does exist,
//it takes the property of an object
let { firstName, lastName='yo', middleName = '', currentAge: age = 18 } = person;
console.log(lastName); //'Doe'
console.log(middleName); // ''
console.log(age); // 28
```

Nested object destructuring

```
let employee = {
    id: 1001,
    name: {
        firstName: 'John',
        lastName: 'Doe'
    }
};
let {id: new_ID_var, name: {firstName,lastName}} = employee;
console.log(new_ID_var); //1001
console.log(firstName); // John
console.log(lastName); // Doe
```

Spread Operator

```
const address = {
  city: 'LA',
  country: 'USA',
  postCode: 'LA44'
}
const name = {
  firstName: 'Joe',
  lastName: 'Jones'
}
const name2 = {
  firstName: 'Joe2',
  lastName: 'Jones2'
}
const person = {...address, ...name, ...name2};
console.log(person);
//{city: "LA", country: "USA", postCode: "LA44", firstName: "Joe2", lastName: "Jones2"}
//notice the firstName and lastName take the value from name2 as opposed to name1
```

Objects vs Arrays

Arrays use numbered indexes while objects used name indexes

String Methods

Length

```
let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
text.length;     // Will return 26
```

Escape Characters

Code	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

Code	Result
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Horizontal Tabulator
\v	Vertical Tabulator

Literals Vs Objects

Normally, JavaScript strings are primitive values, created from literals, but strings can also be defined as objects with the keyword new.

```
let x = "John";
let y = new String("John");
console.log(typeof x);// typeof x will return string
console.log(typeof y);// typeof y will return object
```

Comparing Strings and Objects/Strings

The == operator returns true if the values of two strings or objects are equal

The === operator returns true if the values and data types are equal

```
let x = "John";
let z = 'John';
let y = new String("John");
console.log(x==z); //true
console.log(x==y); //true
console.log(x===z);//true
console.log(x===y);//false
```

Comparing Objects and Objects

```
let x = new String("John");
let y = new String("John");
console.log(x==y); //false
console.log(x===y); //false
```

Slice()

extracts a part of a string and returns the extracted part in a new string

```
let str = "Apple, Banana, Kiwi";
let s1 = str.slice(7)
let s2 = str.slice(7, 13)
console.log(s1); //Banana, Kiwi
console.log(s2); //Banana
```

Replace()

The replace() method replaces a the first instance of a specified value with another value in a string
The replace() method is case sensitive

```
let text = "Please visit Microsoft! Microsoft";
let newText = text.replace("Microsoft", "W3Schools");
console.log(newText); //Please visit W3Schools! Microsoft
```

To replace case insensitive, use a regular expression with an /i flag (insensitive)

```
let text = "Please visit Microsoft!";
let newText = text.replace(/MICROSOFT/i, "W3Schools");
console.log(newText); //Please visit W3Schools!
```

To replace all matches, use a regular expression with a /g flag (global match)

```
let text = "Please visit Microsoft and Microsoft!";
let newText = text.replace(/Microsoft/g, "W3Schools");
console.log(newText); //Please visit W3Schools and W3Schools!
```

charAt()

The charAt() method returns the character at a specified index (position) in a string

```
let text = "HELLO WORLD";
text.charAt(0)            // Returns H
```

indexOf()

The indexOf() method returns the index of (the position of) the first occurrence of a specified text in a string, or -1 if not found

```
let str = "Please locate where 'locate' occurs!";
str.indexOf("locate")     // Returns 7
```

lastIndexOf()

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

```
let str = "Please locate where 'locate' occurs!";
str.lastIndexOf("locate")     // Returns 21
```

Convert String to Array

```
let text = "a,b,c,d,e,f";
const a1 = text.split(",");
const a2 = text.split("");
const a3 = text.split();
console.log(a1);//["a", "b", "c", "d", "e", "f"]
console.log(a2);//["a", ",", "b", ",", "c", ",", "d", ",", "e", ",", "f"]
console.log(a3);//["a,b,c,d,e,f"]
```

Random

Math.random() returns a random number between 0 (inclusive) and 1 (exclusive)
Ex: Return a random integer between 1 and 10: Math.floor((Math.random() * 10) + 1);

Immutable Vs Mutable Data Types

Immutable: Numbers, Strings, Null, Undefined, Booleans

```
let a = 123
let b = a;
a++;
console.log(a); //124
console.log(b); //123
```

Mutable: Arrays, Functions, Classes, Maps, Sets

```
const a = [1,2,3]
const b = a;
a.push(4)
console.log(b);//[1,2,3,4]
```

If Statements

```
if (condition1) {
  //  block of code to be executed if condition1 is true
} else if (condition2) {
  //  block of code to be executed if the condition1 is false and condition2 is true
} else {
  //  block of code to be executed if the condition1 is false and condition2 is false
}
```

Logical Operators in Javascript such as && or || evaluate from left to right and they short circuit. Short circuiting means that in JavaScript when we are evaluating an AND expression (&&), if the first operand is false, JavaScript will short-circuit and not even look at the second operand.
Through short circuiting, the following two blocks of code are the same.

```
if (online){
  getData();
}
```

```
online && getData();
```

In the left block of code, if online is true, getData is run. If online is false, getData isn't even looked at. In the right block of code, it reads from left to right so it first checks if online is true. If online is false, it short circuits and doesn't even read the getData(). If online is true, it runs getData(). Note that getData() is any function and doesn't have to return a boolean.

```
var online = true;

if (online){//prints 'hi'
  print();
}
online && print();//prints 'hi'

function print(){
  console.log('hi');
}
```

Ternary Operator

The syntax for a ternary operator is condition ? expr_if_cond_is_true : expr_if_cond_is_false

```
var age = 19;

// condition ? expr_if_true : expr_if_false
console.log((age>=18) ? 'adult' : 'child'); //prints adult
/*
line 4 is the same as the following:
if (age>=18) {
  console.log('adult');
}
else{
  console.log('child');
}
*/
```

Multiple Operations Per Condition

```
var age = 19;
var stop;
age>18?(
  console.log('stop is false'),
  stop = false
):(
  console.log('stop is true'),
  stop = true
)
//prints stop is false
```

Nested Ternary Operations

```
var firstcheck = false, secondcheck = false;
var access = firstcheck ? 'Output1' : secondcheck ? 'Output2':'Output3';
/*
check if firstcheck is true or false
if firstcheck is true, it runs Output 1 and stops
if firstcheck is false, it runs secondcheck ? 'Output2':'Output3'
since firstcheck is false, it then checks if secondcheck is true or false
if secondcheck is true, it runs Output2 and stops
if secondcheck is false, it runs Output3 and stops
*/
console.log(access); //prints output 3
```

Loops

For Loops

```
let str = "";
for (let i =0; i<10; i++){
  str+=i;
}
console.log(str); //prints 0123456789
```

While Loops

```
let str = "";
let i =0;
while (i<10){
  str+=i;
  i++;
}
console.log(str); //prints 0123456789
```

Break and Continue

```
let str = "";
let i =0;
while (i<10){
  i++;
  if (i==3){  // if i is 3, go to next iteration of while loop
    continue;
  }
  str+=i;
  if (i>5){ // if i is greater than 5, exit the loop
    break;
  }
}
console.log(str); //prints 12456
```

Function

Declaration

The function declaration (function statement) defines a function with the specified parameters.
Function declarations in JavaScript are hoisted to the top of the enclosing function or global scope. You can use the function before you declared it

```
console.log(toCelsius(31)); //prints -0.5555555555555556
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}
```

Expression/Anonymous

A function expression can be stored in a variable

Definition

The function keyword can be used to define a function inside an expression.

```
const getRectArea = function(width, height) {
  return width * height;
};
```

Hoisting

Function expressions in JavaScript are not hoisted, unlike function declarations. You can't use function expressions before you create them.

```
//console.log(f(31)); if this code was uncommented, it returns TypeError
var f = function(fahrenheit) {
  return (5/9) * (fahrenheit-32);
};
console.log(f(31)); //returns -0.5555555555555556
```

Named Function Expression

```
let math = {
  'fact': function factorial(n) {
    console.log(n)
    if (n <= 1) {
      return 1;
    }
    return n * factorial(n - 1);
  },
  'hi': function hi(n) {
    console.log('hi');
  }
};
math.hi(); //prints hi
math.fact(3); //prints 3;2;1;
```

Arrow Functions

Arrow functions allow us to write shorter function syntax

```
//function declaration
function hi1(){
  console.log('hi');
  console.log('1');
}
//function expression
let hi2 = function(){
  console.log('hi');
  console.log('2');
}
//arrow function for function declaration
//syntax: funcName = () => {line1; line 2};
hi3 = () => {console.log('hi'); console.log('3')};
//arrow function for function expression
//syntax: let funcName = () => {line1; line 2};
let hi4 = () => {console.log('hi'); console.log('4')};
hi1(); //prints hi; 1
hi2(); //prints hi; 2
hi3(); //prints hi; 3
hi4(); //prints hi; 4
```

Template Literals

Template Literals use back-ticks (``) rather than the quotes ("") to define a string.
With template literals, you can use both single and double quotes inside a string, multiline strings, and interpolate variables and expressions into strings.

```
let firstName = "John";
let lastName = "Doe";
let text = `He's often
called "${firstName} ${lastName}"
who is ${2+2} years old
`;
console.log(text);
```

```
He's often
called "John Doe"
who is 4 years old
```

Error Handling

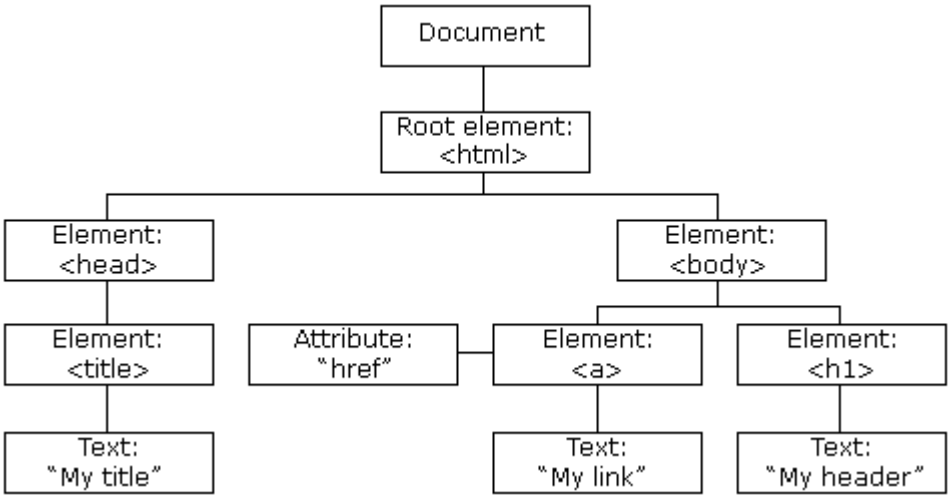
```
try {
  Block of code to try
}
catch(err) {
  Block of code to handle errors
}
finally {
  Block of code to be executed regardless of the try / catch result
}
```

Ex:

```
let x = document.getElementById("demo").value;
try {
  if(x == "") throw "is empty";
  if(isNaN(x)) throw "is not a number";
  x = Number(x);
  if(x > 10) throw "is too high";
  if(x < 5) throw "is too low";
}
catch(err) {
  message.innerHTML = "Error: " + err + ".";
}
finally {
  document.getElementById("demo").value = "";
}
```

DOM (Document Object Model)

The HTML DOM model is constructed as a tree of Objects



HTML DOM properties are values (of HTML Elements) that you can set or change.

HTML DOM methods are actions you can perform (on HTML Elements)

Finding HTML Elements
Method
document.getElementById("id_name")
document.getElementsByTagName("p")
document.getElementsByClassName("class_name")

Changing HTML Elements	
Property	**Examples**
element.innerHTML = new html content	document.getElementById("p1").innerHTML = "New text!";
element.attribute = new value	document.getElementById("myImage").src = "landscape.jpg";
element.style.property = new style	document.getElementById("p2").style.color = "blue";
Method	
element.setAttribute(attribute, value)	

Adding and Deleting Elements
Method
document.createElement(element)
document.removeChild(element)
document.appendChild(element)
document.replaceChild(new, old)

Adding Events Handlers
Method
document.getElementById(id).onclick = function(){code} Examples: document.getElementById("myBtn").onclick = displayDate;

Events
onclick
onmousedown
onmouseup
onload
onunload
onchange
onmouseover
onmouseout

```
<!DOCTYPE html>
<html>
<body>
  <h1 onclick="this.innerHTML = 'Ooops!'">Click on this text!</h1>
</body>
</html>
```

```html
<!DOCTYPE html>
<html>
<body>
  <h1 onclick="changeText(this)">Click on this text!</h1>
  <script>
    function changeText(id) {
      id.innerHTML = "Ooops!";
    }
  </script>
</body>
</html>
```

Event Listener

Syntax: element.addEventListener(event, function, useCapture);

- The first parameter is the type of the event (like "click" or "mousedown" or any other HTML DOM Event.). Note that you don't use the "on" prefix for the event; use "click" instead of "onclick".
- The second parameter is the function we want to call when the event occurs.
- The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

```html
<!DOCTYPE html>
<html>
<body>
  <h2>JavaScript addEventListener()</h2>
  <p>This example uses the addEventListener() method to execute a function when a user clicks on a button.</p>
  <button id="myBtn">Try it</button>
  <script>
    document.getElementById("myBtn").addEventListener("click", myFunction);
    function myFunction() {
      alert ("Hello World!");
    }
  </script>
</body>
</html>
```

Forms

Get the value of an input field through document.getElementById("input_field_id").value;

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h2>JavaScript Validation</h2>
    <p>Please input a number between 1 and 10:</p>
    <input id="numb">
    <button type="button" onclick="myFunction()">Submit</button>
    <p id="demo"></p>
    <script src = 'script.js'></script>
  </body>
</html>
```

```
function myFunction() {
  // Get the value of the input field with id="numb"
  let x = document.getElementById("numb").value;
  // If x is Not a Number or less than one or greater than 10
  let text;
  if (isNaN(x) || x < 1 || x > 10) {
    text = "Input not valid";
  } else {
    text = "Input OK";
  }
  document.getElementById("demo").innerHTML = text;
}
```

Classes

A JavaScript class is not an object. It is a template for JavaScript objects.

```
class ClassName{
  constructor(prop1, prop2) {
    this.prop1 = prop1;
    this.prop2 = prop2;
  }
  method_1(){
    console.log(this.prop1);
  }
  method_2(){
    console.log(this.prop2);
  }
}
let obj1 = new ClassName(1,2);
obj1.method_2(); //prints 2
```

It has to have the exact name "constructor". The constructor method is called automatically when a new object is created. If you do not define a constructor method, JavaScript will add an empty constructor method.

Inheritance

To create a class inheritance, use the extends keyword.

A class created with a class inherits and inherits all the methods from another class.

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

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}
class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model;
  }
}
let model = new Model("Ford", "Mustang");
document.getElementById("demo").innerHTML = model.show();
```

Getters and Setters

To add getters and setters in the class, use the get and set keywords.

The name of the getter/setter method cannot be the same as the name of the property, in this case carname.

Many programmers use an underscore character _ before the property name to separate the getter/setter from the actual property. Notice that setters and getter methods can have the same name.

To use a getter or setter, use the same syntax as when you set a property value, without parentheses.

```
class Car {
  constructor(brand) {
    this._carname = brand;
  }
  get carname() {
    return this._carname;
  }
  set carname(x) {
    this._carname = x;
  }
}
let myCar = new Car("Ford");
//To use a setter, use the method without parentheses as below
myCar.carname = "Volvo";
//To use a getter, use the method without parentheses as below
console.log(myCar.carname); //prints Volvo
```

Hoisting

Cass declarations are not hoisted so you must declare a class before you can use it.

```
//You cannot use the class yet.
//myCar = new Car("Ford")
//This would raise an error.
class Car {
  constructor(brand) {
    this.carname = brand;
  }
}
//Now you can use the class:
let myCar = new Car("Ford")
```

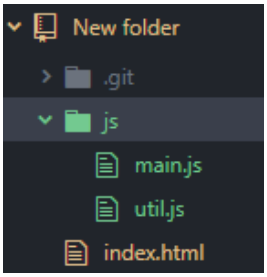
Static Class Properties and Methods

```
class ClassWithStaticMethod {
  static staticProperty = 'someValue';
  static staticMethod() {
    return 'static method has been called.';
  }
  static staticMethod2() {
    this.staticProperty+='1';
    return this.staticProperty;
  }
}
console.log(ClassWithStaticMethod.staticProperty);//prints someValue
console.log(ClassWithStaticMethod.staticMethod());//prints static method has been called.
console.log(ClassWithStaticMethod.staticMethod2());//prints someValue1
```

Modules

Modules can only be used in live servers, by directly clicking the file on our desktop.

Assume we have the following directory structure where main.js imports code from util.js.



In the html, make sure to link to the main js file with type = 'module'.

```html
<!DOCTYPE html>
<html>
<body>
  <h2>JavaScript addEventListener()</h2>
  <p>This example uses the addEventListener() method to
  <button id="myBtn">Try it</button>
  <script type = 'module' src = 'js/main.js'></script>
</body>
</html>
```

Single Export

In the utils.js file, we have the following. We have a const and function declaration as normal, but we stick the word export in front. This makes the const and function exportable.

```js
export const name = 'Dom';
export function double(n){
  return n*2;
}
```

We can also do the following

```js
const name = 'Dom';
function double(n){
  return n*2;
}
export {name, double};
```

In the main.js file, we do import {item1, item2, item3, …} from 'filePath.js';

```js
import {double, name} from 'utils.js';
console.log(double(5));
console.log(name);
```

We can also change the names of imported items

```js
import {double as utilsDouble, name} from 'utils.js';
console.log(utilsDouble(5));
console.log(name);
```

Multiple Export

We still have to put export for each thing we want to export in the utils.js file.
But in the html file, we can do * instead of listing each item we want to import.

```js
import * as Utils from 'utils.js';
console.log(Utils.double(5));
console.log(Utils.name);
```

Default Export

The export default keyword means the item is the default export for the module. One module can only have 1 default export.

main.js util.js

```
export const name = 'Dom';
export default function double(n){
  return n*2;
}
```

In the main.js folder, we can do the following. Notice there is no item called bottle in the utls.js module. Thus, bottle will be used for the default export which is the function.

```
import bottle from 'utils.js';
/*
line 1 is the same as the line below
import {default as bottle} from 'utils.js';
*/
bottle();
```

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.

Consider the following 2 scenarios that don't use callbacks that allow us to do a calculation, and then display the result.

Scenario 1: You could call a calculator function (myCalculator), save the result, and then call another function (myDisplayer) to display the result:

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2) {
  let sum = num1 + num2;
  return sum;
}
let result = myCalculator(5, 5);
myDisplayer(result);
```

However, the problem is that you have to call two functions to display the result.

Scenario 2: You could call a calculator function (myCalculator), and let the calculator function call the display function (myDisplayer):

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2) {
  let sum = num1 + num2;
  myDisplayer(sum);
}
myCalculator(5, 5);
```

The problem is that you cannot prevent the calculator function from displaying the result.

Now, we will use callbacks.
You could call the calculator function (myCalculator) with a callback, and let the calculator function run the callback after the calculation is finished.

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}
myCalculator(5, 5, myDisplayer);
```

In the example above, myDisplayer is the name of a function and is passed to myCalculator() as an argument.
When you pass a function as an argument, remember not to use parenthesis.
Right: myCalculator(5, 5, myDisplayer);
Wrong: myCalculator(5, 5, myDisplayer());

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

Asynchronous Functions

Functions running in parallel with other functions are called asynchronous.
setTimeout()
A good example is JavaScript setTimeout() to specify a callback function to be executed on time-out.

```
setTimeout(myFunction, 3000);
function myFunction() {
  document.getElementById("demo").innerHTML = "I love You !!";
}
```

In the example above, myFunction is used as a callback.
The function (the function name) is passed to setTimeout() as an argument.
3000 is the number of milliseconds before time-out, so myFunction() will be called after 3 seconds.
When you pass a function as an argument, remember not to use parenthesis.

setInterval()

you can specify a callback function to be executed every few seconds(the interval time)

```
setInterval(myFunction, 1000);
function myFunction() {
  let d = new Date();
  document.getElementById("demo").innerHTML=
  d.getHours() + ":" +
  d.getMinutes() + ":" +
  d.getSeconds();
}
```

In the example above, myFunction is used as a callback.

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

1000 is the number of milliseconds between intervals, so myFunction() will be called every second.

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.