JavaScript Guide

1. Grammar and types

- JavaScript is case-sensitive and uses the Unicode character set. For example, the word Früh could be used as a variable name.

Declarations

There are three kinds of declarations in JavaScript.

`var`

Declares a variable, optionally initializing it to a value.

`let`

Declares a block-scoped, local variable, optionally initializing it to a value.

`const`

Declares a block-scoped, read-only named constant.

>> Naming Rules

- Start with letter, underscore or dollar sign $
- the names can include letters A-Z, a-z or numbers 0-9 or underscore in between

>> Declaring variables

- var x = 43 //
- let y = 13 // block scoped declaration
- x = 432 // **note:** creates an undeclared **global** variable. Not good to use.

>> Points on Variables

- A variable declared but not initialized will have value of **undefined**
  - var a;
  - console.log('The value of a is ' + a); // The value of a is undefined
- If a variable is not declared but tried to access results in
  - // Uncaught ReferenceError: c is not defined
- You can use undefined to determine whether a variable has a value. In the following code, the variable input is not assigned a value, and the if statement evaluates to true.

```
var input;
if (input === undefined) {
  doThis();
} else {
  doThat();
}
```

- The undefined value converts to NaN when used in numeric context.

```
var a;
a + 2;  // Evaluates to NaN
```

- The null value behaves as 0 in numeric contexts and as false in boolean contexts. For example:

```
var n = null;
console.log(n * 32); // Will log 0 to the console
```

>> Variable Scope

- Variable declared outside of a function - global variable.
- Variable declared within a function - local variable.
- var - global, let - local. (from ES6)

```
if (true) {
  var x = 5;
}
console.log(x);  // x is 5
```

```
if (true) {
  let y = 5;
}
console.log(y);  // ReferenceError: y is not defined
```

>> Variable Hoisting

- Any variable that is declared later in the code will be interpreted as if it is declared at the top.

```
x = 32  // variable initialized
console.log(x)  // variable used here. o/p: 32
var x   // variable declared later in the code
```

- **Note: works only for 'var'.**

```
console.log(x === undefined); // true
var x = 3;
```

The above example will be interpreted the same as:

```
var x;
console.log(x === undefined); // true
x = 3;
```

- The hoisted variable remains **undefined** until the point it is initialized.

```
console.log(x)       // undefined.
var x = 32           // x is initialized here
console.log(x);      // 32
```

- **Note:** Always declare the variable before using - best thing to do.

>> Function Hoisting

- **Function declaration** - similar to a variable declaration.
    - function keyword, function name, params. (opt), function body.

```
function add(num1, num2) {
    return num1 + num2;
}
console.log(add(3, 3)) // returns 6
```

- **Function expression**
    - declaration type, function name, function keyword, params., function body.

```
var add = function (num1, num2) {
    return num1 + num2;
};
console.log(add(3, 4)); // returns 7
```

- **Function declaration** SUPPORTS hoisting.

```
console.log(add(3, 9)) // returns 12
function add(num1, num2) {  // this function gets hoisted.
    return num1 + num2;
}
```

the above function will be interpreted as,

```
function add(num1, num2) {
    return num1 + num2;
}
console.log(add(3, 9)) // returns 12
```

- **Function Expression** DOESNOT SUPPORT hoisting.

```
var add = function (num1, num2) {
    return num1 + num2;
};
console.log(add(3, 4)); // returns 7
```

but,

```
console.log(add(3, 4)); // TypeError: add is not a function
var add = function (num1, num2) {
    return num1 + num2;
};
```

>> Global Variables (Pending)

>> Constants

- A constant cannot change value through assignment or be re-declared while the script is running. It must be initialized to a value.
- Scope is limited to the block.
- You cannot declare a constant with the same name as a function or variable in the same scope.

```
1  // THIS WILL CAUSE AN ERROR
2  function f() {};
3  const f = 5;
4
5  // THIS WILL CAUSE AN ERROR TOO
6  function f() {
7    const g = 5;
8    var g;
9
10   //statements
11 }
```

- However, a **CONSTANT OBJECT** can change its properties during the execution or after the first initialization.

```
const myObj = {
    name: 'me',
    age: 23,
    role: 'dev',
    isActive: true,
}
console.log(myObj.age); // returns 23

myObj.age = 22

console.log(myObj.age); // returns 22
```

- A **CONSTANT ARRAY** can also change its elements by push and pop methods.

```
const myArray = ['alpha', 'beta', 'gamma']
console.log(myArray);       // [ 'alpha', 'beta', 'gamma' ]

myArray.push('delta')
console.log(myArray);       // [ 'alpha', 'beta', 'gamma', 'delta' ]
```

- The problem arises when the same object name or array name is declared again.

```
const myArray = ['alpha', 'beta', 'gamma']
console.log(myArray);       // [ 'alpha', 'beta', 'gamma' ]

myArray.push('delta')
console.log(myArray);       // [ 'alpha', 'beta', 'gamma', 'delta' ]

let myArray = 34        // SyntaxError: Identifier 'myArray' has already been declared
```

==========

Data structures and types

>> Data types

- Six data types that are primitives:
  - Boolean. `true` and `false`.
  - null. A special keyword denoting a null value. Because JavaScript is case-sensitive, `null` is not the same as `Null`, `NULL`, or any other variant.
  - undefined. A top-level property whose value is not defined.
  - Number. An integer or floating point number. For example: `42` or `3.14159`.
  - String. A sequence of characters that represent a text value. For example: "Howdy"
  - Symbol (new in ECMAScript 2015). A data type whose instances are unique and immutable.
- and Object

>> Data type conversions

- JavaScript is a dynamically typed language. That means you don't have to specify the data type of a variable when you declare it, and data types are converted automatically as needed during script execution.

```
let xie = 23
console.log(xie);       // 23

xie = 'hey'
console.log(xie);       // hey
```

- In a line involving numbers and strings, the + operator converts the number to a string.

```
x = 'The answer is ' + 42 // "The answer is 42"
y = 42 + ' is the answer' // "42 is the answer"
```

- But, in statements involving other operators, JavaScript does not convert numeric values to strings
    - look for **21. Coercion** in JS >> LCO >> Basics.docx

```
'37' - 7 // 30
'37' + 7 // "377"
```

>> Converting strings to numbers

- If a value representing a number is in memory as a string, then these methods are used for conversion.
- **parseInt(string, radix);** and **parseFloat(string, radix);**
- **string -** any number which is in the string format
- **radix -** the base of the numerical system used in that string

```
let w = parseInt('23423/23')
console.log(w);     // 23423

let w1 = parseInt("567.23")
console.log(w1);    // 567

let w2 = parseInt("okay 33")
console.log(w2);    // NaN

let x = parseInt("A", 16)
console.log(x);     // 10

let x1 = parseInt("000023")
console.log(x1);    // 23

let x2 = parseInt("d", 16)
console.log(x2);    // 13

let x3 = parseInt("97 years old")
console.log(x3);    // 97

let x4 = parseInt("0x11")
console.log(x4);    // 17

let x5 = parseInt("0x10")
console.log(x5);    // 16
```

- **Note:** Because some numbers include the e character in their string representation (e.g. 6.022e23), using parseInt to truncate numeric values will produce unexpected results when used on very large or very small numbers. **parseInt should not be used as a substitute for Math.floor()**.
- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt

An alternative method of retrieving a number from a string is with the `+` (unary plus) operator:

```
'1.1' + '1.1' // '1.11.1'
(+'1.1') + (+'1.1') // 2.2
// Note: the parentheses are added for clarity, not required.
```

==========

Literals

>> Array Literals

- Extra commas in an array literal - You do not have to specify all elements in an array literal. If you put two commas in a row, the array is created with undefined for the unspecified elements.
- If you include a trailing comma at the end of the list of elements, the comma is ignored.
- **Note :** Trailing commas can create errors in older browser versions and it is a best practice to remove them.

```
let xie = ['hey', , 'okay', 23, false, , , , true,]
console.log(xie);
//  // [ 'hey', <1 empty item>, 'okay', 23, false, <3 empty items>, true ]
```

>> Boolean Literals

- Read further: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_Types#Boolean_literals

>> Numeric Literals

- A decimal integer literal consists of a sequence of digits without a leading 0 (zero).
- A leading 0 (zero) on an integer literal, or a leading 0o (or 0O) indicates it is in octal. Octal integers can include only the digits 0-7.
- A leading 0x (or 0X) indicates a hexadecimal integer literal. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F. (The case of a character does not change its value, e.g. 0xa = 0xA = 10 and 0xf = 0xF = 15.)
- A leading 0b (or 0B) indicates a binary integer literal. Binary integers can only include the digits 0 and 1.

```
0, 117 and -345 (decimal, base 10)
015, 0001 and -0o77 (octal, base 8)
0x1123, 0x00111 and -0xF1A7 (hexadecimal, "hex" or base 16)
0b11, 0b0011 and -0b11 (binary, base 2)
```

>> Floating-point Literals

- A floating point has the following parts,
  - A decimal integer which can be signed (preceded by "+" or "-"),
  - A decimal point ("."),
  - A fraction (another decimal number),
  - An exponent.
- A floating-point literal must have at least one digit and either a decimal point or "e" (or "E").

- Syntax: [(+|-)][digits].[digits][(E|e)[(+|-)]digits]
- Example: -3.1E+12

>> Object Literals

- Object

```
var myObj = {
    name: 'vinay',
    isActive: true,
    22: 'Red',
    "": 'nothing',
}
```

- Adding new properties to an existing object

```
// ADDING NEW VALUES / PROPERTIES TO AN OBJECT

myObj.age = '22'
console.log(myObj);
//  { name: 'vinay', isActive: true, age: '22' }

myObj["role"] = 'dev'
console.log(myObj);
//  { name: 'vinay', isActive: true, age: '22', role: 'dev' }
```

- Property names other than typical strings,

```
myObj[''] = 'empty again!'
//  // THE PREVIOUS EMPTY PROPERTY GETS OVERWRITTEN BY THIS
console.log(myObj);

myObj[54] = 'Oh! A number!!'
console.log(myObj);
```

```
{ '22': 'Red',
  '54': 'Oh! A number!!',
  name: 'vinay',
  isActive: true,
  '': 'empty again!' }
```

- Accessing such properties

```
console.log(myObj[54]);     // Oh! A number!!
console.log(myObj['']);     // empty again!
```

- Accessing nested objects,

```
let myObj2 = {
    name: 'vin',
    address: {
        firstLine: '#351, 1st main, 2nd cross',
        secondLine: 'Kuvempu Nagar',
        Area: 'Ramamurthy Nagar',
        City: 'Bangalore',
        func: function () {
            myObj.isActive = false
        }
    }
}

console.log(myObj2.address.City);         // Bangalore
```

>> String Literals

- Examples,

```
'foo'
"bar"
'1234'
'one line \n another line'
"John's cat"
```

- **String interpolation / template strings / template literals**

```
// String interpolation
var name = 'Bob', time = 'today';
`Hello ${name}, how are you ${time}?`
```

- Cannot display the contents of an object when the object is being displayed inside ${objName} along with some other string.

```
let objName = {
    name: 'vin',
    age: 22
}

console.log(`Details are: ${objName}`); //Details are: [object Object]
console.log(objName);    // { name: 'vin', age: 22 }
```

- Special and escape characters: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_Types#String_literals
- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x

2. Control flow and Error Handling

>> Block statements

```
while (x < 10) {
  x++;
}
```

- Block statements do not define a scope.

```
var x = 234;
{
    let x = 3
}
console.log(x); // 234
```

- Declaration type **var** makes the variable global, so any changes to the variable inside a block will effect on the outside also.

```
var x = 234;
{
    var x = 3
}
console.log(x); // 3
```

==========

Conditional statements

- **Falsy** values
  - `false`
  - `undefined`
  - `null`
  - `0`
  - `NaN`
  - the empty string (`""`)
- All other values, including all objects, evaluate to **true** when passed to a conditional statement.
- Switch:
  - expression to check for mentioned in **switch (*here*)**
  - cases and default statements
  - **break**'s are not compulsory, break ensures control goes out of switch after required **case** is found.
  - If break is not used, control goes on until the end of the switch or until some break statement is found.
  - **NOTE:** By convention, the default clause is the last clause, but it does not need to be so.

```
let x = 3;
switch (x) {
    case 1:
    case 2:
    default: console.log('haha');
    case 3: console.log('okay');
}
```

==========

Exception handling statements

- Refer : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#Exception_handling_statements

>> throw statement

- When you throw an exception, you specify the expression containing the value to be thrown (custom error messages)

>> try-catch

- try block, which contains one or more statements, and a catch block, containing statements that specify what to do if an exception is thrown in the try block.
- If no exception is thrown in the try block, the catch block is skipped. The finally block executes after the try and catch blocks execute
- You can use a catch block to handle all exceptions that may be generated in the try block.

```
catch (catchID) {
  statements
}
```

- The catch block specifies an identifier (catchID in the preceding syntax) that holds the value specified by the throw statement; you can use this identifier to get information about the exception that was thrown. JavaScript creates this identifier when the catch block is entered; the identifier lasts only for the duration of the catch block; after the catch block finishes executing, the identifier is no longer available.

>> finally block

- The finally block contains statements to execute after the try and catch blocks execute but before the statements following the try...catch statement. The finally block executes whether or not an exception is thrown.
- You can use the finally block to make your script fail gracefully when an exception occurs; for example, you may need to release a resource that your script has tied up.
- The following example opens a file and then executes statements that use the file (server-side JavaScript allows you to access files). If an exception is thrown while the file is open, the finally block closes the file before the script fails.

```
openMyFile();
try {
  writeMyFile(theData); //This may throw an error
} catch(e) {
  handleError(e); // If we got an error we handle it
} finally {
  closeMyFile(); // always close the resource
}
```

- If the finally block returns a value, this value becomes the return value of the entire try-catch-finally production, regardless of any return statements in the try and catch blocks:
- The return values and the exceptions thrown in **try and catch** are ignored until the control reaches finally block. If there is a return or throw statement in finally block that is the final returning value from the try-catch-finally block (overriding other returns and throws)

>> Error objects

- If you are throwing your own exceptions, in order to take advantage of these properties, you can use the Error constructor.

```
function doSomethingErrorProne() {
  if (ourCodeMakesAMistake()) {
    throw (new Error('The message'));
  } else {
    doSomethingToGetAJavascriptError();
  }
}
....
try {
  doSomethingErrorProne();
} catch (e) {
  console.log(e.name); // logs 'Error'
  console.log(e.message); // logs 'The message'
}
```

>> onerror() Method

- The onerror event handler provides three pieces of information to identify the exact nature of the error,
  - Error message – The same message that the browser would display for the given error
  - URL – The file in which the error occurred
  - Line number– The line number in the given URL that caused the error

```
window.onerror = function (msg, url, line) {
    alert(`error message  ${msg}, in page/url ${url} and line ${line}`)
}
```

  - Any error shown or thrown wi

- onerror with html tags

  You can use an **onerror** method, as shown below, to display an error message in case there is any problem in loading an image.

  ```
  <img src="myimage.gif" onerror="alert('An error occurred loading the image.')" />
  ```

  You can use **onerror** with many HTML tags to display appropriate messages in case of errors.

==========

Promises

- Refer,
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#Promises
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises
- Rough idea:
  - a promise is used to run a code block/function in async.
  - It takes two arguments - one representing success of the code block and one for failure of code block.
  - You probably can define what a success is and what a failure is.
  - The two variables are used to return respective messages or whatever upon completion of the promise.
  - This can later be used in then() method of the promise.

```
var prom = new Promise(function (successVariable, failureVariable) {
    let x = Math.floor((Math.random() * (6 - 1 + 1)) + 1)
    if (x % 2 == 0) {
        successVariable(`Even`)
        successVariable(`even is kind of positive`)     // RETURNS ONLY THE FIRST REOLVE OR REJECT..
    } else {
        failureVariable('Odd')
        failureVariable('odd is always seen as very negative.. hmm. very odd')
    }

})
```

```
prom.then((xie) => console.log(`YO! ${xie}`), (yie) => console.log(`YO! ${yie}`))
/**
 * YO! Even - if success, YO! Odd - if failure of promise
 */
prom.then(x => console.log(`${x} it is`), x => console.log(`${x} it is`))
/**
 * Even it is - if success, Odd it is - if failure of promise
 */
```

>> then() method

- Refer:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then

- Rough idea:
  - then() takes two arguments - one for success and one for failure from the main promise method.
  - these two values are used accordingly as required.

=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x

3. Loops and iteration

>> for statement

```
for ([initialExpression]; [condition]; [incrementExpression])
  statement
```

- If the condition expression is omitted entirely, the condition is assumed to be true.
- initialize > check condition > statements > inc/dec > condition > repeat

>> do...while statement

```
do
  statement
while (condition);
```

>> while statement

```
while (condition)
  statement
```

>> labeled statement

```
var str = "";

loop1:
for (var i = 0; i < 5; i++) {
  if (i === 1) {
    continue loop1;
  }
  str = str + i;
}

console.log(str);
// expected output: "0234"

```

terminates/skips that particular loop with that label.

>> continue statement

- skip the current iteration
- In contrast to the break statement, continue does not terminate the execution of the loop entirely.

- **In a while loop, it jumps back to the condition. In a for loop, it jumps to the increment-expression.**

>> for...in statement

- Used over an object
- Iterates over all the properties of an object.

```
var person = { fname: "John", lname: "Doe", age: 25 };

var text = "";
var x;
for (x in person) {
    text += person[x] + " ";
    console.log(x);

}
console.log(text);
```

```
fname
lname
age
John Doe 25
```

- Gets/iterates over the names of the properties used in the object
- **Note:** If used on an array, **it gets only the index of the elements**. Not useful according to the mdn docs.

```
var person = [22, 34, 12, 345, 12, 3, 53, 2,]
console.log(person);

var text = "";
var x;
for (x in person) {
    person[x] += 0.2
}
console.log(person);
```

```
[ 22, 34, 12, 345, 12, 3, 53, 2 ]
[ 22.2, 34.2, 12.2, 345.2, 12.2, 3.2, 53.2, 2.2 ]
```

>> for...of statement

- The for...of statement creates a loop Iterating over iterable objects (including Array, Map, Set, arguments object and so on), invoking a custom iteration hook with statements to be executed for the value of each distinct property.
- Gets/iterates over the values of the elements in array

=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x

4. Functions

- A method is a function that is a property of an object.
- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function

Defining functions

- **Note:**

- Primitive parameters (such as a number) are passed to functions by value; the value is passed to the function, but if the function changes the value of the parameter, this change is not reflected globally or in the calling function
- If you pass an object (i.e. a non-primitive value, such as Array or a user-defined object) as a parameter and the function changes the object's properties, that change is visible outside the function

```
function objChanger(obj) {
    obj.age += 1
}

let x = {
    name: 'vinay',
    age: 22,
    role: 'dev',
    isActive: true,
}

console.log(`before calling the function`);
console.log(x); // name: 'vinay', age: 22, role: 'dev', isActive: true

objChanger(x)

console.log(`after calling the function`);
console.log(x); // { name: 'vinay', age: 23, role: 'dev', isActive: true }
```

- age is increased by one in the object after the function execution.

>> Function declaration

```
function square(number) {
  return number * number;
}
```

- **Function declarations** do get **hoisted.**

>> Function expression

```
var square = function(number) { return number * number; };
var x = square(4); // x gets the value 16
```

- Syntax:

```
var myFunction = function [name]([param1[, param2[, ..., paramN]]]) {
   statements
};
```

- The function **name**. Can be omitted, in which case the function is anonymous. The name is only local to the function body.
- **Note:** The **name** can be used inside the function to refer to itself, or in a debugger to identify the function in stack traces

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function

```
var factorial = function fac(n) { return n < 2 ? 1 : n * fac(n - 1); };

console.log(factorial(3));
```

- Function expressions are NOT HOISTED.
- Function expressions are convenient when passing a function as an argument to another function.
- The following example shows a map named function that should receive a function as first argument and an array as second argument.
- Function receives a function defined by a function expression and executes it for every element of the array received as a second argument.

```
function map(f, a) {
  var result = []; // Create a new Array
  var i; // Declare variable
  for (i = 0; i != a.length; i++)
    result[i] = f(a[i]);
  return result;
}
var f = function(x) {
   return x * x * x;
}
var numbers = [0, 1, 2, 5, 10];
var cube = map(f,numbers);
console.log(cube);
```

Function returns: [0, 1, 8, 125, 1000].

==========

Calling functions

- Functions must be in scope when they are called, but the function declaration can be hoisted.
- The scope of a function is the function in which it is declared, or the entire program if it is declared at the top level.
- There are other ways to call functions. There are often cases where a function needs to be called dynamically, or the number of arguments to a function vary, or in which the context of the function call needs to be set to a specific object determined at runtime. It turns out that functions are, themselves, objects, and these objects in turn have methods (see the Function object). One of these, the apply() method, can be used to achieve this goal.
- Refer:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Calling_functions
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply

==========

Function scope

- Variables defined inside a function cannot be accessed from anywhere outside the function.

```
function haha() {
    let x = 4
}

console.log(x); // ReferenceError: x is not defined
```

- A function defined in the global scope can access all variables defined in the global scope.

```
let outSideVariable = 34

function aGlobalFunction() {
    console.log(outSideVariable);
}

aGlobalFunction()   //  34. function can access outside variable
```

- A function defined inside another function can also access all variables defined in its parent function and any other variable to which the parent function has access to.

```
let outtieVar = `I'm a global variable`
function parentFunc() {
    let parentVariable = 'I belong to parent function'
    function childFunc() {
        let childVariable = `I belong to child function`
        console.log(`${outtieVar} - ${parentVariable} - ${childVariable}
        : All this returned from child function`);
    }
    return childFunc()
}

parentFunc()
/**
 * I'm a global variable - I belong to parent function -
 * I belong to child function : All this returned from child function
 */
```

==========

Scope and the function stack

>> Recursion

- A function can refer to and call itself. There are three ways for a function to refer to itself:
    - The function's name
    - arguments.callee()
    - an in-scope variable that refers to the function

```
let x = 1
let xie = function recur() {
    if (x > 10) {
        console.log(`it reached 10, so I'm returning`);
        return
    }

    else {
        console.log(x);
        x++
        recur()     // function name
        // xie()    // in-scope variable that refers to the function
        // arguments.callee()
    }
}

xie()
```

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Recursion

>> Nested functions and closures

- A nested function is private to its outer function. It also forms a closure.
- The inner function contains the scope of the outer function. It can access the variables of the parent function and the global variables that the parent function has access to.
- Outer function cannot use the arguments and variables of the inner function.
- **Note:** The inner function can be accessed only from statements in the outer function.
- Since the inner function forms a closure, you can call the outer function and specify arguments for both the outer and inner function.

```
let globalNumber = 2

function parentFunc(n1, n2) {
    let parentVariable = 4 + n1

    function childFunc(no2) {
        let childVariable = 6 + no2
        return (`${globalNumber} - ${parentVariable} - ${childVariable}
        : All this returned from child function`);
    }

    let childResult = childFunc(n2) // child function call
    return childResult
}

let i = parentFunc(10, 4)   // parent function call
console.log(i);
// 2 - 14 - 10 : All this returned from child function
```

or

```
let globalNumber = 10
function outtie(n1 = 0) {
    let parentVariable = 40
    function innie(n2) {
        let childVariable = 2
        return ((parentVariable * n1) + (childVariable * n2) + globalNumber)
    }
    return innie
    /**
     * return has to be like this to use the following
     * function calls with that type of arguements passing..
     *  */
}

/**
 * can pass arguements like this also.
 * first one is for parent function and second one for child function..
 */
console.log(outtie(100)(2000)); // 8010..

//  // another way of passing parameters to parent and child functions..
let passingValueForInner = outtie(100)  // 100 is the param. for parent function.
let result = passingValueForInner(2000) // 2000 is the param. for the child function.
console.log(result);    // 8010
```

>> Preservation of variables

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Preservation_of_variables

>> Multiply-nested functions

- Functions can be multiply-nested, i.e. a function (A) containing a function (B) containing a function (C).
- Both functions B and C form closures here, so B can access A and C can access B. In addition, since C can access B which can access A, C can also access A. Thus, the closures can contain multiple scopes.
- This is called scope chaining.

```
function A(x) {
  function B(y) {
    function C(z) {
      console.log(x + y + z);
    }
    C(3);
  }
  B(2);
}
A(1); // logs 6 (1 + 2 + 3)
```

In this example, `C` accesses `B`'s `y` and `A`'s `x`. This can be done because:

1. `B` forms a closure including `A`, i.e. `B` can access `A`'s arguments and variables.
2. `C` forms a closure including `B`.
3. Because `B`'s closure includes `A`, `C`'s closure includes `A`, `C` can access both `B` *and* `A`'s arguments and variables. In other words, `C` *chains* the scopes of `B` and `A` in that order.

The reverse, however, is not true. `A` cannot access `C`, because `A` cannot access any argument or variable of `B`, which `C` is a variable of. Thus, `C` remains private to only `B`.

>> Name conflicts

- When two arguments or variables in the scopes of a closure have the same name, there is a name conflict. More inner scopes take precedence, so the inner-most scope takes the highest precedence, while the outer-most scope takes the lowest. This is the scope chain. The first on the chain is the inner-most scope, and the last is the outer-most scope.
- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Name_conflicts

==========

Closures

- JavaScript allows for the nesting of functions and grants the inner function full access to all the variables and functions defined inside the outer function (and all other variables and functions that the outer function has access to).
- However, the outer function does not have access to the variables and functions defined inside the inner function. This provides a sort of encapsulation for the variables of the inner function.
- Since the inner function has access to the scope of the outer function, the variables and functions defined in the outer function will live longer than the duration of the outer function execution, if the inner function manages to survive beyond the life of the outer function.
- Having a parent function and multiple child functions, all acting on parent variable and parameter. A variable linking to the parent function is used to access all the child functions.

```
var createPet = function(name) {
  var sex;

  return {
    setName: function(newName) {
      name = newName;
    },

    getName: function() {
      return name;
    },

    getSex: function() {
      return sex;
    },

    setSex: function(newSex) {
      if(typeof newSex === 'string' && (newSex.toLowerCase() === 'male' ||
        newSex.toLowerCase() === 'female')) {
        sex = newSex;
      }
    }
  }
}

var pet = createPet('Vivie');
pet.getName();                  // Vivie

pet.setName('Oliver');
pet.setSex('male');
pet.getSex();                   // male
pet.getName();                  // Oliver
```

- The name variable of the outer function is accessible to the inner functions, and there is no other way to access the inner variables except through the inner functions.
- The inner variables of the inner functions act as safe stores for the outer arguments and variables. They hold "persistent" and "encapsulated" data for the inner functions to work with.
- The functions do not even have to be assigned to a variable, or have a name.
- There are a number of pitfalls to watch out for when using closures. **If an enclosed function defines a variable with the same name as the name of a variable in the outer scope, there is no way to refer to the variable in the outer scope again.**

```
var createPet = function (name) {  // The outer function defines a variable called "name".
    return {
        setName: function (name) {    // The enclosed function also defines a variable called "name".
            name = name;              // How do we access the "name" defined by the outer function?
        }
    }
}
```

==========

Using the arguments object

- The arguments of a function are maintained in an array-like object.
- Within a function, you can address the arguments passed to it as follows:

```
arguments[i]
```

- where i, is the ordinal number of the argument, starting at zero. So, the first argument passed to a function would be arguments[0]. The total number of arguments is indicated by arguments.length.
- Using the arguments object, you can call a function with more arguments than it is formally declared to accept. This is often useful if you don't know in advance how many arguments will be passed to the function.
- It is possible to access each argument using the arguments object.
- Ex: this is a function that **formally** takes a parameter to put in between other values. You can pass any number of arguments to this function, and it concatenates each argument into a string "list"

```
function myConcat(separator) {
    var result = ''; // initialize list
    var i;
    // iterate through arguments
    for (i = 1; i < arguments.length; i++) {
        // result += arguments[i] + separator;
        // result += `${arguments[i]} ${arguments[0]} `
        result += `${arguments[i]} ${separator} `
    }
    console.log(typeof (result));

    return result;
}

console.log(myConcat(`~`, `vinay`, true, 86));
//  vinay ~ true ~ 86 ~
```

- **Note:** The arguments variable is "array-like", but not an array. It is array-like in that it has a numbered index and a length property. However, it does not possess all of the array-manipulation methods.

==========

Function parameters

>> Default parameter

- In the past, the general strategy for setting defaults was to test parameter values in the body of the function and assign a value if they are undefined. If in the following example, no value is provided for b in the call, its value would be undefined when evaluating a*b and the call to multiply would have returned NaN. However, this is caught with the second line in this example:

```
function multiply(a, b) {
  b = typeof b !== 'undefined' ?  b : 1;

  return a * b;
}

multiply(5); // 5
```

- With default parameters, the check in the function body is no longer necessary. Now, you can simply put 1 as the default value for b in the function head:

```
function multiply(a, b = 1) {
  return a * b;
}

multiply(5); // 5
```

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters

>> Rest parameters

- If a function doesn't know how many parameters it's going to receive and some actions has to be done on these parameters, then this rest params. is useful.
- All the parameters passed to a function are taken and converted into an array internally by the … rest operator.
- The rest parameter syntax allows us to represent an indefinite number of arguments as an array. In the example, we use the rest parameters to collect arguments from the second element to the end. We then multiply them by the first one.
- Converts list to an array
- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters
- Tag: array.map ,

```
function multiply(multiplier, ...theArgs) {
    let anotherArray = theArgs.map(function (eachArgument) {
        /**
        *  takes each element from the first parameter array and
        * multiplies (function statements in general) the element with the multiplier
        * mentioned in the first parameter and stores the result in a new array.
        * that is what .map does on any array.
        */
        return eachArgument * multiplier
    })
    return anotherArray
}

var arr = multiply(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

==========

Arrow functions

- Refer:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Arrow_functions
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

>> Shorter functions

- Two factors influenced the introduction of arrow functions: shorter functions and non-binding of this.
- This doesn't work for function declarations. Works only with function expressions.

```
let x = (name) => {
    console.log(`haha! Hey ${name}!`);
}

x('Vin')    // haha! Hey Vin!
```

- remove **function** keyword and remove the **return** keyword **IF IT IS A SINGLE LINE FUNCTION** and remove the **flower brackets.**

```
let x = () => 54

let y = x()
console.log(y); // 54
```

```
let x = () => console.log(`okay`);

let y = x()
console.log(y);
/* not required as function
    is not returning anything */
```

```
var a = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];

var a2 = a.map(function(s) { return s.length; });

console.log(a2); // logs [8, 6, 7, 9]

var a3 = a.map(s => s.length);

console.log(a3); // logs [8, 6, 7, 9]
```

>> No separate this

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#No_separate_this

Predefined functions

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Predefined_functions

=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x

5. Expressions and operators

Operators

>> Assignment operators

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Assignment_operators

-->> 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.

```
var a, b, rest;
[a, b] = [10, 20];

console.log(a);
// expected output: 10

console.log(b);
// expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);
// expected output: [30,40,50]
```

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment
- Destructuring for objects,

```
var obj = {
    name: `vin`,
    age: 22,
    role: `dev`,
    isActive: true,
}

//  // VARIABLES DECLARED SEPARETELY
var name, age, remain;
({ name, age, ...remain } = obj)
//  //round brackets are needed when declaring the variables separately..
console.log(age);   // 22

//  // VARIABLES DECLARED ALONGSIDE
let { name, age, ...rem } = obj
console.log(rem);   // { role: 'dev', isActive: true }
```

- Destructuring for array,

```
let arr = [
    `vinay`, 22, `dev`, `active`
]

let [one, two, ...remainingstuff] = arr
console.log(`${two} `); // 22
```

>> Comparison operators

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Comparison_operators
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators

>> Arithmetic operators

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Arithmetic_operators

>> Bitwise operators

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Bitwise_operators

>> Logical operators

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Logical_operators

>> String operators

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#String_operators

>> Conditional (ternary) operator

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Conditional_(ternary)_operator

>> Comma operator

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Comma_operator

>> Unary operators

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Unary_operators

-->> delete

- The delete operator deletes an object, an object's property, or an element at a specified index in an array.

```
delete objectName;
delete objectName.property;
delete objectName[index];
```

- You can use the delete operator to delete variables declared implicitly but not those declared with the var statement.
- If the delete operator succeeds, it sets the property or element to undefined. The delete operator returns true if the operation is possible; it returns false if the operation is not possible.

```
x = 42;
var y = 43;
myobj = new Number();
myobj.h = 4;    // create property h
console.log(delete x)       // returns true (can delete if declared implicitly)
delete y;       // returns false (cannot delete if declared with var)
delete Math.PI; // returns false (cannot delete predefined properties)
delete myobj.h; // returns true (can delete user-defined properties)
delete myobj;   // returns true (can delete if declared implicitly)
```

- Deleting array elements:
    - When you delete an array element, the array length is not affected. For example, if you delete a[3], a[4] is still a[4] and a[3] is undefined.
    - When the delete operator removes an array element, that element is no longer in the array. In the following example, trees[3] is removed with delete. However, trees[3] is still addressable and returns undefined.

```
var trees = ['redwood', 'bay', 'cedar', 'oak', 'maple'];
delete trees[3];
if (3 in trees) {
  // this does not get executed
}
```

    - If you want an array element to exist but have an undefined value, use the undefined keyword instead of the delete operator. In the following example, trees[3] is assigned the value undefined, but the array element still exists.

```
var trees = ['redwood', 'bay', 'cedar', 'oak', 'maple'];
trees[3] = undefined;
if (3 in trees) {
  // this gets executed
}
```

-->> typeof

```
typeof operand
typeof (operand)
```

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof
- Also read under https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Unary_operators

-->> void

- read under https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Unary_operators
- The void operator specifies an expression to be evaluated without returning a value.

>> Relational operators

- A relational operator compares its operands and returns a Boolean value based on whether the comparison is true.

-->> in operator

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in
- The in operator returns true if the specified property is in the specified object.

```
propNameOrNumber in objectName
```

```
// Arrays
var trees = ['redwood', 'bay', 'cedar', 'oak', 'maple'];
0 in trees;        // returns true
3 in trees;        // returns true
6 in trees;        // returns false
'bay' in trees;    // returns false (you must specify the index number,
                   // not the value at that index)
'length' in trees; // returns true (length is an Array property)

// built-in objects
'PI' in Math;          // returns true
var myString = new String('coral');
'length' in myString;  // returns true

// Custom objects
var mycar = { make: 'Honda', model: 'Accord', year: 1998 };
'make' in mycar;  // returns true
'model' in mycar; // returns true
```

-->> instanceof

- The instanceof operator returns true if the specified object is of the specified object type.

```
objectName instanceof objectType
```

    - where objectName is the name of the object to compare to objectType, and objectType is an object type, such as Date or Array.
- Use instanceof when you need to confirm the type of an object at runtime. For example, when catching exceptions, you can branch to different exception-handling code depending on the type of exception thrown.

```
var theDay = new Date(1995, 12, 17);
if (theDay instanceof Date) {
  // statements to execute
}
```

>> Operator precedence

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Operator_precedence

==========

Expressions

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Expressions

>> Primary expressions

-->> this

- Use the this keyword to refer to the current object. In general, this refers to the calling object in a method. Use this either with the dot or the bracket notation.
- If an object has some properties and those properties are used later inside another function type property, then “this.propertyName” is used to refer to the previous properties of THAT object.
- Tags: this in objects, “this”, ‘this’,

```
let x = {
    name: 'vin',
    age: 22,
    role: 'dev',
    isActive: true,
    myFunc: function () {
        console.log(`Hey ${this.name}! You're ${age} years old..`);

    }
}
x.myFunc()  // without "this" for age - o/p: ReferenceError: age is not defined
```

```
let x = {
    name: 'vin',
    age: 22,
    role: 'dev',
    isActive: true,
    myFunc: function () {
        console.log(`Hey ${this.name}! You're ${this.age} years old..`);

    }
}
x.myFunc()  // with "this" - o/p: Hey vin! You're 22 years old..
```

```
/**
 * "this" is used in an object to reference to the object itself.
 * If there is a method in an object, you cannot directly access the
 * properties of that object inside the method.
 * "this.propertyName" has to be used to access that property..
 */
let x = {
    name: 'vin',
    age: 22,
    role: 'dev',
    isActive: true,
    myFunc: function () {
        console.log(`Hey ${this.name}! You're ${this.age} years old..`);
        /*if "this" is not used for 'name' or 'age',
        it gives a reference error - ReferenceError: age is not defined*/
    }
}
x.myFunc()  // with "this" - o/p: Hey vin! You're 22 years old..
```

```
this['propertyName']
this.propertyName
```

- 'this' on a html element

Suppose a function called `validate` validates an object's `value` property, given the object and the high and low values:

```
function validate(obj, lowval, hival) {
  if ((obj.value < lowval) || (obj.value > hival))
    console.log('Invalid Value!');
}
```

You could call `validate` in each form element's `onChange` event handler, using `this` to pass it the form element, as in the following example:

```
<p>Enter a number between 18 and 99:</p>
<input type="text" name="age" size=3 onChange="validate(this, 18, 99);">
```

-->> Grouping operator (*things here*)

- Used to override default operator precedence. * > / > + > - thingy

>> Left-hand-side expressions

-->> new

- You can use the new operator to create an instance of a user-defined object type or of one of the built-in object types.

```
var objectName = new objectType([param1, param2, ..., paramN]);
```

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}

var car1 = new Car('Eagle', 'Talon TSi', 1993);

console.log(car1.make);
// expected output: "Eagle"
```

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new

-->> super

- The super keyword is used to call functions on an object's parent. It is useful with classes to call the parent constructor.
- When used in a constructor, the super keyword appears alone and must be used before the this keyword is used. The super keyword can also be used to call functions on a parent object.

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super
- Example

```
class Rectangle {
  constructor(height, width) {
    this.name = 'Rectangle';
    this.height = height;
    this.width = width;
  }
  sayName() {
    console.log('Hi, I am a ', this.name + '.');
  }
  get area() {
    return this.height * this.width;
  }
  set area(value) {
    this.height = this.width = Math.sqrt(value);
  }
}

class Square extends Rectangle {
  constructor(length) {
    this.height; // ReferenceError, super needs to be called first!

    // Here, it calls the parent class' constructor with lengths
    // provided for the Rectangle's width and height
    super(length, length);

    // Note: In derived classes, super() must be called before you
    // can use 'this'. Leaving this out will cause a reference error.
    this.name = 'Square';
  }
}
```

-->> spread operator

- This is used to add extra set values to an array in an easier manner.
- The extra values to add to a final array is stored in a different array and that different array is passed as an element to the final array along with … (three dots). (somewhat like rest parameter.)
- Converts array to a list.
- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

```
let initialArray = [1, 2, 3, 4, 5,]

let extraToAdd = [1.2, 2.5, 6.8]

finalArray = [1, ...extraToAdd, 2, 3, 4, 5,]

console.log(finalArray);    // [ 1, 1.2, 2.5, 6.8, 2, 3, 4, 5 ]
```

=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x

6. Numbers and dates

Numbers

>> Decimal numbers

- Note that decimal literals can start with a zero (0) followed by another decimal digit, but if every digit after the leading 0 is smaller than 8, the number gets parsed as an octal number.

```
let x = 0777    // 511 (as octal)
let y = 0778    // 778 (as decimal)
console.log(x);
console.log(y);
```

>> Binary numbers

- Binary number syntax uses a leading zero followed by a lowercase or uppercase Latin letter "B" (0b or 0B). If the digits after the 0b are not 0 or 1, the following SyntaxError is thrown: "Missing binary digits after 0b.

```
//  // BINARY NUMBERS
let x = 0b1010
let y = 0B1111
console.log(x); // 10
console.log(y); // 15
```

>> Octal numbers

- Octal number syntax uses a leading zero. If the digits after the 0 are outside the range 0 through 7, the number will be interpreted as a decimal number.

```
let n1 = 0565
console.log(n1);     // 373
```

- Strict mode in ECMAScript 5 forbids octal syntax. Octal syntax isn't part of ECMAScript 5, but it's supported in all browsers by prefixing the octal number with a zero: 0644 === 420 and"\045" === "%". In ECMAScript 2015, octal numbers are supported if they are prefixed with 0o

```
var a = 0o10; // ES2015: 8
```

>>Hexadecimal numbers

- Hexadecimal number syntax uses a leading zero followed by a lowercase or uppercase Latin letter "X" (0x or 0X).
- If the digits after 0x are outside the range (0123456789ABCDEF), the following SyntaxError is thrown

```
let n1 = 0xa
let n2 = 0xD
console.log(n1);    // 10
console.log(n2);    // 13
```

>> Exponential

```
1E3   // 1000
2e6   // 2000000
0.1e2 // 10
```

==========

Number object

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#Number_object

```
var biggestNum = Number.MAX_VALUE;
var smallestNum = Number.MIN_VALUE;
var infiniteNum = Number.POSITIVE_INFINITY;
var negInfiniteNum = Number.NEGATIVE_INFINITY;
var notANum = Number.NaN;
```

Properties of `Number`

Property	Description
`Number.MAX_VALUE`	The largest representable number (`±1.7976931348623157e+308`)
`Number.MIN_VALUE`	The smallest representable number (`±5e-324`)
`Number.NaN`	Special "not a number" value
`Number.NEGATIVE_INFINITY`	Special negative infinite value; returned on overflow
`Number.POSITIVE_INFINITY`	Special positive infinite value; returned on overflow
`Number.EPSILON`	Difference between `1` and the smallest value greater than `1` that can be represented as a `Number` (`2.220446049250313e-16`)
`Number.MIN_SAFE_INTEGER`	Minimum safe integer in JavaScript ($-2^{53} + 1$, or `-9007199254740991`)
`Number.MAX_SAFE_INTEGER`	Maximum safe integer in JavaScript ($+2^{53} - 1$, or `+9007199254740991`)

Methods of `Number`

Method	Description
`Number.parseFloat()`	Parses a string argument and returns a floating point number. Same as the global `parseFloat()` function.
`Number.parseInt()`	Parses a string argument and returns an integer of the specified radix or base. Same as the global `parseInt()` function.
`Number.isFinite()`	Determines whether the passed value is a finite number.
`Number.isInteger()`	Determines whether the passed value is an integer.
`Number.isNaN()`	Determines whether the passed value is `NaN`. More robust version of the original global `isNaN()`.
`Number.isSafeInteger()`	Determines whether the provided value is a number that is a safe integer.

The `Number` prototype provides methods for retrieving information from `Number` objects in various formats. The following table summarizes the methods of `Number.prototype`.

Methods of `Number.prototype`

Method	Description
`toExponential()`	Returns a string representing the number in exponential notation.
`toFixed()`	Returns a string representing the number in fixed-point notation.
`toPrecision()`	Returns a string representing the number to a specified precision in fixed-point notation.

==========

Math object

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#Math_object
- **Note:** Unlike many other objects, you never create a Math object of your own. You always use the built-in Math object.

Methods of `Math`

Method	Description
`abs()`	Absolute value
`sin()`, `cos()`, `tan()`	Standard trigonometric functions; with the argument in radians.
`asin()`, `acos()`, `atan()`, `atan2()`	Inverse trigonometric functions; return values in radians.
`sinh()`, `cosh()`, `tanh()`	Hyperbolic functions; argument in hyperbolic angle.
`asinh()`, `acosh()`, `atanh()`	Inverse hyperbolic functions; return values in hyperbolic angle.
`pow()`, `exp()`, `expm1()`, `log10()`, `log1p()`, `log2()`	Exponential and logarithmic functions.
`floor()`, `ceil()`	Returns the largest/smallest integer less/greater than or equal to an argument.
`min()`, `max()`	Returns the minimum or maximum (respectively) value of a comma separated list of numbers as arguments.
`random()`	Returns a random number between 0 and 1.
`round()`, `fround()`, `trunc()`,	Rounding and truncation functions.
`sqrt()`, `cbrt()`, `hypot()`	Square root, cube root, Square root of the sum of square arguments.
`sign()`	The sign of a number, indicating whether the number is positive, negative or zero.
`clz32()`, `imul()`	Number of leading zero bits in the 32-bit binary representation. The result of the C-like 32-bit multiplication of the two arguments.

==========

Date object

- The Date object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties.
- JS store dates as the number of milliseconds since January 1, 1970, 00:00:00, with a Unix Timestamp being the number of seconds since January 1, 1970, 00:00:00.
- The Date object range is -100,000,000 days to 100,000,000 days relative to 01 January, 1970 UTC.
- Syntax,

```
var dateObjectName = new Date([parameters]);
```

- Calling Date without the new keyword returns a string representing the current date and time.

The `parameters` in the preceding syntax can be any of the following:

- Nothing: creates today's date and time. For example, `today = new Date();`.
- A string representing a date in the following form: "Month day, year hours:minutes:seconds." For example, `var Xmas95 = new Date("December 25, 1995 13:30:00")`. If you omit hours, minutes, or seconds, the value will be set to zero.
- A set of integer values for year, month, and day. For example, `var Xmas95 = new Date(1995, 11, 25)`.
- A set of integer values for year, month, day, hour, minute, and seconds. For example, `var Xmas95 = new Date(1995, 11, 25, 9, 30, 0);`.

- Example,

```
var dateIs = new Date("December 25, 1995 13:30:00").toLocaleString("en-US", { timeZone: "Asia/Calcutta" })
console.log(dateIs);    // 12/25/1995, 1:30:00 PM
```

>> Methods of the Date object

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#Methods_of_the_Date_object

=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x

7. Text formatting

Strings

>> String literals

-->> Hexadecimal escape sequences

- The number after \x is interpreted as a hexadecimal number.

```
'\xA9' // "©"
```

-->> Unicode escape sequences

- The Unicode escape sequences require at least four hexadecimal digits following \u.

```
'\u00A9' // "©"
```

-->> Unicode code point escapes

- Read under https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Text_formatting#String_literals

>> String objects

- The String object is a wrapper around the string primitive data type.
- You should use string literals unless you specifically need to use a String object, because String objects can have counterintuitive behavior.

```
const firstString = '2 + 2'; // Creates a string literal value
const secondString = new String('2 + 2'); // Creates a String object
eval(firstString); // Returns the number 4
eval(secondString); // Returns the string "2 + 2"
```

- A string object has a property length, which gives the number of characters in that string. Each character can be accessed using brackets (like in an array).

```
let t1 = 'okay'
console.log(t1[2]); // a
```

- But you can't change individual characters because strings are immutable array-like objects.

```
let t1 = 'okay'
console.log(t1);    // okay
t1[2] = 'u'
console.log(t1);    // okay - no change
```

Methods of `String`

Method	Description
`charAt`, `charCodeAt`, `codePointAt`	Return the character or character code at the specified position in string.
`indexOf`, `lastIndexOf`	Return the position of specified substring in the string or last position of specified substring, respectively.
`startsWith`, `endsWith`, `includes`	Returns whether or not the string starts, ends or contains a specified string.
`concat`	Combines the text of two strings and returns a new string.
`fromCharCode`, `fromCodePoint`	Constructs a string from the specified sequence of Unicode values. This is a method of the String class, not a String instance.
`split`	Splits a `String` object into an array of strings by separating the string into substrings.
`slice`	Extracts a section of a string and returns a new string.
`substring`, `substr`	Return the specified subset of the string, either by specifying the start and end indexes or the start index and a length.
`match`, `matchAll`, `replace`, `search`	Work with regular expressions.
`toLowerCase`, `toUpperCase`	Return the string in all lowercase or all uppercase, respectively.
`normalize`	Returns the Unicode Normalization Form of the calling string value.
`repeat`	Returns a string consisting of the elements of the object repeated the given times.
`trim`	Trims whitespace from the beginning and end of the string.

>> Multi-line template literals

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals
- These are strings which allow expressions to be inside them. Placeholder type thing.

Multi-lines

Any new line characters inserted in the source are part of the template literal. Using normal strings, you would have to use the following syntax in order to get multi-line strings:

```
console.log('string text line 1\n\
string text line 2');
// "string text line 1
// string text line 2"
```

To get the same effect with multi-line strings, you can now write:

```
console.log(`string text line 1
string text line 2`);
// "string text line 1
// string text line 2"
```

==

Internationalization

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Text_formatting#Internationalization

>> Date and time formatting

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DateTimeFormat

>> Number formatting

- The NumberFormat object is useful for formatting numbers, for example currencies.
- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NumberFormat
- tags: currency, international, convert

>> Collation

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Text_formatting#Collation

=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x

8. Indexed collections

Array object

>> Creating an array

```
var arr = new Array(element0, element1, ..., elementN);
var arr = Array(element0, element1, ..., elementN);
var arr = [element0, element1, ..., elementN];
```

To create an array with non-zero length, but without any items, either of the following can be used:

```
var arr = new Array(arrayLength);
var arr = Array(arrayLength);

// This has exactly the same effect
var arr = [];
arr.length = arrayLength;
```

Note : in the above code, `arrayLength` must be a `Number`. Otherwise, an array with a single element (the provided value) will be created. Calling `arr.length` will return `arrayLength`, but the array actually contains empty (undefined) elements. Running a `for...in` loop on the array will return none of the array's elements.

- Initializing an array with no elements but setting the length of the array
- Tags: setting array length, set initialize

If you wish to initialize an array with a single element, and the element happens to be a `Number`, you must use the bracket syntax. When a single `Number` value is passed to the Array() constructor or function, it is interpreted as an `arrayLength`, not as a single element.

```
var arr = [42];      // Creates an array with only one element:
                     // the number 42.

var arr = Array(42); // Creates an array with no elements
                     // and arr.length set to 42; this is
                     // equivalent to:
var arr = [];
arr.length = 42;
```

>> Populating an array

- **Note:** if you supply a non-integer value to the array operator in the code above, a property will be created in the object representing the array, instead of an array element.
- And the array.length property shows the length of only array elements.

```
let x = []
x[0] = 'haha'
x[1] = 'haha1'
x[2] = 'haha2'
x[3] = 'haha3'
x['four'] = 'haha4'
x[`five`] = 'haha5'
console.log(x);
//  //[ 'haha', 'haha1', 'haha2', 'haha3', four: 'haha4', five: 'haha5' ]
console.log(x.length);  // 4
```

```
var myArray = new Array('Hello', myVar, 3.14159);
var myArray = ['Mango', 'Apple', 'Orange'];
```

>> Referring to array elements

> **Note :** the array operator (square brackets) is also used for accessing the array's properties (arrays are also objects in JavaScript). For example,

```
var arr = ['one', 'two', 'three'];
arr[2];  // three
arr['length'];  // 3
```

>> Array length

You can also assign to the `length` property. Writing a value that is shorter than the number of stored items truncates the array; writing 0 empties it entirely:

```
var cats = ['Dusty', 'Misty', 'Twiggy'];
console.log(cats.length); // 3

cats.length = 2;
console.log(cats); // logs "Dusty, Misty" - Twiggy has been removed

cats.length = 0;
console.log(cats); // logs []; the cats array is empty

cats.length = 3;
console.log(cats); // logs [ <3 empty items> ]
```

>> Iterating over array

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#Iterating_over_arrays
- for

```
var colors = ['red', 'green', 'blue'];
for (var i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
```

- for each

```
var colors = ['red', 'green', 'blue'];
colors.forEach(function(color) {
  console.log(color);
});
// red
// green
// blue
```

```
var colors = ['red', 'green', 'blue'];
colors.forEach(color => console.log(color));
// red
// green
// blue
```

- In foreach the array item passed as the argument to the function. Unassigned values are not iterated in a forEach loop.
- **Note:** The elements of an array that are omitted when the array is defined are not listed when iterating by forEach, but are listed when undefined has been manually assigned to the element.

```
var array = ['first', 'second', , 'fourth'];

array.forEach(function(element) {
  console.log(element);
});
// first
// second
// fourth

if (array[2] === undefined) {
  console.log('array[2] is undefined'); // true
}

array = ['first', 'second', undefined, 'fourth'];

array.forEach(function(element) {
  console.log(element);
});
// first
// second
// undefined
// fourth
```

>> Array methods

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#Array_methods

>> Multi-dimensional array

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#Multi-dimensional_arrays

>> Arrays and regular expressions

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#Arrays_and_regular_expressions

>> Working with array-like objects

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#Working_with_array-like_objects

==

Typed Arrays

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#Typed_Arrays

x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x

9. Keyed collections

Maps

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map
- The Map object holds key-value pairs and remembers the original insertion order of the keys.

>> Example

- The value of a map can be an object, array, or any other variable.

```
let me1 = {
    name: 'vinay',
    age: 22,
}

let me2 = {
    name: 'Vin',
    age: 23,
}

let mees = new Map()
mees.set('wonie', me1)
mees.set('twoie', me2)
mees.set('role', 'dev')
mees.set('arr', [1, 2, 3, 4, false, true, null, undefined, 0, 'someString'])
mees.set('age', 23)
console.log(mees);
```

```
Map {
  'wonie' => { name: 'vinay', age: 22 },
  'twoie' => { name: 'Vin', age: 23 },
  'role' => 'dev',
  'arr' => [ 1, 2, 3, 4, false, true, null, undefined, 0, 'someString' ],
  'age' => 23 }
```

- **Any value** (both objects and primitive values) may be used as either a key or a value - which means even an array or an object can be set to act as a key. That array or object has to be set to a variable and that variable should be used as a key later on.

```
// array set to a variable, and this variable is used as a key in map.
let x = [9, 8, 7]
mees.set(x, 'array')

// object set to a variable, and this variable is used as a key in map.
let y = { runs: 22, "6's": 2, "4's": 0 }
mees.set(y, { name: 'vinaypn16', age: 23 })

console.log(mees.get(x));   // array
console.log(mees.get(y));   // { name: 'vinaypn16', age: 23 }
```

>> Ways to have a Map object

- using .set method

```
let mees = new Map()
mees.set('wonie', me1)
mees.set('twoie', me2)
mees.set('role', 'dev')
mees.set('arr', [1, 2, 3, 4, false, true, null, undefined, 0, 'someString'])
mees.set('age', 23)
```

- by passing key-value pairs in arrays and enclosing them all.

```
let mappie = new Map(
    [   // BIG BRACKET START - one big bracket enclosing all the arrays containing pairs..
        ['wonie', me1],
        ["twoie", me2],
        ['role', 'dev']
    ]   // BIG BRACKET END.
)

console.log(mappie.get("wonie"));
```

>> Basic operations with a Map

```
var sayings = new Map();
sayings.set('dog', 'woof');
sayings.set('cat', 'meow');
sayings.set('elephant', 'toot');
sayings.size; // 3
sayings.get('fox'); // undefined
sayings.has('bird'); // false
sayings.delete('dog');
sayings.has('dog'); // false

for (var [key, value] of sayings) {
  console.log(key + ' goes ' + value);
}
// "cat goes meow"
// "elephant goes toot"

sayings.clear();
sayings.size; // 0
```

>> Methods of Map

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map#Methods

>> Object and Map compared

Traditionally, objects have been used to map strings to values. Objects allow you to set keys to values, retrieve those values, delete keys, and detect whether something is stored at a key. `Map` objects, however, have a few more advantages that make them better maps.

- The keys of an `Object` are `Strings`, where they can be of any value for a `Map`.
- You can get the size of a `Map` easily while you have to manually keep track of size for an `Object`.
- The iteration of maps is in insertion order of the elements.
- An `Object` has a prototype, so there are default keys in the map. (this can be bypassed using `map = Object.create(null)`).

These three tips can help you to decide whether to use a `Map` or an `Object`:

- Use maps over objects when keys are unknown until run time, and when all keys are the same type and all values are the same type.
- Use maps if there is a need to store primitive values as keys because object treats each key as a string whether it's a number value, boolean value or any other primitive value.
- Use objects when there is logic that operates on individual elements.

>> WeakMap object

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Keyed_collections#WeakMap_object

==========

Sets

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

>> Set object

- Unique collection of values. Can iterate its elements in insertion order.
- **Note:** A value in a Set may only occur once; it is **UNIQUE** in the Set's collection.

```
let x = new Set()
x.add('haha')
x.add('okay')
x.add(34)
x.add(false)
console.log(x); // Set { 'haha', 'okay', 34, false }
x.add(34)   // won't be added
x.add(34)   // won't be added
console.log(x); // Set { 'haha', 'okay', 34, false }
x.add('haha')// won't be added
console.log(x.has(`haha`)); // true
```

>> Converting between Array and Set

```
let x = new Set()
x.add('haha')
x.add('okay')
x.add(34)
x.add(false)
console.log(x); // Set { 'haha', 'okay', 34, false }
x.add(34)   // won't be added
x.add(34)   // won't be added
console.log(x); // Set { 'haha', 'okay', 34, false }
x.add('haha')// won't be added
console.log(x.has(`haha`)); // true

//  // CONVERTING FROM SET TO ARRAY
let y = Array.from(x)
console.log(y); // [ 'haha', 'okay', 34, false ]
console.log(y.includes(false)); // true
```

>> Array and Set compared

- Checking whether an element exists in a collection using `indexOf` for arrays is slow.
- `Set` objects let you delete elements by their value. With an array you would have to splice based on an element's index.
- The value `NaN` cannot be found with `indexOf` in an array.
- `Set` objects store unique values; you don't have to keep track of duplicates by yourself.

>> WeakSet object

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Keyed_collections#WeakSet_object

>> Key and value equality of Map and Set

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Keyed_collections#Key_and_value_equality_of_Map_and_Set

x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x

10. Working with objects

Objects and properties

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Objects_and_properties
- Any property name that is not a valid JavaScript identifier (for example, a property name that has a space or a hyphen, or that starts with a number) can only be accessed using the square bracket notation.
- This notation is also very useful when property names are to be dynamically determined (when the property name is not determined until runtime).

```
let x = {
    one: true,
    name: 'vin',
    active: true,
    //  12prop_perty: 98, // this property will not be set from here.
}

x.age   // unassigned property, so won't be added to the object.
x['12prop_perty'] = 76  // setting that kind of property name will work with [] brackets..
console.log(x); // { one: true, name: 'vin', active: true, '12prop_perty': 76 }
```

>> Ways of giving property names

```
// four variables are created and assigned in a single go,
// separated by commas
var myObj = new Object(),
    str = 'myString',
    rand = Math.random(),
    obj = new Object();

myObj.type              = 'Dot syntax';
myObj['date created']   = 'String with space';
myObj[str]              = 'String value';
myObj[rand]             = 'Random Number';
myObj[obj]              = 'Object';
myObj['']               = 'Even an empty string';

console.log(myObj);
```

```
{ type: 'Dot syntax',
  'date created': 'String with space',
  myString: 'String value',
  '0.3355220328853863': 'Random Number',
  '[object Object]': 'Object',
  '': 'Even an empty string' }
```

>> Accessing properties by using a string value that is stored in a variable.

```
var propertyName = 'make';
myCar[propertyName] = 'Ford';

propertyName = 'model';
myCar[propertyName] = 'Mustang';
```

==========

Enumerate the properties of an object

- Ways to list object properties,

- `for...in` loops
  This method traverses all enumerable properties of an object and its prototype chain
- `Object.keys(o)`
  This method returns an array with all the own (not in the prototype chain) enumerable properties' names ("keys") of an object `o`.
- `Object.getOwnPropertyNames(o)`
  This method returns an array containing all own properties' names (enumerable or not) of an object `o`.

```
let x = {
    one: 'okayw',
    name: 'vin',
    active: true,
    //  12prop_perty: 98, // this property will not be set from here.
}

x.age   // unassigned property, so won't be added to the object.
x['12prop_perty'] = 76  // setting that kind of property name will work with [] brackets..
// console.log(x); // { one: true, name: 'vin', active: true, '12prop_perty': 76 }

console.log(Object.keys(x));    // [ 'one', 'name', 'active', '12prop_perty' ]
console.log(Object.getOwnPropertyNames(x)); //[ 'one', 'name', 'active', '12prop_perty' ]
```

- Way to show hidden properties of an object,

```
function listAllProperties(o) {
  var objectToInspect;
  var result = [];

  for(objectToInspect = o; objectToInspect !== null;
      objectToInspect = Object.getPrototypeOf(objectToInspect)) {
    result = result.concat(
      Object.getOwnPropertyNames(objectToInspect)
    );
  }

  return result;
        }
```

==========

Creating new objects

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Creating_new_objects
- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer
- Tags: ways to create an object, create object,

>> Using a constructor function

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Using_a_constructor_function
- First: create a function **(constructor function)** that takes in the required values for the properties of the object.

```
function emp(name, age, role, status) {
    this.nameIs = name
    this.ageIs = age
    this.roleIs = role
    this.statusIs = status
    // properties are nameIs, ageIs and all those
}
```

- Second: create an object using 'new' keyword followed by the function name.
- me is the new object.

```
let me = new emp('vinay', 22, 'dev', 'active')
me.okay = true  // adding additional properties later on.
console.log(me);
console.log(Object.keys(me));
//[ 'nameIs', 'ageIs', 'roleIs', 'statusIs', 'okay' ]
```

- Object inside an object (object as a property)

```
function addressOfEmp(fLine, sLine, city, postalCode) {
    this.firstLine = fLine
    this.secondLine = sLine
    this.cityIs = city
    this.pinCode = postalCode
}

function emp(name, age, role, status, address) {
    this.nameIs = name
    this.ageIs = age
    this.roleIs = role
    this.statusIs = status
    this.addressIs = address
    // properties are nameIs, ageIs and all those
}

let myAdd = new addressOfEmp('#88, 23rd main, 2nd cross', 'KNagar', 'Bangalore', 578964)

let me = new emp('vinay', 22, 'dev', 'active', myAdd)
me.okay = true  // adding additional properties later on.
console.log(me);
console.log(Object.keys(me));
//[ 'nameIs', 'ageIs', 'roleIs', 'statusIs', 'addressIs', 'okay' ]
```

```
emp {
  nameIs: 'vinay',
  ageIs: 22,
  roleIs: 'dev',
  statusIs: 'active',
  addressIs:
   addressOfEmp {
     firstLine: '#88, 23rd main, 2nd cross',
     secondLine: 'KNagar',
     cityIs: 'Bangalore',
     pinCode: 578964 },
  okay: true }
[ 'nameIs', 'ageIs', 'roleIs', 'statusIs', 'addressIs', 'okay' ]
```

>> Using the Object.create method(way)

- This method can be very useful, because it allows you to choose the prototype object for the object you want to create, without having to define a constructor function.

```
let fav = {
    color: 'red',
    display: function () {
        console.log(this.color);
    }
}
//  //me with default favourite color
let me = Object.create(fav)
me.display()    // red

//  // nonMe with another favourite color
let nonMe = Object.create(fav)
nonMe.color = 'grey'
nonMe.display()     //grey
```

>> Ways to write a function inside an object

```
let obj = {
    name: 'vin',
    age: 22,
    propertyName: function () {
        console.log(`This is function 1 - uses a property name
        and the "function" keyword to define a function.
        To call this function, the property name "obj.myFunc1()" is used.`);
    },
    functionName() {
        console.log(`This is function 2 - uses a function name and that's it.
        To call this function, the name of the function is used
        "obj.myFunc2()"`);
    }
}

let me1 = Object.create(obj)
me1.propertyName()
me1.functionName()
```

>> Issue with object and prototype things.

```
let obj = {
    name: 'vin',
    age: 22,
    propertyName: function () {
        console.log(`This is function 1 - uses a property name
        and the "function" keyword to define a function.
        To call this function, the property name "obj.myFunc1()" is used.`);
    },
    functionName() {
        console.log(`This is function 2 - uses a function name and that's it.
        To call this function, the name of the function is used
        "obj.myFunc2()"`);
    }
}

let me1 = Object.create(obj)
me1.propertyName()
me1.functionName()

/**
 * the properties initially will be set in the prototype of the object.
 * So it has to be manually called like - me1.age, me1.name, etc.,
 * logging just me1 won't display any values.
 * But once you change value of any property,
 * that property will be available in the current instance hence only
 * those changed properties will showup when just "me1" is logged.
 */

console.log(me1);   //  {} - initially nothing is shown.
me1.name = 'Vinay'  // changing values here.
me1.age = 23
console.log(me1);
// { name: 'Vinay', age: 23 } - the changed values reflecting in the current instance(me1)
```

==========

Inheritance

- All objects in JavaScript inherit from at least one other object. The object being inherited from is known as the prototype, and the inherited properties can be found in the prototype object of the constructor.
- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

==========

Indexing object properties

- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Indexing_object_properties
- If the second <FORM> tag in a document has a NAME attribute of "myForm", you can refer to the form as document.forms[1] or document.forms["myForm"] or document.forms.myForm.

==========

Defining properties for an object type

- You can add a property to a previously defined object type by using the prototype property. This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object.
- Read more: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Defining_properties_for_an_object_type

==========

Defining methods

- A function inside an object is called a **method.**
- Methods are defined the way normal functions are defined, except that they have to be assigned as the property of an object.
- Refer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions
- Read: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Defining_methods

==========

Using “this” for object references

- **this** is used to refer to the current object / calling object.

When combined with the `form` property, `this` can refer to the current object's parent form. In the following example, the form `myForm` contains a `Text` object and a button. When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onclick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<form name="myForm">
<p><label>Form name:<input type="text" name="text1" value="Beluga"></la
<p><input name="button1" type="button" value="Show Form Name"
     onclick="this.form.text1.value = this.form.name">
</p>
</form>
```

==========

Defining getters and setters

- Read more: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Defining_getters_and_setters
- Gets and/or sets the values to properties of an object or how ever required.
- Get followed by a function performs that function whenever that getter is called.
- **NOTE:** If a getter is used, that function can be called as a normal property itself. without braces () like - "me.fAge" instead of "me.fAge()"
- **Set** followed by a function does the things mentioned in the function, taking the value that is passed when the setter is called. (here “me.newAge” = 23 is passed and so it takes 23 and does things based on that).

```
let me = {
    pAge: 22,
    get fAge() {
        return this.pAge + 2000
    },
    set newAge(newAge) {
        this.pAge = newAge
    }
}

let u = me.pAge
console.log(`current age is ${u}`);
//current age is 22
console.log(me.fAge)
/**
 * if a getter is used, that function can be called
 * as a normal property itself.
 * without braces () like - "me.fAge" instead of "me.fAge()"
 */
me.newAge = 23
console.log(`new age is ${me.pAge}`);
//new age is 23
```

==========

Deleting properties

- **Note:** cannot delete the object itself - JS garbage collector will take care of it.
- delete operator is used to remove user defined properties from an object.

```
let obj = {
    name: 'vin',
    age: 22,
    role: 'dev',
    status: 'active',
}

console.log(obj);
//{ name: 'vin', age: 22, role: 'dev', status: 'active' }
delete obj.status
console.log(obj);
//{ name: 'vin', age: 22, role: 'dev' }
```

- It also deletes variables if no keywords(var, let, const) are used when declaring the variable.

```
x = 89
if (x) {
    console.log(`yes`);
}
delete x    // deletes variable if no keywords(var, let, const) are used when declaring the variable..
if (typeof (x) !== 'undefined') {
    console.log(x);
}
else {
    console.log(`variable deleted using delete operator`);
}
```

==

Comparing Objects

- **Note:** In JavaScript, objects are a reference type. Two distinct objects are never equal, even if they have the same properties. Only comparing the same object reference with itself yields true.

```
let obj1 = {
    name: 'vinay',
    age: 22,
}

let obj2 = {
    name: 'vinay',
    age: 22,
}

let x = obj1    //
console.log(obj1 == x);
/* true .. x is referring to the same object obj1. x is not a new object..
therfore this gives 'true'*/
console.log(obj1 == obj2);  // false
console.log(obj1 === obj2);  // false

x.age = 97
console.log(obj1.age);
// 97.. changing the referring variable x, will also change in the original object obj1
```

x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x=x

11. Details of the object model