JS_Beginning_to_Mastery_Part1_1

April 21, 2023

[ ]:
```
We can create variables using both `let` and `var` keywords.
```

In modern JavaScript, it's generally recommended to use `let` instead of `var` to declare variables.

The main reason for this is that `let` is block-scoped, whereas `var` is function-scoped. This means that a variable declared with `let` is only accessible within the block it was declared in (including any nested blocks), while a variable declared with `var` is accessible within the entire function it was declared in.

For example:

[13]:
```
function example() {
  var x = 1;
  if (true) {
    var x = 2; // same variable as above
    console.log(x); // 2
  }
  console.log(x); // 2
}

function example2() {
  let y = 1;
  if (true) {
    let y = 2; // different variable than above
    console.log(y); // 2
  }
  console.log(y); // 1
}
```

[14]:
```
example()
```

```
2
2
```

[15]:
```
example2()
```

```
2
1
```

In the first example, the variable `x` is overwritten within the nested block, and this change is reflected outside of the block as well. In the second example, the variable `y` is only accessible

within the block it was declared in, and a separate variable with the same name could be declared outside of the block without interfering with it.

In general, using `let` can help prevent accidental variable reassignment and make code easier to reason about, especially when dealing with more complex functions and control flow. However, there may be some cases where using var is still appropriate, such as when you intentionally want to create a variable that's accessible throughout an entire function

[1]:
```
console.log("Hello from file1.js");
console.log("Hello again");
```

```
Hello from file1.js
Hello again
```

ctrl + forward slash for comment

[5]:
```
// ctrl + forward slash for comment
// console.log();

let age = 20;
let firstName = "John";
```

```
evalmachine.<anonymous>:1
// ctrl + forward slash for comment
^

SyntaxError: Identifier 'age' has already been declared
    at Script.runInThisContext (node:vm:129:12)
    at Object.runInThisContext (node:vm:313:38)
    at run ([eval]:1020:15)
    at onRunRequest ([eval]:864:18)
    at onMessage ([eval]:828:13)
    at process.emit (node:events:513:28)
    at emit (node:internal/child_process:937:14)
    at process.processTicksAndRejections (node:internal/process/task_queues:83:
 ↪21)
```

[79]:
```
console.log(typeof(age)); // same as the below
console.log(typeof age);
```

```
number
number
```

0.0.1 convert number to string

[4]:
```
// convert number to string
age = String(age);
console.log(typeof age);
```

```
string
```

Convert number to string – again

[80]:
```
age = age.toString();
console.log(typeof age);
```

```
string
```

[81]:
```
console.log(age)
```

```
20
```

[82]:
```
// convert number to string -- again
age = age + "";
console.log(typeof age);
```

```
string
```

Convert string to number

[6]:
```
// convert string to number
age = Number(age);
console.log(typeof age);
```

```
number
```

[7]:
```
// convert string to number -- again
age = +age;
console.log(typeof age);
```

```
number
```

[8]:
```
// convert string to number -- again
age = parseInt(age);
console.log(typeof age);
```

```
number
```

String concatenation

[9]:
```
let string1 = "Shammu";
let string2 = "Khan";

let fullName = string1 + " " + string2;
console.log(fullName);
```

```
Shammu Khan
```

[10]:
```
// string concatenation -- again
let string3 = "17";
let string4 = "20";
```

```
let addedString = string3 + string4;
console.log(addedString);
```

```
1720
```

[11]:
```
// converts string to number and then adds
let addedNumber = +string3 + +string4; // +string3 converts string to number
console.log(addedNumber);
```

```
37
```

Template string

[12]:
```
// template string
let age2 = 22;
var firstName2 = "John";

console.log(`My name is ${firstName2} and I am ${age2} years old.`);
```

```
My name is John and I am 22 years old.
```

[2]:
```
// template string -- again
let aboutMe = `My name is ${firstName2} and I am ${age2} years old` ;
```

[3]:
```
console.log(aboutMe);
```

```
My name is John and I am 22 years old
```

[16]:
```
let myVariable = null;
console.log(myVariable); // prints null
myVariable = "ss";
console.log(myVariable, typeof myVariable); // ss string
console.log(typeof null); // this is bug or error as it shows object
```

```
null
ss string
object
```

BigInt

[17]:
```
// BigInt
let bigInt = 12345678901234567890123456789012345678901234567890n;
```

[18]:
```
console.log(bigInt, typeof bigInt);
```

```
12345678901234567890123456789012345678901234567890n bigint
```

[19]:
```
// BigInt -- again
let number = BigInt(123456);
console.log(number, typeof number);
```

```
123456n bigint
```

[20]:
```
console.log(Number.MAX_SAFE_INTEGER);
console.log(Number.MIN_SAFE_INTEGER);
```

```
9007199254740991
-9007199254740991
```

[21]:
```
// BigInt -- again
let number2 = 123n;
console.log(number2, typeof number2); // 123n bigint

console.log(number + number2); // 123579n
```

```
123n bigint
123579n
```

0.0.2 Booleans and comparison operators

[22]:
```
let a = 7;
let b = 8;
```

[23]:
```
console.log(a >= b);
```

```
false
```

== vs === == checks only value and === checks value and type

[24]:
```
// == vs ===
// == checks only value and === checks value and type
console.log(a == 2); // this is false as it checks only value
console.log(a === 2); // this is false as it checks value and type
console.log(a == "7"); // this is true as it checks only value
console.log(a === "7"); // this is false as it checks value and type
```

```
false
false
true
false
```

!= vs !== != checks only value and !== checks value and type

[25]:
```
console.log(a != 2);
console.log(a !== 2);
console.log(a != "7"); // this is false as it checks only value
console.log(a !== "7"); // this is true as it checks value and type
```

```
true
true
false
true
```

0.0.3 if conditional statement

[26]:
```
let age3 = 8;
let drink = age3 >= 18 ? "coffee" : "juice";
console.log(drink);
```

```
juice
```

[27]:
```
let firstName3 = "John";
let weight = 62;
let height = 1.8;

if(firstName3[0] === "J" && weight > 50 && height > 1.8){
    console.log("You are eligible to play");
} else {
    console.log("You are not eligible to play");
}
```

```
You are not eligible to play
```

[28]:
```
let weight2 = 62;
let height2 = 1.8;

let message = firstName3[0] === "J" || weight2 > 50 || height2 > 1.8 ? "You are␣
 ↪eligible to play" : "You are not eligible to play";
console.log(message); // You are eligible to play
```

```
You are eligible to play
```

0.0.4 switch statement

[29]:
```
let weight3 = 62;
let height3 = 1.8;

switch(firstName3[0]){
    case "J":
        console.log("You are eligible to play");
        break;
    case "A":
        console.log("You are eligible to play");
        break;
    case "B":
        console.log("You are eligible to play");
        break;
    default:
        console.log("You are not eligible to play");
}
```

```
You are eligible to play
```

[30]:
```
// groups multiple cases together
switch(firstName3[0]){
    case "J":
    case "A":
    case "B":
        console.log("You are eligible to play");
        break;
    default:
        console.log("You are not eligible to play");
}
```

```
You are eligible to play
```

[19]:
```
let winningNumber = 7;
//let userGuess = +prompt("Guess a number between 1 and 10"); // +prompt␣
 ↪converts string to number
let userGuess = 7;
```

[20]:
```
if(userGuess === winningNumber) {
  alert("You win!");
} else if(userGuess > winningNumber) {
  alert("Too high, try again!");
} else {
  alert("Too low, try again!");
}
```

```
evalmachine.<anonymous>:2
  alert("You win!");
  ^

ReferenceError: alert is not defined
    at evalmachine.<anonymous>:2:3
    at Script.runInThisContext (node:vm:129:12)
    at Object.runInThisContext (node:vm:313:38)
    at run ([eval]:1020:15)
    at onRunRequest ([eval]:864:18)
    at onMessage ([eval]:828:13)
    at process.emit (node:events:513:28)
    at emit (node:internal/child_process:937:14)
    at process.processTicksAndRejections (node:internal/process/task_queues:83:
 ↪21)
```

[51]:
```
let userGuess = 7;
```

```
evalmachine.<anonymous>:1
let userGuess = 7;
^
```

```
SyntaxError: Identifier 'userGuess' has already been declared
    at Script.runInThisContext (node:vm:129:12)
    at Object.runInThisContext (node:vm:313:38)
    at run ([eval]:1020:15)
    at onRunRequest ([eval]:864:18)
    at onMessage ([eval]:828:13)
    at process.emit (node:events:513:28)
    at emit (node:internal/child_process:937:14)
    at process.processTicksAndRejections (node:internal/process/task_queues:83:
↪21)
```

0.0.5 for loop

[31]:
```
for(let i=0; i <10; i++){
    console.log(i);
}
```

```
0
1
2
3
4
5
6
7
8
9
```

Sum of numbers between 0 and 10.

[32]:
```
let total = 0;
let num = 10;

for(let i=0; i <= num; i++){
    total += i;
}

console.log(total); // sum of numbers between 0 and 10
```

```
55
```

while loop

[33]:
```
// while loop

let i = 0;
while(i <= 10){
    console.log(i);
```

```
    i++;
}
```

```
0
1
2
3
4
5
6
7
8
9
10
```

[33]: 10

do while loop

[24]:
```
let j = 0;
do{
    console.log(j);
    j++
}while(j <=10);
```

```
0
1
2
3
4
5
6
7
8
9
10
```

[24]: 10

0.0.6 arrays

[34]:
```
// arrays
let fruits = ["apple", "banana", "orange", "mango", "grapes"];
let numbers = [10, 34, 56, 78, 90, 100];
let mixed = ["apple", 10, "banana", 34, "orange", 56, "mango", 78, "grapes",␣
 ↪90, 100];
```

[35]:
```
console.log(fruits); // ["apple", "banana", "orange", "mango", "grapes"]
fruits[1] = "pineapple";
```

```
console.log(fruits); // ["apple", "pineapple", "orange", "mango", "grapes"]
```

```
[ 'apple', 'banana', 'orange', 'mango', 'grapes' ]
[ 'apple', 'pineapple', 'orange', 'mango', 'grapes' ]
```

[36]:
```
console.log(numbers); // [10, 34, 56, 78, 90, 100]
numbers[1] = 100;
console.log(numbers); // [10, 100, 56, 78, 90, 100]
```

```
[ 10, 34, 56, 78, 90, 100 ]
[ 10, 100, 56, 78, 90, 100 ]
```

[37]:
```
console.log(mixed); // ["apple", 10, "banana", 34, "orange", 56, "mango", 78,␣
 ↪"grapes", 90, 100]
mixed[1] = 100;
console.log(mixed); // ["apple", 100, "banana", 34, "orange", 56, "mango", 78,␣
 ↪"grapes", 90, 100]
```

```
[
  'apple',  10,
  'banana', 34,
  'orange', 56,
  'mango',  78,
  'grapes', 90,
  100
]
[
  'apple',  100,
  'banana', 34,
  'orange', 56,
  'mango',  78,
  'grapes', 90,
  100
]
```

arrays are reference types and reference types are objects

[29]:
```
console.log(typeof fruits); // object
```

```
object
```

In JavaScript, an object literal is a way of creating a new object by specifying its properties and values in a concise syntax.

Here's an example of an object literal:

[38]:
```
const person = {
    name: 'John',
    age: 30,
    gender: 'male',
    occupation: 'developer'
```

```
};
```

In this example, we're using an object literal to create a new object called person. The object has four properties: name, age, gender, and occupation. The values of these properties are specified after the property name, separated by a colon. Each property-value pair is separated by a comma.

[39]:
```
console.log(person); // prints the object
```

```
{ name: 'John', age: 30, gender: 'male', occupation: 'developer' }
```

Object literals can also contain methods (i.e., functions that are properties of the object). Here's an example:

[40]:
```
const calculator = {
    add: function(x, y) {
      return x + y;
    },
    subtract: function(x, y) {
      return x - y;
    }
};
```

In this example, we're creating a new object called calculator with two methods: add and subtract. The methods are defined using function expressions and are assigned as properties of the object.

[41]:
```
// using dot notation to call the add and subtract methods
const sum = calculator.add(4, 9);
const difference = calculator.subtract(10, 4);
console.log(`The sum of 4 and 9 is ${sum}`);
console.log(`The difference between 10 and 4 is ${difference}`);
```

```
The sum of 4 and 9 is 13
The difference between 10 and 4 is 6
```

[42]:
```
let obj = {}; // object literal
console.log(typeof obj); // object
console.log(`type of obj is Array, true or false -- ${Array.isArray(obj)}`); //␣
 ↪true
```

```
object
type of obj is Array, true or false -- false
```

0.0.7 array methods

push() – adds element at the end

[43]:
```
// array methods
// push()
fruits.push("strawberry"); // adds element at the end
console.log(`push() adds strawberry at the end -- ${fruits}`);
```

```
push() adds strawberry at the end --
apple,pineapple,orange,mango,grapes,strawberry
```

pop() – removes last element

[44]:
```
// pop
fruits.pop(); // removes last element
console.log(`pop() removes last element or strawberry -- ${fruits}`);
```

```
pop() removes last element or strawberry -- apple,pineapple,orange,mango,grapes
```

unshift() – adds element at the beginning

[45]:
```
// unshift
fruits.unshift("strawberry"); // adds element at the beginning
console.log(`unshift(strawberry) adds strawberry at the beginning --␣
 ↪${fruits}`);
```

```
unshift(strawberry) adds strawberry at the beginning --
strawberry,apple,pineapple,orange,mango,grapes
```

shift() – removes first element

[47]:
```
// shift
fruits.shift(); // removes first element
console.log(`shift() removes first element or removes strawberry -- ${fruits}`);
```

```
shift() removes first element or removes strawberry --
pineapple,orange,mango,grapes
```

splice() – removes elements from array

[48]:
```
// splice -- removes elements from array
fruits.splice(1, 2); // removes 2 elements from index 1
console.log(`splice(1, 2) removes 2 elements from index 1 and so removes␣
 ↪pineapple and orange -- ${fruits}`);
```

```
splice(1, 2) removes 2 elements from index 1 and so removes pineapple and orange
-- pineapple,grapes
```

splice() – adds elements to array

[49]:
```
// exact opposite of the previous one
fruits.splice(1, 0, "banana", "mango"); // adds 2 elements at index 1
console.log(`fruits.splice(1, 0, "banana", "mango") adds banana and mango at␣
 ↪index 1 -- ${fruits}`);
```

```
fruits.splice(1, 0, "banana", "mango") adds banana and mango at index 1 --
pineapple,banana,mango,grapes
```

concat()

[50]:
```
let vegetables = ["tomato", "potato", "brinjal"];
let all = fruits.concat(vegetables);
```

```
console.log(all);
```

```
[
  'pineapple',
  'banana',
  'mango',
  'grapes',
  'tomato',
  'potato',
  'brinjal'
]
```

slice() – returns elements from one index to another

[51]:
```
let sliced = all.slice(1, 4); // returns elements from index 1 to 3
console.log(`slice(1, 4) returns elements from index 1 to 3 -- ${sliced}`);
```

```
slice(1, 4) returns elements from index 1 to 3 -- banana,mango,grapes
```

reverse() – reverses the array

[52]:
```
// reverse
console.log(`reverse() reverses the array -- ${all.reverse()}`); // reverses␣
 ↪the array
```

```
reverse() reverses the array --
brinjal,potato,tomato,grapes,mango,banana,pineapple
```

sort() – sorts an array

[53]:
```
console.log(`sort() sorts the array -- ${all.sort()}`); // sorts the array
```

```
sort() sorts the array -- banana,brinjal,grapes,mango,pineapple,potato,tomato
```

join() – joins the elements of an array by a symbol provided in the function

[54]:
```
console.log(`join(' - ') joins the array elements by ' - ':   ${all.join(' -␣
 ↪')}`); // joins the array elements with -
```

```
join(' - ') joins the array elements by ' - ':   banana - brinjal - grapes -
mango - pineapple - potato - tomato
```

indexOf() – returns index of the element

[55]:
```
console.log(`all array -- ${all}`);
console.log(`indexOf() gives the index of "banana" in all -- ${all.
 ↪indexOf("banana")}`); // returns index of the element
```

```
all array -- banana,brinjal,grapes,mango,pineapple,potato,tomato
indexOf() gives the index of "banana" in all -- 0
```

lastIndexOf() – returns the index of the last occurrence of the specified value

[56]:
```
console.log(`lastIndexOf() gives the index of the last occurrence of "banana"␣
↪${all.lastIndexOf("banana")}`); // returns index of the last occurrence of␣
↪the element "banana"
```

```
lastIndexOf() gives the index of the last occurrence of "banana" 0
```

includes() – returns true if an element is present in the array

[57]:
```
console.log(all.includes("banana")); // returns true if element is present
```

```
true
```

0.0.8 find method

returns the first element of an array that satisfies some condition.

[58]:
```
let numbers2 = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100];
let found = numbers2.find(function(value){
    return value > 50;
});
console.log(found); // returns first element greater than 50
```

```
60
```

0.0.9 findIndex()

returns the index of the first element that satisfies some condition.

[59]:
```
let found2 = numbers2.findIndex(function(value){
    return value > 50;
});
console.log(found2); // returns index of first element greater than 50
```

```
5
```

0.0.10 filter()

returns all the elements that satisfies some condition.

[60]:
```
let filtered = numbers2.filter(function(value){
    return value > 50;
});
console.log(filtered); // returns all elements greater than 50
```

```
[ 60, 70, 80, 90, 100 ]
```

0.0.11 forEach()

Gives both value and index which allows looping through an array

[61]:
```
numbers2.forEach(function(value, index){
    console.log(value, index);
```

```
});
```

```
10 0
20 1
30 2
40 3
50 4
60 5
70 6
80 7
90 8
100 9
```

[62]:
```
let array5 = ["item1", "item2"];
let array6 = array5;

console.log(array5);
console.log(array6);
```

```
[ 'item1', 'item2' ]
[ 'item1', 'item2' ]
```

[63]:
```
array5.push("item3");
console.log(`After pushing element to array5, array5 is ${array5}`);
```

```
After pushing element to array5, array5 is item1,item2,item3
```

[65]:
```
console.log(`Now, array6 is also changed to: ${array6}`)
```

```
Now, array6 is also changed to: item1,item2,item3
```

0.0.12 clone array

[1]:
```
let array10 = ["item1", "item2"];
```

Clone – First way

Use three dots ...

[2]:
```
console.log("First method using ...");
let array11 = [...array10];
```

```
First method using …
```

[3]:
```
console.log(array11);
```

```
[ 'item1', 'item2' ]
```

[4]:
```
array10.push("item3"); // this will have no effect on array11, a clone of␣
↪array10
console.log("After pushing element to array10");
```

```
// Now, array10 and array11 are different
console.log("Are array10 and array11 are the same", array10 === array11);
```

```
After pushing element to array10
Are array10 and array11 are the same false
```

[5]:
```
console.log(array11) // "item3" is not added to array11
```

```
[ 'item1', 'item2' ]
```

Clone – Second way

slice()

[6]:
```
console.log("Second method using slice");
let array12 = array10.slice();
console.log(array12);
```

```
Second method using slice
[ 'item1', 'item2', 'item3' ]
```

[7]:
```
array10.push("item4");
console.log("After pushing element to array10");
console.log("Are array10 and array12 are the same", array10 === array12);
```

```
After pushing element to array10
Are array10 and array12 are the same false
```

[8]:
```
console.log("array10",array10);
console.log("array12",array12);
console.log("");
```

```
array10 [ 'item1', 'item2', 'item3', 'item4' ]
array12 [ 'item1', 'item2', 'item3' ]
```

Clone – Third way

concat()

[9]:
```
console.log("Third method using concat");
let array13 = array10.concat();
```

```
Third method using concat
```

[11]:
```
array10.push("item5");
console.log(`After pushing element to array10, array10 is ${array10}`);
console.log("Are array10 and array13 are the same", array10 === array13);
```

```
After pushing element to array10, array10 is item1,item2,item3,item4,item5,item5
Are array10 and array13 are the same false
```

[14]:
```
console.log("array10",array10);
console.log("array13",array13);
```

```
array10 [ 'item1', 'item2', 'item3', 'item4', 'item5' ]
array13 [ 'item1', 'item2', 'item3', 'item4' ]
```

0.0.13 for loop

[15]:
```
let fruits3 = ["apple", "orange", "banana"];

let fruits4 = [];
for(let i=0; i<fruits3.length; i++) {
    fruits4.push(fruits3[i].toUpperCase());
}
console.log(fruits4);
```

```
[ 'APPLE', 'ORANGE', 'BANANA' ]
```

The value of `const` object can't be changed once the value is assigned.

[16]:
```
const pi=3.14;
// the below line will throw error
// pi= 12;
console.log(pi);
```

```
3.14
```

We can push to constant array. This will be stored in heap memory when doing push, we are not changing address and so we can push.

[17]:
```
const fruits5 = ["apple", "orange", "banana"];
fruits5.push("banana");
// the below line will throw error
// fruits5 = ["banana", "mango"]
```

[17]:
```
4
```

[18]:
```
console.log(fruits5)
```

```
[ 'apple', 'orange', 'banana', 'banana' ]
```

We can change the value of constant object. This will be stored in heap memory when doing push, we are not changing address and so we can push.

[19]:
```
const person10 = {
    name: "John",
    age: 30
};
person10.name = "Peter";
person10.age = 40;
// the below line will throw error
```

```
/*
person10 = {
     name: "Peter",
     age: 40
}
*/
```

[19]: 40

[20]:
```
console.log(person10)
```

```
{ name: 'Peter', age: 40 }
```

0.0.14 More Looping

let x of array

[21]:
```
const fruits10 = ["apple", "orange", "banana"];

// print all the fruits separately
for(let fruit of fruits10) {
    console.log(fruit);
}
```

```
apple
orange
banana
```

while loop

[22]:
```
// prints all the fruits in upper case
let k=0;
while(k<fruits10.length) {
    console.log(fruits10[k].toUpperCase());
    k++;
}
```

```
APPLE
ORANGE
BANANA
```

[22]: 2

0.0.15 array destructuring

[23]:
```
// saving array values to variables
const myArray = ["value1", "value2"];
let myVar1 = myArray[0];
let myVar2 = myArray[1];
console.log("value of myVar1", myVar1);
console.log("value of myVar2", myVar2);
```

```
value of myVar1 value1
value of myVar2 value2
```

[24]:
```
// array destructuring -- alternative to the previous steps
const myArray2 = ["value1", "value2"];
// the below line assigns the first value of the array to var1 and the second␣
 ↪value to var2
let [var1, var2] = myArray2; // this is called array destructuring
```

The above line assigns the first value of the array to `var1` and the second value to `var2`.

[3]:
```
var1 = "new value"; // we can do this as we are using let
console.log("value of var1", var1);
console.log("value of var2", var2);
```

```
value of var1 new value
value of var2 value2
```

But if we array destructure using const, we will not be able to change the values of the variables.

[25]:
```
const [constVar1, constVar2] = myArray2;
// constVar1 = "new value"; // this will throw an error
```

array destructuring – assigning values to less number of variables

[26]:
```
const myArray3 = ["value1", "value2", "value3"];
let [var3, var4] = myArray3;
console.log("value of var3", var3);
console.log("value of var4", var4);
```

```
value of var3 value1
value of var4 value2
```

array destructuring – assigning values to more number of variables

[27]:
```
const myArray4 = ["value1", "value2"];
let [var5, var6, var7] = myArray4;
console.log("value of var5", var5);
console.log("value of var6", var6);
console.log("value of var7", var7); // this will be undefined
```

```
value of var5 value1
value of var6 value2
value of var7 undefined
```

array destructuring – skipping values

[28]:
```
// skipping values
const myArray5 = ["value1", "value2", "value3"];
let [var8, , var9] = myArray5;
console.log("value of var8", var8);
console.log("value of var9", var9);
```

```
value of var8 value1
value of var9 value3
```

array destructuring – using rest operator ..

rest operator ..

[29]:
```
const myArray6 = ["value1", "value2", "value3", "value4"];
```

The below line assigns the first value of the array to var10 and the rest of the values to var11

[30]:
```
// the below line assigns the first value of the array to var10
// and the rest of the values to var11
let [var10, ...var11] = myArray6; // this is called rest operator
console.log("value of var10", var10);
console.log("value of var11", var11);
```

```
value of var10 value1
value of var11 [ 'value2', 'value3', 'value4' ]
```

[31]:
```
let myNewArray2 = myArray6.slice(2); // this assigns values from the third␣
 ↪index to the end of the array to myNewArray
console.log("value of myNewArray", myNewArray2);
```

```
value of myNewArray [ 'value3', 'value4' ]
```

array destructuring – more Can't do the following as rest operator should be the last element

[32]:
```
// let [..., var12] = myArray6; // can't do this as rest operator should be the␣
 ↪last element
// console.log(var12);
```

0.0.16 Objects

In JavaScript, objects are one of the fundamental data types, used to store and manipulate collections of data in a structured way.

An object in JavaScript is a container for properties, which are essentially key-value pairs. The keys are always strings, while the values can be any data type, including other objects, functions, arrays, and primitive types such as numbers and strings.

Objects can be created using object literals, which use curly braces {} to enclose the properties and their values:

[33]:
```
const person = {
  name: "John",
  age: 30,
  hobbies: ["reading", "playing guitar"],
  address: {
    street: "123 Main St",
    city: "Anytown",
```

```
    state: "CA"
  }
};
```

In this example, `person` is an object with four properties: `name`, `age`, `hobbies`, and `address`. The address property is itself an object with its own properties.

You can access the properties of an object using dot notation or square bracket notation:

[34]:
```
console.log(person.name);       // "John"
console.log(person["age"]);   // 30
console.log(person.hobbies[0]);   // "reading"
console.log(person.address.state); // "CA"
```

```
John
30
reading
CA
```

Objects in JavaScript are dynamic, which means you can add or remove properties at any time:

[35]:
```
person.job = "programmer";  // add a new property
delete person.hobbies;     // remove the hobbies property
```

[35]:
```
true
```

[36]:
```
console.log(person)
```

```
{
  name: 'John',
  age: 30,
  address: { street: '123 Main St', city: 'Anytown', state: 'CA' },
  job: 'programmer'
}
```

Objects can also have methods, which are functions stored as properties:

[37]:
```
const calculator = {
  add: function(a, b) {
    return a + b;
  },
  subtract: function(a, b) {
    return a - b;
  }
};

console.log(calculator.add(2, 3));       // 5
console.log(calculator.subtract(5, 2));  // 3
```

```
5
3
```

In this example, calculator is an object with two methods, add and subtract

[39]:
```
const person2 = {"name": "John", "age": 30, "city": "New York"};
```

0.0.17 Accessing object values using dot notation

[40]:
```
// accessing object values using dot notation
console.log(person.name);
console.log(person.age);
console.log(person.city);
```

```
John
30
undefined
```

[41]:
```
console.log(typeof person);
```

```
object
```

0.0.18 Accessing object values using bracket notation

[42]:
```
// accessing object values using bracket notation
console.log(person["name"]);
console.log(person["city"]);
```

```
John
undefined
```

0.0.19 We can have array as values of an object

[45]:
```
// we can have array as values of an object
const person4 = {"name": "John", "age": 30, "city": "New York", "hobbies":␣
↪["music", "movies", "sports"]};
console.log(person4.hobbies[1]);
```

```
movies
```

0.0.20 How to add key value pairs to an object

[47]:
```
// how to add key value pairs to an object
person4["email"] = "sha_is13@gmail.com";
console.log(person4);
```

```
{
  name: 'John',
  age: 30,
  city: 'New York',
  hobbies: [ 'music', 'movies', 'sports' ],
  email: 'sha_is13@gmail.com'
}
```

[48]:
```
console.log(person4.email);
console.log(person4["email"]);
```

```
sha_is13@gmail.com
sha_is13@gmail.com
```

0.0.21 Create new property

[49]:
```
const person6 = {
    "name": "John",
    "age": 30,
    "city": "New York",
    "hobbies": ["music", "movies", "sports"]
};

person6["gender"] = "male";
console.log(person6.gender);
```

```
male
```

0.0.22 Difference between dot and bracket notation

- dot notation is used when we know the key
- bracket notation is used when we don't know the key
- bracket notation is used when we have a variable as key
- bracket notation is used when we have a key with spaces

[50]:
```
const key = "email";
const person7 = {
    name: "John",
    age: 30,
    city: "New York",
    "person hobbies": ["music", "movies", "sports"]
};
```

[51]:
```
console.log(person7["person hobbies"]);
```

```
[ 'music', 'movies', 'sports' ]
```

Suppose we want the value of the variable key to be the key for person4 object. Now, using dot notation will not work.

[52]:
```
// Suppose we want the value of the variable key to be the key
// for person4 object. Now, using dot notation will not work

person4.key = "sha_is13@gmail.com";
console.log(person4);
```

```
{
  name: 'John',
```

```
  age: 30,
  city: 'New York',
  hobbies: [ 'music', 'movies', 'sports' ],
  email: 'sha_is13@gmail.com',
  key: 'sha_is13@gmail.com'
}
```

But we want the value of the variable key to be the key solution is using bracket notation

[53]:
```
// But we want the value of the variable key to be the key
// solution is using bracket notation
person4[key] = "sha_is13@gmaill.com";
console.log(person4);
```

```
{
  name: 'John',
  age: 30,
  city: 'New York',
  hobbies: [ 'music', 'movies', 'sports' ],
  email: 'sha_is13@gmaill.com',
  key: 'sha_is13@gmail.com'
}
```

0.0.23 Remove a key value pair from an object

[55]:
```
// remove a key value pair from an object
delete person4["key"];
console.log(person4);
```

```
{
  name: 'John',
  age: 30,
  city: 'New York',
  hobbies: [ 'music', 'movies', 'sports' ],
  email: 'sha_is13@gmaill.com'
}
```

0.0.24 Iterate objects

[57]:
```
const person8 = {
    name: "John",
    age: 30,
    city: "New York",
    hobbies: ["music", "movies", "sports"]
};
```

[62]:
```
// for in loop
for(let key in person8){
    console.log(key, person8[key]);
```

```
}
```

```
name John
age 30
city New York
hobbies [ 'music', 'movies', 'sports' ]
```

[63]:
```
// Object.keys() returns an array of keys
console.log(typeof (Object.keys(person8)));
```

```
object
```

[64]:
```
// for in loop -- another example
for(let key in person8){
    console.log(`${key}: ${person8[key]}`);
}
```

```
name: John
age: 30
city: New York
hobbies: music,movies,sports
```

[66]:
```
// for key of loop
for(let key of Object.keys(person8)){
    console.log(person8[key]);
}
```

```
John
30
New York
[ 'music', 'movies', 'sports' ]
```

0.0.25 Computed properties

[67]:
```
const key1 = "objkey1";
const key2 = "objkey2";

const value1 = "objvalue1";
const value2 = "objvalue2";

obj = {};
obj[key1] = value1;
obj[key2] = value2;
console.log(obj);
```

```
{ objkey1: 'objvalue1', objkey2: 'objvalue2' }
```

0.0.26 Spread operator ...

Spread operator is used to split up array elements or object properties

[69]:
```
const array1 = [1,2,3];
const array2 = [5,6,7];

const newArray = [...array1]; // copies array1 into newArray
console.log(newArray);
```

```
[ 1, 2, 3 ]
```

[70]:
```
const newArray2 = [...array1, ...array2]; // copies array1 and array2 into␣
 ↪newArray2
console.log(newArray2);
```

```
[ 1, 2, 3, 5, 6, 7 ]
```

[71]:
```
// spread operator -- continued
// for array2 we are not using spread operator
const newArray3 = [...array1, array2]; // copies values of array1 as elements␣
 ↪and array2 as array into newArray3
console.log(newArray3);
```

```
[ 1, 2, 3, [ 5, 6, 7 ] ]
```

[72]:
```
// spread operator -- continued
const newArray4 = [...array1, ...array2, 78, 56];
console.log(newArray4);
```

```
[
  1, 2,  3,  5,
  6, 7, 78, 56
]
```

[73]:
```
// spread string
const newArray5 = [..."Hello World"];
console.log(newArray5);
```

```
[
  'H', 'e', 'l', 'l',
  'o', ' ', 'W', 'o',
  'r', 'l', 'd'
]
```

Spread operator doesn't work for integers

[75]:
```
// spread operator in objects
const person0 = {
    key1: "value1",
    key2: "value2"
};
console.log(person0);
```

```
{ key1: 'value1', key2: 'value2' }
```

[77]:
```
// just to make things clear that same key can't
// be present more than one time in an object
const person22 = {
    key1: "value1",
    key2: "value2",
    key1: "LastkeyValue" // this will overwrite the previous key1
};
console.log(person22);
```

```
{ key1: 'LastkeyValue', key2: 'value2' }
```

[78]:
```
// spread operator in objects -- continued
const obj1 = {
    key1: "value1",
    key2: "value2"
};

const obj2 = {
    key3: "value3",
    key4: "value4"
};
```

[79]:
```
const newObject = { ...obj1}; // copies obj1 into newObject
console.log(newObject);
```

```
{ key1: 'value1', key2: 'value2' }
```

[80]:
```
// spread operator in objects -- continued
const newObject2 = { ...obj1, ...obj2}; // copies obj1 and obj2 into newObject2
console.log(newObject2);
```

```
{ key1: 'value1', key2: 'value2', key3: 'value3', key4: 'value4' }
```

suppose we are using spread operator to copy obj1 and obj2 and they both have a same key. In this case, the value of the key in obj2 will overwrite the value of the key in obj1.

[81]:
```
const obj3 = {
    key1: "value1",
    key2: "value2"
};

const obj4 = {
    key1: "value3",
    key4: "value4",
    key5: "value5"
};

const newObject3 = {...obj3, ...obj4}; // copies obj3 and obj4 into newObject3
console.log(newObject3);
```

```
{ key1: 'value3', key2: 'value2', key4: 'value4', key5: 'value5' }
```

[82]:
```
// spread operator in objects -- continued
// adding new key with spread operator
const newObject4 = {...obj3, ...obj4, key6: "value6"};
console.log(newObject4);
```

```
{
  key1: 'value3',
  key2: 'value2',
  key4: 'value4',
  key5: 'value5',
  key6: 'value6'
}
```

Spread string will result into index as keys and characters as values

[83]:
```
const newObject5 = {..."Hello World"};
console.log(newObject5);
```

```
{
  '0': 'H',
  '1': 'e',
  '2': 'l',
  '3': 'l',
  '4': 'o',
  '5': ' ',
  '6': 'W',
  '7': 'o',
  '8': 'r',
  '9': 'l',
  '10': 'd'
}
```

Spread array will result into index as keys and elements as values

[84]:
```
const newObjects = {...["item1", "item2"]};
console.log(newObjects);
```

```
{ '0': 'item1', '1': 'item2' }
```

[85]:
```
// spread string of characters
const newObjects2 = {..."abcdefghijklmnopqrstuvwxyz"};
console.log(newObjects2);
```

```
{
  '0': 'a',
  '1': 'b',
  '2': 'c',
  '3': 'd',
  '4': 'e',
```

```
    '5': 'f',
    '6': 'g',
    '7': 'h',
    '8': 'i',
    '9': 'j',
    '10': 'k',
    '11': 'l',
    '12': 'm',
    '13': 'n',
    '14': 'o',
    '15': 'p',
    '16': 'q',
    '17': 'r',
    '18': 's',
    '19': 't',
    '20': 'u',
    '21': 'v',
    '22': 'w',
    '23': 'x',
    '24': 'y',
    '25': 'z'
}
```

[ ]:

JS_Beginning_to_Mastery_Part1_2

April 21, 2023

0.0.1 Object destructuring

[1]:
```
const band = {
    bandName: "The Beatles",
    members: 4,
    genre: "Rock",
    famousSong: "Yesterday",
    famousAlbum: "Abbey Road"
};

let {bandName, members, ...restProps} = band;
console.log(bandName);
console.log(restProps);
```

```
The Beatles
{ genre: 'Rock', famousSong: 'Yesterday', famousAlbum: 'Abbey Road' }
```

0.0.2 Objects inside arrays

[2]:
```
const users = [
    {userId: 1, name: 'John', age: 25},
    {userId: 2, name: 'Mary', age: 30},
    {userId: 3, name: 'Peter', age: 28}
];
```

[3]:
```
for(let user of users){
    console.log(user.name);
}
```

```
John
Mary
Peter
```

0.0.3 Nested destructuring

[4]:
```
const users2 = [
    {"userId": 1, "name": "John", "age": 25, "address": {"city": "New York",␣
 ↪"state": "NY"}},
```

```
    {"userId": 2, "name": "Mary", "age": 30, "address": {"city": "Boston",␣
  ↪"state": "MA"}},
    {"userId": 3, "name": "Peter", "age": 28, "address": {"city": "Chicago",␣
  ↪"state": "IL"}}
];
```

[5]:
```
const [Myuser1, Myuser2, Myuser3] = users2;
console.log(Myuser2)
```

```
{
  userId: 2,
  name: 'Mary',
  age: 30,
  address: { city: 'Boston', state: 'MA' }
}
```

But suppose, we need only the name of user1 and age of user3, we can use nested destructuring

[6]:
```
const[{name}, , {age}] = users2;
```

[7]:
```
console.log(name)
```

```
John
```

[8]:
```
console.log(age);
```

```
28
```

We can also change the name of the variables

[9]:
```
const[{name: myName}, , {age:myAge}] = users2;
console.log(myName);
console.log(myAge);
```

```
John
28
```

Suppose, we need the userId and name of user1 and age of user 3

[10]:
```
const[{name: myName2, userId}, , {age: myAge2}] = users2;
console.log(myName2);
console.log(userId);
console.log(myAge2);
```

```
John
1
28
```

Returns the index at which the target is found if the target is not found, return -1

[11]:
```
function findTarget(array, target){
    for(let i = 0; i<array.length; i++){
        if(array[i]===target){
            return i;
        }
    }
    return -1;
}
const myArray = [1,3,8,90]
const ans = findTarget(myArray, 4);
```

[12]:
```
console.log(ans);
```

```
-1
```

[13]:
```
console.log(findTarget(myArray, 3));
```

```
1
```

0.0.4 Arrow function – =>

[14]:
```
// conventional function -- multiple arguments
const sumThreeNumbers = function(a, b, c){
    return a + b + c;
}
```

[15]:
```
// arrow function -- multiple arguments
const sumThreeNumbers2 = (a, b, c) => {
    return a + b + c;
};
```

[16]:
```
console.log(sumThreeNumbers2(1,2,3));
```

```
6
```

[17]:
```
// arrow function -- single argument
const even = num => num % 2 === 0;
console.log(even(5));
```

```
false
```

[18]:
```
// arrow function -- single argument -- continued
const square = (num) => {
    return num * num
};
console.log(square(5));
```

```
25
```

[19]:
```
// arrow function -- single argument -- continued
const square2 = num => num * num; // don't use return if it's a single line or␣
 ↪there is no parenthesis
console.log(square2(5));
```

25

[20]:
```
// arrow function
const firstChar = anyString => anyString[0];
console.log(firstChar("hello"));
```

h

[21]:
```
// arrow function -- no arguments
const sayHello = () => "hello";
console.log(sayHello());
```

hello

[24]:
```
// arrow function -- no arguments -- continued
const sayHello3 = () => {
    return "hello";
};
console.log(sayHello3());
```

hello

[25]:
```
// without arrow function
function findTarget(array, target){
    for(let i = 0; i<array.length; i++){
        if(array[i]===target){
            return i;
        }
    }
    return -1;
}
```

[26]:
```
// arrow function
const findTarget2 = (array, target) => {
    for(let i=0; i <array.length; i++){
        if(array[i]==target){
            return i;
        }
    }
    return -1;
};
```

[29]:
```
const myArray4 = [1,3,8,90];
const ans4 = findTarget(myArray4, 4);
```

```
console.log(ans4);
```

```
-1
```

[30]:
```
const ans5= findTarget2(myArray4, 4);
console.log(ans5);
```

```
-1
```

[31]:
```
const ans6= findTarget2(myArray4, 8);
console.log(ans6);
```

```
2
```

[32]:
```
const ans7= findTarget2(myArray4, 8);
console.log(ans7);
```

```
2
```

0.0.5 Hoisting

Hoisting for variables

0.0.6 var – variable can be accessed before declared

undefined will be returned if hoisted Hoisting is a JavaScript mechanism where variable and function declarations are moved to the top of their respective scope before code execution. This means that regardless of where variables and functions are declared in the code, they are interpreted as if they were declared at the beginning of their scope.

For example, consider the following code snippet:

[33]:
```
console.log(a);
var a = 10;
```

```
undefined
```

Even though a is declared after the `console.log()` statement, the code will still execute without throwing an error. This is because of hoisting. The code is interpreted by the JavaScript engine as follows:

```
var a;
console.log(a);
a = 10;
```

As you can see, the variable declaration `var a;` is moved to the top of the scope, so when the `console.log(a)` statement is executed, `a` has already been declared (but not yet assigned a value), so it does not throw an error.

It's important to note that only the declaration of the variable or function is hoisted, not the initialization or assignment. So, in the example above, `a` is declared at the top of its scope, but its value is not assigned until the second line of code.

Hoisting can be a helpful feature of JavaScript, but it can also lead to confusion and bugs if not used properly. It's best practice to declare all variables at the top of their respective scope to avoid unexpected behavior.

[40]:
```
console.log(y);

// variable declaration
var y = 5; // with var undefined will be returned when this variable is called␣
 ↪before this line
```

```
undefined
```

let – variable can't be accessed before declared With `let`, error will be thrown when this variable is called before this line

```
console.log(x); // will throw error

// variable declaration
let x = 5;
```

Hoisting for functions

Hoisting will work for function declaration Hoisting also applies to functions in JavaScript. Function declarations are hoisted to the top of their respective scope, which means that they can be called before they are defined in the code.

For example, consider the following code snippet:

[34]:
```
foo();

function foo() {
  console.log("Hello, world!");
}
```

```
Hello, world!
```

Even though the `foo()` function is called before its definition, the code will still execute without throwing an error. This is because of hoisting. The code is interpreted by the JavaScript engine as follows:

[35]:
```
function foo() {
  console.log("Hello, world!");
}

foo();
```

```
Hello, world!
```

As you can see, the function declaration `function foo() {...}` is moved to the top of the scope, so when the foo() function is called, it has already been declared.

[38]:
```
// hoisting
// will work for function declaration
sayHello4();

// function declaration
function sayHello4(){
    console.log("hello");
}
```

```
hello
```

It's important to note that only function declarations are hoisted, not function expressions. Function expressions are created when a function is assigned to a variable, and they are not hoisted. For example:

```
bar(); // Throws an error

var bar = function() {
  console.log("Hello, world!");
};
```

In this example, the `bar()` function is defined as a function expression and assigned to the variable bar. Since function expressions are not hoisted, the code throws an error when `bar()` is called before its definition.

It's best practice to declare all functions at the top of their respective scope, or to use function expressions and assign them before they are called. This can help avoid unexpected behavior and errors in your code.

Hoisting will not work for function expression

```
hello2(); // this will not work

const hello2 = function(){
    console.log("hello");
};
```

Hoisting will not work for arrow function

```
hello3(); // this will not work

const hello3 = () => {
    console.log("hello");
};
```

0.0.7 functions inside a function

[41]:
```
function app(){
    const myFunc = () => {
        console.log("Hello from myFunc");
    };
```

```
    const addTwoNumbers = (a, b) => {
        return a + b;
    };

    const mul = (num1, num2) => num1*num2;

    console.log("inside app");

    myFunc();
    console.log(addTwoNumbers(2, 3));
    console.log(mul(5, 6));
}
app();
```

```
inside app
Hello from myFunc
5
30
```

0.0.8 Lexical scoping

Lexical scoping is a concept in programming languages that describes how variable scope is determined at the time of writing code, rather than when it is executed. In JavaScript, lexical scoping means that a variable's scope is determined by its location in the code, or more specifically, by where it is declared.

When a function is defined in JavaScript, it creates a new scope for the variables declared inside it. This is known as the function's closure. The variables declared inside the function are available only within the function itself, and any nested functions or closures declared inside the function.

In other words, when a function is defined, it captures the values of any variables in its enclosing scope, and those values are available to the function when it is executed, even if the variables themselves are no longer in scope.

Here's an example:

[42]:
```
function outerFunction() {
  var outerVar = 'I am declared in outerFunction';

  function innerFunction() {
    console.log(outerVar);
  }

  innerFunction();
}

outerFunction(); // Output: "I am declared in outerFunction"
```

```
I am declared in outerFunction
```

In this example, `outerFunction` declares a variable called `outerVar`. `innerFunction` is defined inside `outerFunction`, which means it has access to `outerVar` because of lexical scoping. When `outerFunction` is executed, it calls `innerFunction`, which outputs the value of `outerVar` to the console.

Lexical scoping is an important concept in JavaScript, and it helps to avoid naming conflicts and unintended consequences that can occur when variables are not properly scoped.

[43]:
```
function myApp(){
    const myVar = "value1";
    function myFunc(){
        const myVar = "value 60";
        // but if we comment out the above line, then the value of myVar will␣
 ↪be "value1"
        console.log("inside myFunc", myVar);
    };
    const myFunc2 = function(){};
    const myFunc3 = () => {};
    console.log(myVar); // prints "value1"
    myFunc(); // prints "inside myFunc value 60"
}

myApp();
```

```
value1
inside myFunc value 60
```

[44]:
```
// lexical scoping
const myVar = "value 1";

function myApp2(){
    function myFunc(){
        // the value of myVar will be "value 1"
        console.log("inside myFunc", myVar);
    };
    const myFunc2 = function(){};
    const myFunc3 = () => {};
    console.log(myVar);
    myFunc();
}

myApp2();
```

```
value 1
inside myFunc value 1
```

[1]:
```
// lexical scoping
const myVar2 = "value 2";
```

```
function myApp3(){
    function myFunc(){
        const myFunc2 = () => {
            console.log("inside myFunc", myVar2); // the value of myVar will be␣
 ↪"value 2"
        };
        myFunc2();
    };
    console.log(myVar2);
    myFunc();
}

myApp3();
```

```
value 2
inside myFunc value 2
```

0.0.9 Block scope vs Function scope

`let` and `const` are block scoped, `var` is function scoped

In JavaScript, variables can be declared with either function scope or block scope.

Function scope means that variables declared inside a function are only accessible within that function and any nested functions or closures defined within it. Here's an example:

[1]:
```
function exampleFunction() {
  var a = 1;
  if (a === 1) {
    var b = 2;
    console.log(b); // Output: 2
  }
  console.log(b); // Output: 2
}

exampleFunction();
```

```
2
2
```

In this example, `a` is declared inside `exampleFunction` and is only accessible within that function. `b` is also declared inside `exampleFunction`, but since it uses `var`, it has function scope, which means it is accessible within the entire function, including the `if` block.

Block scope means that variables declared inside a block of code (i.e., within curly braces `{}`) are only accessible within that block and any nested blocks. Block scope is achieved using the `let` and `const` keywords, which were introduced in ES6. Here's an example:

[2]:
```
function exampleFunction() {
  let a = 1;
```

```
  if (a === 1) {
    const b = 2;
    console.log(b); // Output: 2
  }
  console.log(b); // Output: ReferenceError: b is not defined
}

exampleFunction();
```

```
2
```

```
evalmachine.<anonymous>:7
  console.log(b); // Output: ReferenceError: b is not defined
              ^


ReferenceError: b is not defined
    at exampleFunction (evalmachine.<anonymous>:7:15)
    at evalmachine.<anonymous>:10:1
    at Script.runInThisContext (node:vm:129:12)
    at Object.runInThisContext (node:vm:313:38)
    at run ([eval]:1020:15)
    at onRunRequest ([eval]:864:18)
    at onMessage ([eval]:828:13)
    at process.emit (node:events:513:28)
    at emit (node:internal/child_process:937:14)
    at process.processTicksAndRejections (node:internal/process/task_queues:83:
↪21)
```

In this example, `a` is declared using `let`, which means it has block scope and is only accessible within the block where it is defined. `b` is declared using `const`, which also has block scope, but cannot be reassigned once it is defined.

To summarize, function scope means that variables declared with `var` are accessible within the entire function, while block scope means that variables declared with `let` or `const` are only accessible within the block where they are defined.

[3]:
```
{
    let a = 10;
    console.log(a); // this will work
}
// console.log(a); // will throw error
```

```
10
```

[4]:
```
{
    let a = 20;
    console.log(a); // this will work
}
```

```
20
```

[5]:
```
{
    const a2 = 10;
    console.log(a2); // this will work
}
// console.log(a2); // will throw error
```

```
10
```

[6]:
```
{
    const a2 = 20;
    console.log(a2); // this will work
}
// console.log(a2); // will throw error
```

```
20
```

[7]:
```
const a2 = 100;
console.log(a2);
```

```
100
```

[8]:
```
// for var, we can access the variable outside the block
{
    var a3 = 200;
    console.log(a3); // this will work
}
console.log(a3); // this will work too
```

```
200
200
```

[9]:
```
// var can be accessed outside the block
// var can be accessed anywhere
{
    var firstName = "John";
    console.log(firstName);
}

{
    console.log(firstName); // it can access the previous block
}
```

```
John
John
```

0.0.10 Rest parameters

[10]:
```
function myFunc(a,b, ...c){
    console.log(`a is ${a}`);
    console.log(`b is ${b}`);
    console.log(`c is ${c}`);
}

myFunc(3,4,5,6,7,8,9,10);
```

```
a is 3
b is 4
c is 5,6,7,8,9,10
```

[11]:
```
function addAll(...numbers){
    console.log(numbers);
    console.log(Array.isArray(numbers));
    let total = 0;
    for(let number of numbers){
        total = total + number;
    }
    return total;
}

const ans = addAll(1,2,3,4,5,6,7,8,9,10);
console.log(ans);
```

```
[
  1, 2, 3, 4,  5,
  6, 7, 8, 9, 10
]
true
55
```

0.0.11 Parameter destructuring in JavaScript

Parameter destructuring is a feature in JavaScript that allows you to extract values from objects and arrays passed as function arguments and assign them to variables with the same name as the object's keys or array's elements. This can be done in the function signature using curly braces `{}` for objects or square brackets `[]` for arrays.

Here's an example of destructuring an object passed as a parameter:

[12]:
```
function exampleFunction({a, b}) {
  console.log(a, b);
}

const obj = {a: 1, b: 2};
exampleFunction(obj); // Output: 1 2
```

```
1 2
```

In this example, the `exampleFunction` takes an object as its parameter and destructures it using curly braces in the function signature. This creates new variables `a` and `b` and assigns them the values of the corresponding properties in the object passed as the argument.

Here's an example of destructuring an array passed as a parameter:

[13]:
```
function exampleFunction([a, b]) {
  console.log(a, b);
}

const arr = [1, 2];
exampleFunction(arr); // Output: 1 2
```

```
1 2
```

In this example, the `exampleFunction` takes an array as its parameter and destructures it using square brackets in the function signature. This creates new variables `a` and `b` and assigns them the values of the corresponding elements in the array passed as the argument.

Parameter destructuring can make your code more concise and readable, especially when dealing with complex objects or arrays passed as function arguments. However, it's important to use it judiciously and avoid excessive nesting or complexity.

[14]:
```
const person = {
    firstName: "John",
    gender: "male"
};
```

[15]:
```
// without param destructuring
function printDetails(obj){
    console.log(obj.firstName);
    console.log(obj.gender)
}

printDetails(person);
```

```
John
male
```

[16]:
```
// we can use destructuring to make the code more concise
function printDetails2({firstName, gender}){
    console.log(firstName);
    console.log(gender);
}

printDetails2(person);
```

```
John
male
```

0.0.12 Callback function in JavaScript

In JavaScript, a callback function is a function that is passed as an argument to another function and is called when the other function has finished executing. The purpose of a callback function is to allow asynchronous processing of data or events, without blocking the execution of other code.

Here's an example:

[ ]:
```
function fetchData(callback) {
  // Some asynchronous operation to fetch data
  const data = [1, 2, 3, 4, 5];
  callback(data);
}

function process(data) {
  console.log(data.map(x => x * 2));
}

fetchData(process); // Output: [2, 4, 6, 8, 10]
```

In this example, the `fetchData` function takes a callback function as its argument, which is called with the fetched data when it's available. The `process` function is defined separately and takes the data as its argument, processes it, and outputs the result to the console.

When `fetchData` is called, it initiates an asynchronous operation to fetch data and then calls the `callback` function with the fetched data when it's available. The `process` function is passed as the callback function, and it's executed with the fetched data as its argument.

Callbacks are widely used in JavaScript for a variety of purposes, including event handling, data fetching, and asynchronous programming. They allow for more flexible and modular code by decoupling the execution of functions from their definition.

[1]:
```
function myFunc(a){
    console.log(a);
    console.log("hello world!");
}
myFunc([1,2,3]);
myFunc("abc");
```

```
[ 1, 2, 3 ]
hello world!
abc
hello world!
```

[2]:
```
function myFunc2(){
    console.log("Inside myFunc2()");
}

function myFunc3(a){
    console.log(a);
```

```
}

myFunc3(myFunc2); // returns the function myFunc2() itself
```

```
[Function: myFunc2]
```

[3]:
```
// we can pass a function as a parameter to another function
function myFunc4(a){
    // console.log(a);
    console.log("Inside myFunc4()");
    a();
}

myFunc4(myFunc2); // callback function
```

```
Inside myFunc4()
Inside myFunc2()
```

[4]:
```
function myFunc5(callback){
    callback();
}

function myFuncTest(name){
    console.log("inside my func test");
    console.log(`Your name is ${name}`);
}

myFunc5(myFuncTest);
myFunc5(myFuncTest("John"));
```

```
inside my func test
Your name is undefined
inside my func test
Your name is John
```

```
evalmachine.<anonymous>:2
    callback();
    ^

TypeError: callback is not a function
    at myFunc5 (evalmachine.<anonymous>:2:5)
    at evalmachine.<anonymous>:11:1
    at Script.runInThisContext (node:vm:129:12)
    at Object.runInThisContext (node:vm:313:38)
    at run ([eval]:1020:15)
    at onRunRequest ([eval]:864:18)
    at onMessage ([eval]:828:13)
    at process.emit (node:events:513:28)
    at emit (node:internal/child_process:937:14)
```

```
    at process.processTicksAndRejections (node:internal/process/task_queues:83:
↪21)
```

0.0.13 function returning function

[5]:
```
function myFunc(){
    function hello(){
        console.log("Hello world!");
    }
    return hello;
}

const ans = myFunc();
ans()
```

```
Hello world!
```

[6]:
```
console.log(ans());
```

```
Hello world!
undefined
```

[7]:
```
function myFunc2(){
    return function(){
        return "Hello world!";
    };
}

const ans2 = myFunc2();
console.log(ans2()); // returns "Hello world!"
```

```
Hello world!
```

0.0.14 Important array methods

[2]:
```
const numbers = [4, 3, 5, 6, 9];

function multiplyBy2(number, index){
    //console.log(`index is ${index} number is ${number}`);
    console.log(`${number}*2 = ${number*2}`);
}

multiplyBy2(numbers[0], 0);
multiplyBy2(numbers[1], 1);
```

```
4*2 = 8
3*2 = 6
```

[3]:
```
for(let i=0; i < numbers.length; i++){
    //console.log(i);
    multiplyBy2(numbers[i], i);
}

// using forEach method -- forEach method takes a callback function as a␣
 ↪parameter
// similar to the previous one
numbers.forEach(multiplyBy2);
```

```
4*2 = 8
3*2 = 6
5*2 = 10
6*2 = 12
9*2 = 18
4*2 = 8
3*2 = 6
5*2 = 10
6*2 = 12
9*2 = 18
```

[4]:
```
// define callback function inside the forEach method
numbers.forEach(function(number, index){
    console.log(`index is ${index} number is ${number}`);
})
```

```
index is 0 number is 4
index is 1 number is 3
index is 2 number is 5
index is 3 number is 6
index is 4 number is 9
```

[6]:
```
// define callback function inside the forEach method -- another example
numbers.forEach(function(number, index){
    console.log(`${number}*10 = ${number*10}`);
})
```

```
4*10 = 40
3*10 = 30
5*10 = 50
6*10 = 60
9*10 = 90
```

[7]:
```
// forEach with array of objects
const users = [
    { name: "John", age: 30, city: "New York" },
    { name: "Jane", age: 25, city: "Boston" },
    { name: "Jack", age: 40, city: "Miami" }
];
```

```
users.forEach(function(user){
    console.log(user.name);
})
```

```
John
Jane
Jack
```

[8]:
```
// the same above functionality using for loop
for(let i=0; i < users.length; i++){
    console.log(users[i].name);
}
```

```
John
Jane
Jack
```

[9]:
```
// forEach using arrow function
// can have both user and index
users.forEach((user, index)=>{
    console.log(user.name, index);
});
```

```
John 0
Jane 1
Jack 2
```

[10]:
```
// can have only user and not index
users.forEach((user) => {
    console.log(user.name);
})
```

```
John
Jane
Jack
```

0.0.15 map

In JavaScript, `map()` is a built-in function that is used to transform an array by applying a function to each of its elements. The `map()` function creates a new array with the same number of elements as the original array, but with each element transformed according to the function passed as an argument.

`map()` is similar to `forEach()`.

[1]:
```
const num = [3,4,5,6,7,8,9,10];
```

[2]:
```
// using forEach
const square = function(number){
```

```
    console.log(number*2);
}
num.forEach(square)
```

```
6
8
10
12
14
16
18
20
```

[3]:
```
// using map
const square2 = function(number){
    return number*2;
}
console.log(num.map(square2));
```

```
[
   6,  8, 10, 12,
  14, 16, 18, 20
]
```

[4]:
```
const squareNumber = num.map(square2); // returns an array
console.log(squareNumber);
```

```
[
   6,  8, 10, 12,
  14, 16, 18, 20
]
```

[5]:
```
const squareNumber2 = num.map((number)=>{return number*2})
console.log(squareNumber2);
```

```
[
   6,  8, 10, 12,
  14, 16, 18, 20
]
```

[6]:
```
const squareNumber3 = num.map((number)=> number*2)
console.log(squareNumber3);
```

```
[
   6,  8, 10, 12,
  14, 16, 18, 20
]
```

[7]:
```
const users = [
    { name: "John", age: 30, city: "New York" },
    { name: "Jane", age: 25, city: "Boston" },
    { name: "Jack", age: 40, city: "Miami" },
    { name: "Jill", age: 35, city: "Los Angeles" },
    { name: "Joe", age: 20, city: "Chicago" }
]
```

[8]:
```
const userNames = users.map((user)=>{
    return user.name;
});
console.log(userNames);
```

```
[ 'John', 'Jane', 'Jack', 'Jill', 'Joe' ]
```

[10]:
```
const userNames3 = users.map((user)=> user.name)
console.log(userNames3);
```

```
[ 'John', 'Jane', 'Jack', 'Jill', 'Joe' ]
```

0.0.16 filter method

[12]:
```
const numbers6 = [1,2,3,4,5,6,7,8,9,10];

const evenNumber = numbers6.filter((number)=> number%2 == 0)
console.log(evenNumber);
```

```
[ 2, 4, 6, 8, 10 ]
```

We can also do the same without using arrow function.

[13]:
```
const isEven = function(number){
    return number%2 == 0;
}
console.log(numbers6.filter(isEven))
```

```
[ 2, 4, 6, 8, 10 ]
```

0.0.17 reduce method

The `reduce()` method is a built-in function in JavaScript that is used to reduce an array of values to a single value. It takes in a callback function as its first argument and an optional initial value as the second argument.

The callback function is executed on each element of the array and has access to two arguments: an accumulator and the current value. The accumulator is the value that is returned and passed to the next iteration of the callback function, and it is initialized to the initial value (if provided) or the first element of the array.

The callback function can perform any operation on the accumulator and the current value, such as addition, multiplication, or concatenation. At the end of the iteration, the final value of the

accumulator is returned.

Here's an example of using the `reduce()` method to find the sum of an array of numbers:

[14]:
```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, currentValue) => accumulator +␣
 ↪currentValue, 0);

console.log(sum); // Output: 15
```

```
15
```

In this example, the `reduce()` method starts with an initial value of 0 and then adds each element of the `numbers` array to the accumulator. At the end of the iteration, the final value of the accumulator (which is the sum of all the numbers) is returned.

The `reduce()` method can also be used to perform other operations, such as finding the maximum or minimum value in an array, concatenating strings, or even grouping objects based on a common property.

[20]:
```
var numbers12 = [1,2,3,4,5,10];

// sum of all the numbers in the array
var sum12 = numbers12.reduce((accumulator, currentValue) => {
    return accumulator + currentValue;
});

console.log(sum12);
```

```
25
```

When `reduce()` is called without an initial value, the first iteration of the callback function uses the first element of the array as the initial value of the accumulator, and the second element as the current value. In subsequent iterations, the accumulator will be the result of the previous iteration, and the current value will be the next element in the array.

In the example above, the first iteration sets `accumulator` to 1 (the first element in the array), and `currentValue` to 2 (the second element in the array). The callback function then adds these two values together and returns the result (3), which becomes the new value of the accumulator for the next iteration.

This process continues for each element in the array until all elements have been processed, and the final value of the accumulator is returned as the result of the `reduce()` method.

So, even though an initial value is not provided, the `reduce()` method still works and returns the correct sum because it uses the first element of the array as the initial value of the accumulator. However, it's generally recommended to provide an initial value to avoid any unexpected behavior in cases where the array is empty.

[21]:
```
const userCart = [
    { name: "laptop", price: 1000, quantity: 1 },
    { name: "desktop", price: 2000, quantity: 2 },
```

```
    { name: "mobile", price: 500, quantity: 4 },
    { name: "tablet", price: 300, quantity: 3 }
];

const totalAmount = userCart.reduce((totalPrice, currentProduct) => {
    return totalPrice + currentProduct.price;
}, 0);

console.log(totalAmount);
```

3800

0.0.18 Sort

[22]:
```
const numbers13 = [5,9,1200,410,3000];
```

[24]:
```
const userNames13 = ['John', 'Jane', 'Jack', 'Jill', 'Joe'];
userNames13.sort();
console.log(userNames13);
```

```
[ 'Jack', 'Jane', 'Jill', 'Joe', 'John' ]
```

[25]:
```
// ascending order
numbers13.sort((a,b)=> a-b)
console.log(numbers13);
```

```
[ 5, 9, 410, 1200, 3000 ]
```

[26]:
```
// descending order
numbers13.sort((a,b)=> b-a);
console.log(numbers13);
```

```
[ 3000, 1200, 410, 9, 5 ]
```

[27]:
```
// sorting array of objects
const products = [
    {productId: 1, productName: "p1", price: 300},
    {productId: 2, productName: "p2", price: 3300},
    {productId: 3, productName: "p3", price: 1200},
    {productId: 4, productName: "p4", price: 300},
    {productId: 5, productName: "p5", price: 1500}
];

// low to high
products.sort((a, b)=>{
    return a.price - b.price;
});
console.log(products);
```

```
[
  { productId: 1, productName: 'p1', price: 300 },
  { productId: 4, productName: 'p4', price: 300 },
  { productId: 3, productName: 'p3', price: 1200 },
  { productId: 5, productName: 'p5', price: 1500 },
  { productId: 2, productName: 'p2', price: 3300 }
]
```

[28]:
```
// high to low
products.sort((a, b)=>{
    return b.price - a.price;
});
console.log(products);
```

```
[
  { productId: 2, productName: 'p2', price: 3300 },
  { productId: 5, productName: 'p5', price: 1500 },
  { productId: 3, productName: 'p3', price: 1200 },
  { productId: 1, productName: 'p1', price: 300 },
  { productId: 4, productName: 'p4', price: 300 }
]
```

Create a new sorted object

[29]:
```
// But this is changing products and we don't to do that. In
// order to avoid that, we can use slice method

// low to high
const lowToHigh = products.slice(0).sort((a, b) => {
    return a.price - b.price;
});
console.log(lowToHigh);
```

```
[
  { productId: 1, productName: 'p1', price: 300 },
  { productId: 4, productName: 'p4', price: 300 },
  { productId: 3, productName: 'p3', price: 1200 },
  { productId: 5, productName: 'p5', price: 1500 },
  { productId: 2, productName: 'p2', price: 3300 }
]
```

[30]:
```
// high to low
const highToLow = products.slice(0).sort((a, b) => {
    return b.price - a.price;
});
console.log(highToLow);
```

```
[
  { productId: 2, productName: 'p2', price: 3300 },
  { productId: 5, productName: 'p5', price: 1500 },
```

```
  { productId: 3, productName: 'p3', price: 1200 },
  { productId: 1, productName: 'p1', price: 300 },
  { productId: 4, productName: 'p4', price: 300 }
]
```

0.0.19 find method

The `find()` method is a built-in method in JavaScript that is used to search for the first element in an array that satisfies a specified condition. The method returns the value of the first element that passes the test implemented by the provided function.

The syntax of the `find()` method is as follows:

```
array.find(function(currentValue, index, array) {
  // code to test each element of the array
});
```

Here, `array` is the array that the method is called upon, and `currentValue`, `index`, and `array` are optional parameters that represent the current value being processed, the index of the current value, and the array being processed, respectively.

The `find()` method executes the provided function once for each element of the array until it finds an element that satisfies the provided condition. If a matching element is found, the method immediately returns its value, and if no matching element is found, the method returns `undefined`.

[31]:
```
const myArray = ["Hello", "World", "I", "am", "a", "string", "array", "how",␣
 ↪"are"];

function isLength3(string){
    return string.length === 3;
}

const ans = isLength3("dog");
console.log(ans);
```

```
true
```

[32]:
```
const ans2 = myArray.find(isLength3);
console.log(ans2); // returns the first string with length 3, here "how"
```

```
how
```

[33]:
```
const ans3 = myArray.find((string) => string.length === 3);
console.log(ans3);
```

```
how
```

[35]:
```
const users10 = [
    {userId: 1, userName: "John"},
    {userId: 2, userName: "Jane"},
    {userId: 3, userName: "Jack"},
```

```
    {userId: 4, userName: "Jill"},
    {userId: 5, userName: "Joe"},
    {userId: 6, userName: "Jenny"}
];

const myUser10 = users10.find((user)=>user.userId === 3);
console.log(myUser10);
```

```
{ userId: 3, userName: 'Jack' }
```

0.0.20 every method

If every element satisfies a condition, returns True

[39]:
```
const numbers14 = [2, 4, 6, 8, 10];

const ans14 = numbers14.every((number)=>number%2 === 0);
console.log(ans14);
```

```
true
```

[41]:
```
function isEven16(number){
    return number % 2 === 0;
}

const ans16 = numbers14.every(isEven);
console.log(ans16);
```

```
true
```

[43]:
```
// every method -- continued
const userCart2 = [
    {productId: 1, quantity: 2, productName: 'laptop', price: 100000},
    {productId: 2, quantity: 1, productName: 'mobile', price: 8000},
    {productId: 3, quantity: 3, productName: 'tablet', price: 30000},
    {productId: 4, quantity: 4, productName: 'watch', price: 2000}
];

const ans17 = userCart2.every((item)=>item.price<200000); // every item price␣
 ↪is less than 200000
console.log(ans17);
```

```
true
```

0.0.21 some method

If some elements satisfy a condition, returns True

[44]:
```

```

```
// some method -- returns true or false
const ans18 = userCart2.some((item)=>item.price>80000); // some item price is␣
 ↪greater than 80000
console.log(ans18);
```

```
true
```

0.0.22 fill method

Fills an array with a static value from a start index to an end index

[46]:
```
const myArray11 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
myArray11.fill(0, 2, 5); // fill with 0 from index 2 to index 4
console.log(myArray11);
```

```
[
  1, 2, 0, 0,  0,
  6, 7, 8, 9, 10
]
```

[47]:
```
const myArray12 = new Array(10).fill(0); // fill with 0 from index 0 to index 9
console.log(myArray12);
```

```
[
  0, 0, 0, 0, 0,
  0, 0, 0, 0, 0
]
```

[48]:
```
const obj3 = {
    key1: "value1",
    key2: "value2"
};

const obj4 = {
    key1: "value3",
    key4: "value4",
    key5: "value5"
};

const newObject3 = {...obj3, ...obj4}; // copies obj3 and obj4 into newObject3
console.log(newObject3);
```

```
{ key1: 'value3', key2: 'value2', key4: 'value4', key5: 'value5' }
```

0.0.23 Maps

Map is a collection of key value pairs; duplicate keys are not allowed; store data in ordered fashion.

In Map, we can use any type as key.

Object literal // the below is called an object literal

object literal

- key -> string
- value -> any type

[49]:
```
const person = {
    firstName: 'John',
    age: 7,
    1: "one" // 1 will be considered as string
};

console.log(person.firstName);
console.log(person["firstName"]);
```

```
John
John
```

[50]:
```
for(let key in person){
    console.log(typeof key);
}
console.log(person[1]);
```

```
string
string
string
one
```

```
console.log(person.1); // error
```

[51]:
```
// key value pair
// for Map, we can keep any type as key
const person2 = new Map();
person2.set('firstName', 'Munna');
person2.set('age', 33);
person2.set(1, 'one');
person2.set([1,2,3], 'onetwothree');
person2.set({1: 'one'}, 'onetwothree'); // object literal as key

console.log(person2);
```

```
Map(5) {
  'firstName' => 'Munna',
  'age' => 33,
  1 => 'one',
  [ 1, 2, 3 ] => 'onetwothree',
  { '1': 'one' } => 'onetwothree'
}
```

[52]:
```
// Accessing values of Map
console.log(person2.get('age'));
console.log(person2.get(1));
console.log(person2.keys());
```

```
33
one
[Map Iterator] { 'firstName', 'age', 1, [ 1, 2, 3 ], { '1': 'one' } }
```

[53]:
```
// any type of data can be used as key

// we can iterate over keys of the Map

for(let key of person2.keys()){
    console.log(key, typeof key);
}
```

```
firstName string
age string
1 number
[ 1, 2, 3 ] object
{ '1': 'one' } object
```

In object, we couldn't use for of loop, we could use for in loop though but in Map, we can directly use for of loop

[54]:
```
// in object, we couldn't use for of loop, we could use for in loop though
// but in Map, // we can directly use for of loop

const person3 = new Map();
person3.set('firstName', 'Munna');
person3.set('age', 33);
person3.set(1, 'one');

for(let key of person3){
    console.log(Array.isArray(key));
}
```

```
true
true
true
```

Destructuring

[55]:
```
// destructuring an array:
let arr = [1, 2, 3, 4, 5];
let [a, b, c] = arr;
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

```
1
2
3
```

[56]:
```
// destructuring an object:
let obj = { x: 1, y: 2, z: 3 };
let { x, y, z } = obj;
console.log(x); // 1
console.log(y); // 2
console.log(z); // 3
```

```
1
2
3
```

Destructuring with map

[57]:
```
// destructuring with Map

for(let [key, value] of person3){
    console.log(key, value);
}
```

```
firstName Munna
age 33
1 one
```

[58]:
```
// another way to create map
const person4 = new Map([
    ['firstName', 'Munna'],
    ['age', 33]
]);
console.log(person4);
```

```
Map(2) { 'firstName' => 'Munna', 'age' => 33 }
```

0.0.24 Add information to object but in a new object using Map

[59]:
```
const person5 = {
    id: 1,
    firstName: 'Munna'
};
const person6 = {
    id: 2,
    firstName: 'Dipti'
};
```

[60]:
```
const extraInfo = new Map();
extraInfo.set(person5, {age: 28, gender: 'Male'});
extraInfo.set(person6, {age: 18, gender: 'Female'});
```

```
console.log(extraInfo);
```

```
Map(2) {
  { id: 1, firstName: 'Munna' } => { age: 28, gender: 'Male' },
  { id: 2, firstName: 'Dipti' } => { age: 18, gender: 'Female' }
}
```

0.0.25 Cloning objects

[61]:
```
const obj1 = {
    key1: 'value1',
    key2: 'value2'
};
```

clone using spread operator

[62]:
```
// clone using spread operator
const obj2 = {...obj1};
obj1.key3 = "value3";
console.log(obj1);
console.log(obj2);
```

```
{ key1: 'value1', key2: 'value2', key3: 'value3' }
{ key1: 'value1', key2: 'value2' }
```

cloning using Object.assign

[65]:
```
// cloning using Object.assign
const obj5 = Object.assign({}, obj1);
obj1.key4 = "value4";
console.log(obj1);
console.log(obj5);
```

```
{ key1: 'value1', key2: 'value2', key3: 'value3', key4: 'value4' }
{ key1: 'value1', key2: 'value2', key3: 'value3' }
```

[ ]:
```

```

JS_Beginning_to_Mastery_Part1_3

April 21, 2023

[1]:
```
const myArray = ["value1", "value2", "value3", "value4", 1, 2, 3, 4, 5, 6];

// delete item 2
myArray.splice(1,1); // starting from index 1, delete 1 item
console.log(myArray);
```

```
[ 'value1', 'value3', 'value4', 1, 2, 3, 4, 5, 6 ]
```

[2]:
```
// delete 6 items starting from index 3
myArray.splice(3,6); // starting from index 3, delete 6 items
console.log(myArray);
```

```
[ 'value1', 'value3', 'value4' ]
```

[3]:
```
const deletedItem = myArray.splice(1,1); // starting from index 1, delete 1 item
console.log(deletedItem);
```

```
[ 'value3' ]
```

[4]:
```
console.log(myArray);
```

```
[ 'value1', 'value4' ]
```

[5]:
```
// insert
myArray.splice(1, 0, "inserted item"); // insert at index 1, no delete as 0 is␣
 ↪passed, and the item to be inserted
console.log(myArray);
```

```
[ 'value1', 'inserted item', 'value4' ]
```

[6]:
```
// insert and delete at the same time
// will delete two items from index 1 and then add two elements
myArray.splice(1, 2, "new inserted item1", "new inserted item4");
console.log(myArray);
```

```
[ 'value1', 'new inserted item1', 'new inserted item4' ]
```

[7]:
```
// I can also store the deleted items when using splice
const deletedItems2 = myArray.splice(1, 2, "new inserted item5", "new inserted␣
 ↪item6");
console.log(deletedItems2);
```

```
[ 'new inserted item1', 'new inserted item4' ]
```

[8]:
```
console.log(myArray);
```

```
[ 'value1', 'new inserted item5', 'new inserted item6' ]
```

[9]:
```
// object is not iterable (cannot read property Symbol(Symbol.iterator))

// string, arrays are iterable

// array like objects
// which has length property and index based access
// string is array like object

const firstName = "John";
console.log(firstName.length);
console.log(firstName[2]);
```

```
4
h
```

0.0.1 Working with Set

[10]:
```
const numbers = new Set([1,2,3,3]);
console.log(numbers);
```

```
Set(3) { 1, 2, 3 }
```

no index-based access, no duplicates, no order

[11]:
```
// add
numbers.add(55);
console.log(numbers);
```

```
Set(4) { 1, 2, 3, 55 }
```

[12]:
```
// delete
numbers.delete(2);
console.log(numbers);
```

```
Set(3) { 1, 3, 55 }
```

[13]:
```
// has
console.log(numbers.has(3));
```

```
true
```

[14]:
```
// clear
numbers.clear();
console.log(numbers);
```

```
Set(0) {}
```

[15]:
```
// size
console.log(numbers.size);
```

```
0
```

[16]:
```
const numbers2 = new Set([1,2,3,4,5,6,7,8,9,10]);
// forEach
numbers2.forEach((value, key) => {
    console.log(key, value);
})
```

```
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
```

[17]:
```
// convert to array
const numbersArray = Array.from(numbers2);
console.log(numbersArray);
```

```
[
  1, 2, 3, 4,  5,
  6, 7, 8, 9, 10
]
```

[18]:
```
// convert to array
const numbersArray2 = [...numbers2];
console.log(numbersArray2);
```

```
[
  1, 2, 3, 4,  5,
  6, 7, 8, 9, 10
]
```

[19]:
```
// convert to array
const numbersArray3 = [...new Set([1,2,3,3,4,5,6,7,8,9,10])];
console.log(numbersArray3);
```

```
[
  1, 2, 3, 4,  5,
  6, 7, 8, 9, 10
]
```

[20]:
```
// convert to array
const numbersArray4 = Array.from(new Set([1,2,3,3,4,5,6,7,8,9,10]));
console.log(numbersArray4);
```

```
[
  1, 2, 3, 4,  5,
  6, 7, 8, 9, 10
]
```

0.0.2 Optional chaining

[21]:
```
// optional chaining

const user = {
    firstName: "John",
    address: {houseNumber: '1234'}
};

console.log(user.firstName);
console.log(user.address.houseNumber);
```

```
John
1234
```

[22]:
```
const user2 = {
    firstName: "John",
    // address: {houseNumber: '1234'}
};

console.log(user2.firstName);
console.log(user2.address); // returns undefined
// console.log(user2.address.houseNumber);
// the above line will returns error as can't read property of undefined
```

```
John
undefined
```

we want to retun undefined and not error if a property is not there

[23]:
```
console.log(user2?.firstName);
// Does user exist? if yes then return firstName
```

```
John
```

[24]:
```
console.log(user2?.addres?.houseNumber);
// Does user exist? if yes then return address.
// Does address exist? if yes then return houseNumber
```

```
undefined
```

Methods – function inside object

[25]:
```
// methods
// function inside object

const person = {
    firstName: "John",
    age: 33,
    about: function(){
        console.log("Person's name is John and age is 33");
    }
};
```

[26]:
```
// Access  this function or method inside person
console.log(person.about); // returns or print the function
```

```
[Function: about]
```

[27]:
```
// calling the function inside person
person.about(); // returns or print the function
```

```
Person's name is John and age is 33
```

this keyword `this` keyword refers to the object that is executing the current function

[28]:
```
const person2 = {
    firstName: "Lipon",
    age: 43,
    about: function(){
        console.log(`Person's name is ${this.firstName} and age is ${this.
 ↪age}`);
    }
    // We can't write console.log(`Person's name is ${firstName} and age is␣
 ↪${age}`), rather have to use this keyword to access the object's property
    // here this will refer to the object person2 which will be calling the␣
 ↪method about
};

person2.about();
```

```
Person's name is Lipon and age is 43
```

[29]:
```
const person3 = {
    firstName: "Lipon",
    age: 43,
    about: function(){
        console.log(this);
    }
};
```

```
person3.about(); // prints the object person3
```

```
{ firstName: 'Lipon', age: 43, about: [Function: about] }
```

[30]:
```
// methods -- continued
const person4 = {
    firstName: "Shammun",
    age: 33,
    about: function(){
        console.log(this.firstName, this.age);
    }
};
person4.about(); // prints the object person4
```

```
Shammun 33
```

[31]:
```
// in below, this refers to the object that is executing the current function
function personInfo(){
    console.log(`person name is ${this.firstName} and age is ${this.age}`);
}
```

[32]:
```
const person5 = {
    firstName: "Shammun",
    age: 37,
    about: personInfo
};

const person6 = {
    firstName: "Mohsin",
    age: 47,
    about: personInfo
};

const person7 = {
    firstName: "Mokhtar",
    age: 57,
    about: personInfo
};
```

[33]:
```
person5.about();
person6.about();
```

```
person name is Shammun and age is 37
person name is Mohsin and age is 47
```

[34]:
```
person7.about();
```

```
person name is Mokhtar and age is 57
```

0.0.3 use strict

`Use strict` is a special string that can be used in JavaScript code to enable a stricter version of the language, which enforces certain rules and restrictions. When the `use strict` mode is enabled, the JavaScript engine will perform stricter syntax checking and will generate more errors in certain situations that might have been ignored before.

Some of the benefits of using `use strict` include:

Preventing the accidental creation of global variables. In non-strict mode, it's possible to accidentally create a global variable by forgetting to declare it with the `var`, `let`, or `const` keyword. `use strict` mode prevents this by throwing an error whenever a variable is used without being declared first.

Making it easier to write secure code. In strict mode, certain actions that could be used maliciously are disallowed, such as assigning a value to the `eval` function, which can be used to execute arbitrary code.

Improving code quality. Strict mode helps catch common coding mistakes and encourages good coding practices.

To use `use strict` in your JavaScript code, simply include the string `use strict` at the beginning of your script or function. For example:

[1]:
```
"use strict";
function myFunction() {
  // Code in here will be executed in strict mode
}
```

[1]:
```
'use strict'
```

```
console.log(this) // returns window object
```

```
console.log(window) // also returns window object
```

```
console.log(this === window) // returns true
```

```
console.log("Hello") // returns "Hello"
```

[4]:
```
function myFunc(){
    "use strict"
    console.log(this);
}
```

```
//window.myFunc(); // returns window object
myFunc(); // returns undefined
window.myFunc(); // returns window object

// with strict mode, to access window object, we need to use window.myFunc()

function myFunc2(){
    //"use strict"
    console.log(this);
}
```

```
//window.myFunc(); // returns window object
myFunc2(); // returns window object
```

0.0.4 More on functions

[5]:
```
function hello(){
    console.log("Hello world");
}
```

```
hello(); // returns Hello world
window.hello(); // returns Hello world
hello.call(); // returns Hello world
```

0.0.5 call(), apply() and bind()

0.0.6 call()

`call()` method is used to call a function with a given this value and arguments provided individually

[7]:
```
const user1 = {
    firstName: "John",
    age: 8,
    about:  function(){
        console.log(this.firstName, this.age);
    }
}

const user2 = {
    firstName: "Jane",
    age: 9,
}
```

using 'about()" method of user1 object from user1 object

[8]:
```
user1.about.call(user2); // returns Jane 9
```

```
Jane 9
```

[9]:
```
user1.about.call(); // returns undefined undefined
```

```
undefined undefined
```

[10]:
```
user1.about.call(user1) // returns John 8
```

```
John 8
```

[11]:
```
user1.about() // returns John 8
```

```
John 8
```

[12]:
```
const  user3 = {
    firstName: "John",
    age: 8,
    about:  function(hobby, favMusician){
        console.log(this.firstName, this.age, hobby, favMusician);
    }
}

const  user4 = {
    firstName: "Jane",
    age: 9,
}
```

using 'about()" method of user3 object from user4 object

[13]:
```
// returns Jane 9 reading Michael Jackson
user3.about.call(user4, ["reading", "Michael Jackson"]);
```

```
Jane 9 [ 'reading', 'Michael Jackson' ] undefined
```

[14]:
```
user3.about.call(user4, "reading", "Michael Jackson", "reading"); // returns␣
↪Jane 9 reading Michael Jackson
```

```
Jane 9 reading Michael Jackson
```

[15]:
```
user3.about.call(user4, "reading"); // returns Jane 9 reading undefined
```

```
Jane 9 reading undefined
```

[16]:
```
user3.about.call(user4, "reading", "Michael Jackson")
```

```
Jane 9 reading Michael Jackson
```

[17]:
```
user3.about.call(user4, ["reading", "Michael Jackson"], "Michael Jackson"); //␣
↪returns Jane 9 reading Michael Jackson Michael Jackson
```

```
Jane 9 [ 'reading', 'Michael Jackson' ] Michael Jackson
```

[18]:
```
function about2(hobby, favMusician){
    console.log(this.firstName, this.age, hobby, favMusician);
}

const  user5 = {
    firstName: "Monty",
    age: 8,

}

const  user6 = {
    firstName: "Ronty",
```

```
    age: 9,
}
```

[19]:
```
about2.call(user5, "guitar", "Aiyub Bacchu"); // returns Monty 8 guitar Aiyub␣
↪Bacchu
```

```
Monty 8 guitar Aiyub Bacchu
```

0.0.7 apply method

In JavaScript, the `apply()` method is a built-in function that allows you to call a function with a specified `this` value and arguments provided as an array. This method is similar to the `call()` method, but **the arguments are passed as an array rather than individually**.

The syntax for using the `apply()` method is as follows:

```
function myFunction(arg1, arg2, arg3) {
  // function body
}

myFunction.apply(thisValue, [arg1, arg2, arg3]);
```

The first argument passed to `apply()` is the value to be used as the `this` value inside the function, and the second argument is an array containing the arguments to be passed to the function.

Here's an example of how to use `apply()`:

[21]:
```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  fullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};

const args = [1, 2, 3];
const result = myFunction.apply(person, args);
```

In the example above, we have an object `person` with a `fullName` function. We use `apply()` to set the `this` value of the `fullName` function to the `person` object, so that it can access the `firstName` and `lastName` properties of the object. The `args` array is then passed as the second argument to `apply()`.

The `apply()` method can also be used with functions that are not attached to objects, such as:

[23]:
```
function myFunction2(a, b, c) {
  return a + b + c;
}

const args2 = [1, 2, 3];
const result2 = myFunction2.apply(null, args); // 6
```

In the example above, we use `apply()` to call the `myFunction` function and pass `null` as the `this` value (since the function is not attached to an object), and pass `[1, 2, 3]` as the second argument. The result is `6`, which is the sum of the arguments.

[24]:
```
// using call
about2.call(user5, "guitar", "Aiyub Bacchu"); // returns Monty 8 guitar Aiyub␣
↪Bacchu
```

```
Monty 8 guitar Aiyub Bacchu
```

[25]:
```
// using apply
about2.apply(user5, ["guitar", "Aiyub Bacchu"]); // returns Monty 8 guitar␣
↪Aiyub Bacchu
```

```
Monty 8 guitar Aiyub Bacchu
```

0.0.8 bind()

In JavaScript, the `bind()` method is a built-in function that allows you to create a new function with a specified this value and pre-set arguments. This method is useful when you want to create a new function based on an existing function but with some of its arguments pre-set.

The syntax for using the `bind()` method is as follows:

```
function myFunction(arg1, arg2, arg3) {
  // function body
}

const boundFunction = myFunction.bind(thisValue, arg1, arg2);
```

The first argument passed to `bind()` is the value to be used as the `this` value inside the function, and subsequent arguments are the arguments to be pre-set in the new function. The `bind()` method returns a new function that can be called later.

Here's an example of how to use `bind()`:

[27]:
```
const person11 = {
  firstName: 'John',
  lastName: 'Doe',
  fullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};

const printName11 = function(prefix, suffix) {
  console.log(prefix + this.fullName() + suffix);
}

const boundPrintName11 = printName11.bind(person11, 'Mr. ', ' Jr.');
boundPrintName11(); // "Mr. John Doe Jr."
```

```
Mr. John Doe Jr.
```

In the example above, we have an object `person` with a `fullName` function and a `printName` function that takes two arguments. We use `bind()` to create a new function `boundPrintName` that is based on `printName` but with the `this` value set to `person`, and with the first argument pre-set to `'Mr. '` and the second argument pre-set to `' Jr.'`. When we call `boundPrintName()`, it logs `"Mr. John Doe Jr."` to the console.

The `bind()` method is particularly useful when you want to create a new function that can be passed as a callback to another function with some arguments pre-set.

The `call()` method sets the this value and immediately invokes the function, while the `bind()` method creates a new function with the `this` value set, but does not invoke the function immediately. Instead, the new function can be called later.

[29]:
```
printName11.call(person11, 'Mr. ', ' Jr.');
```

```
Mr. John Doe Jr.
```

[31]:
```
about2.call(user5, "guitar", "Aiyub Bacchu"); // returns Monty 8 guitar Aiyub␣
↪Bacchu
```

```
Monty 8 guitar Aiyub Bacchu
```

[32]:
```
about2.apply(user5, ["guitar", "Aiyub Bacchu"]); // returns Monty 8 guitar␣
↪Aiyub Bacchu
```

```
Monty 8 guitar Aiyub Bacchu
```

[33]:
```
const func = about2.bind(user5, "guitar", "Aiyub Bacchu"); // returns a function
func()
```

```
Monty 8 guitar Aiyub Bacchu
```

[34]:
```
about2.bind(user5, "guitar", "Aiyub Bacchu")();
```

```
Monty 8 guitar Aiyub Bacchu
```

[36]:
```
const user12 = {
    firstName: "Shammunul",
    age: 37,
    about: function(){
        console.log(this.firstName, this.age);
    }
}

user12.about(); // returns Shammunul 37
```

```
Shammunul 37
```

[38]:
```
// don't do the following mistake

const myFunc12 = user12.about; // this is not bounded yet
myFunc12(); // returns undefined undefined
```

```
// here this value is window object
```

```
undefined undefined
```

[39]:
```
// do the following if you have to save the function inside an object as a␣
 ↪variable
// if we want to bind this to user1 object, we need to use bind method

const myFunc13 = user12.about.bind(user12);
myFunc13(); // returns Shammunul 37
```

```
Shammunul 37
```

[40]:
```
// Without arrow function

// If the arrow function is not used, then this value will be the object itself␣
 ↪where the function is called

const user0 = {
    firstName: "Shammunul",
    age: 37,
    about: function(){
        console.log(this); // this is user0 object
        console.log(this.firstName, this.age);
    }
}

user0.about(); // returns Shammunul 37
```

```
{ firstName: 'Shammunul', age: 37, about: [Function: about] }
Shammunul 37
```

// arrow function

// arrow function takes this function from the surrounding // scope, so it is not bound to the function itself

```
const user13  = {
    firstName: "Shammunul",
    age: 37,
    about: () => {
        console.log(this); // this is window object
        console.log(this.firstName, this.age);
    }
}

// here this value is window object
user13.about(); // returns window followed by undefined undefined

// For arrow function, even if we use call method, it will not work
user1.about.call(user1); // returns window followed by undefined undefined
```

```
user1.about(user1); // returns window followed by undefined undefined
```

[44]:
```
/*
const user1  = {
    firstName: "Shammunul",
    age: 37,
    about: function(){
        //console.log(this); // this is window object
        console.log(this.firstName, this.age);
    }
}
*/

// The above code is equivalent to the following code

const user14  = {
    firstName: "Shammunul",
    age: 37,
    about(){
        //console.log(this); // this is window object
        console.log(this.firstName, this.age);
    }
}

user14.about();
```

```
Shammunul 37
```

0.0.9 Object oriented programming

[45]:
```
// Object oriented programming concept starts from here
```

[46]:
```
const user = {
    firstName: "Shammunul",
    lastName: "Islam",
    email: "sha_is113@yahoo.com",
    age: 37,
    address: "House 40/10, ABCD R/A, Sylhet, Bangladesh",
    about: function(){
        return `${this.firstName} is ${this.age} years old`;
    },
    is18: function(){
        return this.age >= 18;
    }
}
```

But, if we want to do this for different persons, it will become tedious. A more efficient way is described below.

0.0.10 Factory approach

Now we will create a function that will create object, add key value pairs and return that object.

[47]:
```
function createUser(firstName, lastName, email, age, address){
    const user = {};
    user.firstName = firstName;
    user.lastName = lastName;
    user.email = email;
    user.age = age;
    user.address = address;
    user.about = function(){
        return `${this.firstName} is ${this.age} years old`;
    };
    user.is18 = function(){
        return this.age >= 18;
    };
    return user;
}
```

[49]:
```
const user17 = createUser("Shammunul", "Islam", "sha_is13@yahoo.com", 37, "ABC␣
↪R/A");
```

[50]:
```
console.log(user17);
```

```
{
  firstName: 'Shammunul',
  lastName: 'Islam',
  email: 'sha_is13@yahoo.com',
  age: 37,
  address: 'ABC R/A',
  about: [Function (anonymous)],
  is18: [Function (anonymous)]
}
```

[51]:
```
const is18 = user17.is18();
console.log(is18);
```

```
true
```

[52]:
```
const about = user17.about();
console.log(about);
```

```
Shammunul is 37 years old
```

Factory apporach is not a good approach, because it creates a new function for each object.

A better approach is to create a new object containing the two common methods.

[53]:
```
const userMethods = {
    about: function(){
        return `${this.firstName} is ${this.age} years old`;
    },
    is18: function(){
        return this.age >= 18;
    }
}

function createUser(firstName, lastName, email, age, address){
    const user = {};
    user.firstName = firstName;
    user.lastName = lastName;
    user.email = email;
    user.age = age;
    user.address = address;
    user.about = userMethods.about; // we are not calling the function here␣
 ↪rather we are pointing to the address of the function
    user.is18 = userMethods.is18; // referencing to the function
    return user;
}
```

[55]:
```
const user18 = createUser("Liton", "Miah", "liton22@yahoo.com", 29, "Mirpur,␣
 ↪Dhaka");
const user19 =  createUser("Miron", "Manna", "miron@gmail.com", 44, "Dhanmondi,␣
 ↪Dhaka");
console.log(user18.about());
console.log(user19.about());
console.log(user19.is18())
```

```
Liton is 29 years old
Miron is 44 years old
true
```

0.0.11 A better approach

0.0.12 Object.create()

[57]:
```
const userMethods20 = {
    about: function(){
        return `${this.firstName} is ${this.age} years old`;
    },
    is18: function(){
        return this.age >= 18;
    },
    sing: function(){
        return "la la la la";
    }
```

```
}
```

[58]:
```
function createUser20(firstName, lastName, email, age, address){
    const user = Object.create(userMethods);
    user.firstName = firstName;
    user.lastName = lastName;
    user.email = email;
    user.age = age;
    user.address = address;
    return user;
}
```

[59]:
```
const user20 = createUser("Liton", "Miah", "liton22@yahoo.com", 29, "Mirpur,␣
 ↪Dhaka");
const user21 =  createUser("Miron", "Manna", "miron@gmail.com", 44, "Dhanmondi,␣
 ↪Dhaka");
console.log(user20.about());
console.log(user21.about());
console.log(user21.is18())
```

```
Liton is 29 years old
Miron is 44 years old
true
```

[60]:
```
const obj1 = {
    key1: 'value1',
    key2: 'value2'
}

const obj2 = {
    key3: 'value3',
}

console.log(obj2.key3); // prints value3
console.log(obj2.key1); // prints undefined
```

```
value3
undefined
```

we want to see that if key1 is in obj3 and if it is not there go to obj1 and get the value of key1

[61]:
```
const obj3 = Object.create(obj1); // creates an empty object
// console.log(obj3);
obj3.key3 = "value3";
console.log(obj3.key1); // prints value1 from obj1
```

```
value1
```

[62]:
```
obj3.key2 = "unique";
console.log(obj3.key2); // prints unique
console.log(obj3);
```

```
unique
{ key3: 'value3', key2: 'unique' }
```

Note that `key1` is not here.

We will see the following in a browser if we see the output of `console.log(obj3)`.

`__proto__` is a property of an object also written as `[[prototype]]` in some versions of javascript. It is a reference to the object that is used as a prototype.

[63]:
```
console.log(obj3.__proto__); // prints { key1: 'value1', key2: 'value2' }
```

```
{ key1: 'value1', key2: 'value2' }
```

- `prototype` is different than `__proto__` or `[[prototype]]`
- `prototype` is a property of a function
- `__proto__` is a property of an object – also known as dunderscore proto
- `[[prototype]]` is a property of an object
- `__proto__` is a reference while prototype is an object of function
- `proto` is a reference to the chain being created

[66]:
```
const userMethods22 = {
    about: function(){
        return `${this.firstName} is ${this.age} years old`;
    },
    is18: function(){
        return this.age >= 18;
```

```
    },
    sing: function(){
        return "la la la la";
    }
}

function createUser22(firstName, lastName, email, age, address){
    const user = Object.create(userMethods22);
    user.firstName = firstName;
    user.lastName = lastName;
    user.email = email;
    user.age = age;
    user.address = address;
    return user;
}
```

The line `const user = Object.create(userMethods22);` creates a new object user and sets its prototype to the userMethods object.

The `Object.create()` method creates a new object and sets its prototype to the object passed as an argument. In this case, we're passing the `userMethods22` object as the argument, so the `user` object will inherit all the properties and methods defined on the userMethods object.

By setting the user object's prototype to `userMethods22`, we're creating a "prototype chain". This means that when we call a method on the `user` object, JavaScript first looks for the method on the `user` object itself. If the method isn't found, JavaScript looks for the method on the user object's prototype (`userMethods22`). If the method still isn't found, JavaScript continues up the prototype chain until it reaches the Object.prototype object, which is the default prototype for all objects in JavaScript.

In this example, we're using the `Object.create()` method to create a new user object with its prototype set to `userMethods`. This means that the `user` object will inherit the `about()`, `is18()`, and `sing()` methods defined on `userMethods`, and we can call these methods on the `user` object as if they were defined directly on the `user` object itself.

[69]:
```
const user23 = createUser22("Liton", "Miah", "liton22@yahoo.com", 29, "Mirpur,␣
↪Dhaka");
const user24 =  createUser22("Miron", "Manna", "miron@gmail.com", 44,␣
↪"Dhanmondi, Dhaka");
```

[70]:
```
console.log(user23.about());
```

```
Liton is 29 years old
```

[71]:
```
console.log(user24.about());
```

```
Miron is 44 years old
```

[72]:
```
console.log(user24.is18())
```

```
true
```

[73]:
```
console.log(user23);
```

```
{
  firstName: 'Liton',
  lastName: 'Miah',
  email: 'liton22@yahoo.com',
  age: 29,
  address: 'Mirpur, Dhaka'
}
```

0.0.13 Prototype property of function

In JavaScript, every function is an object, and like any object, it has properties and methods. One of those properties is the prototype property. The prototype property of a function is used to add properties and methods to all instances of the function.

The prototype property is an object that will become the prototype of any objects created using the function as a constructor. This means that any properties or methods added to the prototype will be shared by all instances of the function.

[74]:
```
function hello(){
    console.log("Hello world!");
}
```

[75]:
```
// js function -- function + object

// console.log(hello.name); -- function name

// add your own properties

hello.myProperty = "Property defined by me";
console.log("hello.myProperty = " + hello.myProperty);
```

```
hello.myProperty = Property defined by me
```

[76]:
```
// only functions provide prototype property

console.log(hello.prototype); // returns constructor function
```

```
{}
```

We will see the following in a browser if we see the output of `console.log(hello.prototype);`.

[77]:
```
hello.prototype.abc = "abc";
console.log(hello.prototype.abc);

hello.prototype.xyz = "xyz";
console.log(hello.prototype.xyz);

hello.prototype.sing = function(){
    return "la la la";
}
console.log(hello.prototype.sing());
```

```
abc
xyz
la la la
```

proto is property of object and it is a reference to an object

We will add the functionalities of the `userMethods` object to the prototype of the `createUser` function. `prototype` is simply an object

[78]:
```
// createUser is a constructor function
function createUser(firstName, lastName, email, age, address){
    // creates a new object user and sets its prototype to the createUser.
 ↪prototype object
    // __proto__ is being set to the prototype of the createUser function
    const user = Object.create(createUser.prototype); // create sets the␣
 ↪proto's value. Here it sets to prototype of createUser function
```

```
    // proto is a reference to the chain being created
    // In this case, proto will be an object

    user.firstName = firstName;
    user.lastName = lastName;
    user.email = email;
    user.age = age;
    user.address = address;
    return user;
}
```

The line `const user = Object.create(createUser.prototype);` creates a new object user and sets its prototype to the `createUser.prototype` object.

In JavaScript, every function has a special prototype property that is used as the prototype for objects created using the function as a constructor. When you create a new object using the new keyword and a constructor function, the object's prototype is automatically set to the constructor's prototype property.

In this example, we're creating a new object `user` that has its prototype set to `createUser.prototype`. This means that the `user` object will inherit any properties or methods defined on `createUser.prototype`.

The `Object.create()` method is used to create a new object and set its prototype to a specific object. In this case, we're passing `createUser.prototype` as the argument to `Object.create()`, so the `user` object will inherit from `createUser.prototype`.

By setting the user object's prototype to `createUser.prototype`, we're creating a prototype chain that allows us to define shared methods and properties for all objects created using the `createUser` constructor function.

In this example, `createUser.prototype` is an object that can be used to define shared methods and properties for all `user` objects created using the `createUser` constructor function. This means that any methods or properties defined on `createUser.prototype` will be inherited by all user objects created using the `createUser` constructor function.

[79]:
```
createUser.prototype.about = function(){
    return `${this.firstName} is ${this.age} years old`;
}

createUser.prototype.is18 = function(){
    return this.age >= 18;
}

createUser.prototype.sing = function(){
    return "la la la la";
}
```

[79]:
```
[Function (anonymous)]
```

[80]:
```
console.log(createUser.prototype);
```

```
{
  about: [Function (anonymous)],
  is18: [Function (anonymous)],
  sing: [Function (anonymous)]
}
```

[81]:
```
const user28 = createUser("Liton", "Miah", "liton22@yahoo.com", 29, "Mirpur,␣
 ↪Dhaka");
const user29 =  createUser("Miron", "Manna", "miron@gmail.com", 44, "Dhanmondi,␣
 ↪Dhaka");
console.log(user28);
console.log(user28.about());
console.log(user29.about());
console.log(user29.is18())

console.log(user28.sing())
```

```
createUser {
  firstName: 'Liton',
  lastName: 'Miah',
  email: 'liton22@yahoo.com',
  age: 29,
  address: 'Mirpur, Dhaka'
}
Liton is 29 years old
Miron is 44 years old
true
la la la la
```

0.0.14 new

new keyword – creates empty object

[82]:
```
function createUser(firstName, age){
    this.firstName = firstName;
    this.age = age;
}

createUser.prototype.about = function(){
    console.log(`${this.firstName} is ${this.age} years old`);
}
```

[82]:
```
[Function (anonymous)]
```

[83]:
```

```

```
// Object.create(createUser.prototype) -- this work is automatically done by␣
 ↪the new keyword
// But, we had to do it separately in the previous file
const user31 = new createUser('John', 25);
console.log(user31); // returns createUser {firstName: "John", age: 25}
```

```
createUser { firstName: 'John', age: 25 }
```

[84]:
```
// this is why, we don't need to do this "Object.create(createUser.prototype)"
// before calling the about method

console.log(user31.about())
```

```
John is 25 years old
undefined
```

0.0.15 Constructor function

In JavaScript, a constructor function is a special function that is used to create and initialize objects. The purpose of a constructor function is to provide a blueprint for creating new objects of a specific type.

To create a constructor function, you can define a function with a capitalized name, which conventionally indicates that it is a constructor function. Within the constructor function, you can use the "this" keyword to define properties and methods that will be associated with instances of the object created by the constructor.

[85]:
```
// constructor function name should start with capital letter and should be in␣
 ↪camel case

// constructor function
function CreateUser(firstName, lastName, email, age, address){
    // using new to create new empty object will set something like "this = {}"␣
 ↪here
    // that's why the following line is not needed
    //const user = Object.create(createUser.prototype)
    // and because of the new keyword, we can just use this to set the␣
 ↪properties
    // and this is different than what we have done
    // previously (or, in file58.js)
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
    this.age = age;
    this.address = address;
    // we can write ""return this;"" here but it is not needed
}
```

[86]:
```
CreateUser.prototype.about = function(){
    return `${this.firstName} is ${this.age} years old.`;
};
CreateUser.prototype.is18 = function (){
    return this.age >= 18;
}
CreateUser.prototype.sing = function (){
    return "la la la la ";
}
```

[86]:
```
[Function (anonymous)]
```

[88]:
```
const user34 = new CreateUser('harshit', 'vashsith', 'harshit@gmail.com', 18,␣
 ↪"my address");
const user35 = new CreateUser('harsh', 'vashsith', 'harshit@gmail.com', 19, "my␣
 ↪address");
const user36 = new CreateUser('mohit', 'vashsitha', 'harshit@gmail.com', 17,␣
 ↪"my address");
console.log(user34);
console.log(user34.is18());
```

```
CreateUser {
  firstName: 'harshit',
  lastName: 'vashsith',
  email: 'harshit@gmail.com',
  age: 18,
  address: 'my address'
}
true
```

[89]:
```
for(let key in user34){
    console.log(key); // will print all the properties of user1, even in the␣
 ↪prototypes
}
```

```
firstName
lastName
email
age
address
about
is18
sing
```

[91]:
```
// we can also use the following method to get all the properties of an object
// but this will not print the properties of the prototypes

for(let key in user34){
```

```
    if (user34.hasOwnProperty(key)){
        console.log(key);
    }
}
```

```
firstName
lastName
email
age
address
```

[92]:
```
let numbers = [1, 2, 3];
console.log(Array.prototype);
// In fact, Array is called internally by JS as new Array()
// and thus we get prototype of Array
```

```
Object(0) []
```

We will see the following in a browser if we see the output of `console.log(Array.prototype);`.

[93]:
```
console.log(Object.getPrototypeOf(numbers)); //
```

```
Object(0) []
```

The above code will also similar out put in the console as can be seen in the above image.

[94]:
```
function hello(){
    console.log("Hello");
}

console.log(Object.getPrototypeOf(hello)); //
```

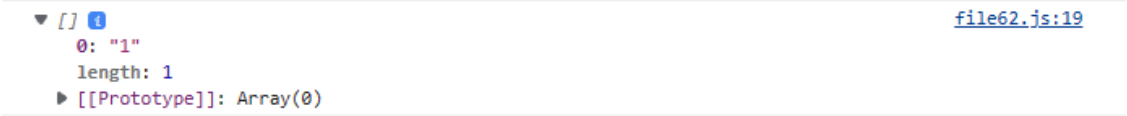
```
{}
```

We will see the following in the console of a browser

[95]:
```
console.log(hello.prototype);
```

```
{}
```

We will see the following in the console of a browser




```
hello.prototype.push('1');
console.log(hello.prototype);
```

The above will show the output below in the console of a browser:

We have already done this:

```
function CreateUser(firstName, lastName, email, age, address){
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
    this.age = age;
    this.address = address;
    // we can write ""return this;"" here but it is not needed
}
CreateUser.prototype.about = function(){
    return `${this.firstName} is ${this.age} years old.`;
};
CreateUser.prototype.is18 = function (){
    return this.age >= 18;
}
CreateUser.prototype.sing = function (){
    return "la la la la ";
}


const user1 = new CreateUser('harshit', 'vashsith', 'harshit@gmail.com', 18, "my address");
const user2 = new CreateUser('harsh', 'vashsith', 'harshit@gmail.com', 19, "my address");
const user3 = new CreateUser('mohit', 'vashsitha', 'harshit@gmail.com', 17, "my address");
```

But, we can also do this using `class`.

classes are fake in JS, but they are useful for documentation

[98]:
```
class CreateUser30{
    constructor(firstName, lastName, email, age, address){
        this.firsName = firstName;
        this.lastName = lastName;
        this.email;
        this.age = age;
        this.address = address;
    }

    about(){
        return `${this.firstName} is ${this.age} years old.`;
    }

    is18(){
        return this.age >= 18;
    }

    sing(){
        return "la la la la";
    }
}
```

[99]:
```
const user41 = new CreateUser('harshit', 'vashsith', 'harshit@gmail.com', 18,␣
 ↪"my address");
const user42 = new CreateUser('harsh', 'vashsith', 'harshit@gmail.com', 19, "my␣
 ↪address");
const user43 = new CreateUser('mohit', 'vashsitha', 'harshit@gmail.com', 17,␣
 ↪"my address");
```

[100]:
```
console.log(user41.about());
```

```
harshit is 18 years old.
```

0.0.16 class inheritance

0.0.17 extends

In JavaScript, the `extends` keyword is used to create a new class that is a child or subclass of an existing class. This is known as class inheritance.

The syntax for creating a subclass using the `extends` keyword is as follows:

```
class Subclass extends Superclass {
  // subclass definition goes here
}
```

In this syntax, the `Subclass` is the new class that is being created, and `Superclass` is the class that the new class is inheriting from. The subclass definition goes inside the curly braces.

By using `extends`, the subclass inherits all of the properties and methods of the superclass, and can also add new properties and methods, or override existing ones. This allows for code reuse and helps to keep the code organized.

Here is an example of using `extends` to create a subclass of a Person class:

[104]:
```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log("Hello, my name is " + this.name + " and I am " + this.age + "␣
 ↪years old.");
  }
}

class Student extends Person {
  constructor(name, age, grade) {
    super(name, age);
    this.grade = grade;
  }

  study() {
    console.log(this.name + " is studying for the " + this.grade + " grade.");
  }
}

var john = new Student("John", 15, "9th");

john.greet(); // outputs "Hello, my name is John and I am 15 years old."
john.study(); // outputs "John is studying for the 9th grade."
```

```
Hello, my name is John and I am 15 years old.
John is studying for the 9th grade.
```

In this example, the `Student` class is created as a subclass of the `Person` class using the `extends` keyword. The `Student` class has a new property `grade` and a new method `study`, while still inheriting the `name`, `age` and `greet` method from the Person class.

The `super` keyword is used in the constructor of the `Student` class to call the constructor of the `Person` class and pass in the `name` and `age` arguments. This ensures that the `name` and `age` properties are initialized properly in the `Student` object.

[101]:
```
class Animal{
    constructor(name, age){
        this.name = name;
        this.age = age;
```

```
    }

    eat(){
        return `${this.name} is eating`;
    }

    isSuperCute(){
        return this.age <= 1;
    }

    isCute(){
        return true
    }
}
```

[102]:
```
class Dog extends Animal{

}
```

[103]:
```
const tommy = new Dog("tommy", 3);
console.log(tommy);
console.log(tommy.isCute());
```

```
Dog { name: 'tommy', age: 3 }
true
```

[106]:
```
// super
class Animal3{
    constructor(name, age){
        this.name = name;
        this.age = age;
    }

    eat(){
        return `${this.name} is eating`;
    }

    isSuperCute(){
        return this.age <= 1;
    }

    isCute(){
        return true;
    }
}

class Dog3 extends Animal3{
    constructor(name, age, speed){
```

```
        super(name, age)
        this.speed = speed;
    }

    run(){
        return `${this.name} is running at ${this.speed} km/hr`;
    }
}

// objects/instances

const tommy3 = new Dog3("tommy", 3, 44);
console.log(tommy3.run());
```

```
tommy is running at 44 km/hr
```

0.0.18 Getter and setter methods

In JavaScript, getter and setter methods are used to get and set the values of an object's properties. They are typically used within classes to provide a way to access and manipulate the internal state of an object, while still maintaining control over how that state is accessed and modified from outside the class.

Getter methods are used to get the value of a property, while setter methods are used to set the value of a property. They are defined using the `get` and `set` keywords respectively, followed by the name of the property. Here is an example:

[110]:
```
class Person4{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    // By using get
    // we can use fullName like property
    // now, we don't need to call it like function or use ()
    // person.fullName will give back the full name
    // we can call this later without ()
    get fullName(){
        return `${this.firstName} ${this.lastName}`;
    }

    // By using set, we can set the value of
    // the property
    // now we can call this like property
    // we can call this later without ()
    set fullName(fullName){
```

```
        const [firstName, secondName] = fullName.split(" ");
        this.firstName = firstName;
        this.lastName = secondName;
    }

    setName(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

[111]:
```
const person13 = new Person4("harshit", "vashsith", 18);
// console.log(person1.fullName()); // will give error

// will give the full name without using () -- this is due to the get method
console.log(person13.fullName);
```

```
harshit vashsith
```

[112]:
```
person13.fullName = "Motin Miah"; // this is due to the set method
console.log(person13);
console.log(person13.fullName);
console.log(person13.firstName);
```

```
Person4 { firstName: 'Motin', lastName: 'Miah', age: 18 }
Motin Miah
Motin
```

[118]:
```
person13.setName("Poran Theon", "Dilan")
```

[119]:
```
console.log(person13);
console.log(person13.fullName);
console.log(person13.firstName);
```

```
Person4 { firstName: 'Poran Theon', lastName: 'Dilan', age: 18 }
Poran Theon Dilan
Poran Theon
```

[117]:
```
person13.fullName = "Thierry Henry"
console.log(person13);
console.log(person13.fullName);
console.log(person13.firstName);
```

```
Person4 { firstName: 'Thierry', lastName: 'Henry', age: 18 }
Thierry Henry
Thierry
```

0.0.19 Static methods and properties

In JavaScript, static methods and properties are class-level members that are associated with the class rather than with instances of the class.

Static methods and properties are declared using the `static` keyword within a class declaration. Here's an example:

[120]:
```
class MyClass {
  static myStaticProperty = 42;

  static myStaticMethod() {
    console.log('Hello from my static method!');
  }
}
```

In the example above, `myStaticProperty` and `myStaticMethod` are both defined as static members of the `MyClass` class.

Static properties can be accessed using the class name directly, like this: `MyClass.myStaticProperty`.

Static methods can also be called directly on the class, like this: `MyClass.myStaticMethod()`.

One common use case for static methods and properties is when you want to define utility functions or constants that are not tied to any particular instance of the class. Another use case is when you want to implement factory methods that create instances of a class in a specific way.

Here's an example of using a static method to create instances of a class:

[123]:
```
class Person14 {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  static createFromFullName(fullName) {
    const [firstName, lastName] = fullName.split(' ');
    return new Person14(firstName, lastName);
  }
}

const john14 = Person14.createFromFullName('John Smith');
console.log(john14.firstName); // Output: John
console.log(john14.lastName); // Output: Smith
```

```
John
Smith
```

In this example, the `createFromFullName` method is a static method that creates instances of the `Person14` class from a full name string. The method is called directly on the `Person14` class, and it returns a new instance of the class.

[124]:
```
// static method and properties

// they are related to class

// static methods and properties are not related to the object

class Person15{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    static classInfo(){
        return "This is a class for Person";
    }

    static desc = "static properties";

    get fullName(){
        return `${this.firstName} ${this.lastName}`;
    }

    set fullName(fullName){
        const [firstName, secondName] = fullName.split(" ");
        this.firstName = firstName;
        this.lastName = secondName;
    }
}

const person15 = new Person15("harshit", "vashsith", 18);
const info = Person15.classInfo();
console.log(info);
```

```
This is a class for Person
```

[126]:
```
console.log(Person15.desc);
```

```
static properties
```

In the console of a web browser, `console.log(Person15.desc)` will show **this is person class**.

// We can't do the following with static method

```
person15.classInfo() // this will throw error
```

Static methods help in application initialization.

[ ]:
```

```

JS_Beginning_to_Mastery_Part2_1

April 21, 2023

1 How JavaScript works?

JavaScript is an interpreted language, which means that the code is executed directly by an interpreter, rather than being compiled into machine code beforehand like in traditional compiled languages such as C or Java.

When you run JavaScript code in a web browser or on the server side with Node.js, the code is interpreted by the JavaScript engine in the runtime environment. The interpreter reads the code line by line, converts it to machine code, and executes it.

However, there are some tools and techniques that can be used to compile JavaScript code for various purposes, such as optimizing performance or transpiling modern JavaScript syntax to older syntax that is more widely supported. These tools include Babel, which is a popular tool for transpiling modern JavaScript code to older syntax, and various tools for minifying and optimizing JavaScript code.

In summary, JavaScript is primarily an interpreted language, but there are also tools and techniques available for compiling and optimizing JavaScript code.

Compilation is the process of translating human-readable source code into machine-executable code that a computer can understand.

In compiled languages, the source code is translated into machine code beforehand, which is then executed by the computer. This machine code can be directly executed by the computer's CPU, without requiring any additional translation or interpretation.

Compiled languages typically offer better performance and security than interpreted languages, because the code is optimized and checked for errors before it is executed. However, they can also be more difficult to develop in, because the development process requires the additional step of compiling the code before it can be tested or executed.

Examples of compiled languages include C, C++, Java, and Rust.

In contrast, interpreted languages, such as JavaScript and Python, are executed directly by an interpreter, which reads and executes the code line by line, without requiring any prior compilation. Interpreted languages typically offer faster development times and greater flexibility, but may sacrifice some performance and security.

JavaScript is a high-level, dynamically-typed, interpreted programming language that is primarily used to create interactive web pages and web applications. It is executed by a JavaScript engine, which is a software component that reads and executes JavaScript code.

When a web page containing JavaScript code is loaded in a web browser, the JavaScript engine parses the code and executes it line by line. It can also interact with other parts of the web page, such as the Document Object Model (DOM) and the browser's window object, in order to create interactive functionality.

JavaScript code can also be executed on the server side using the Node.js runtime environment, which allows JavaScript to be used for creating back-end web applications and services.

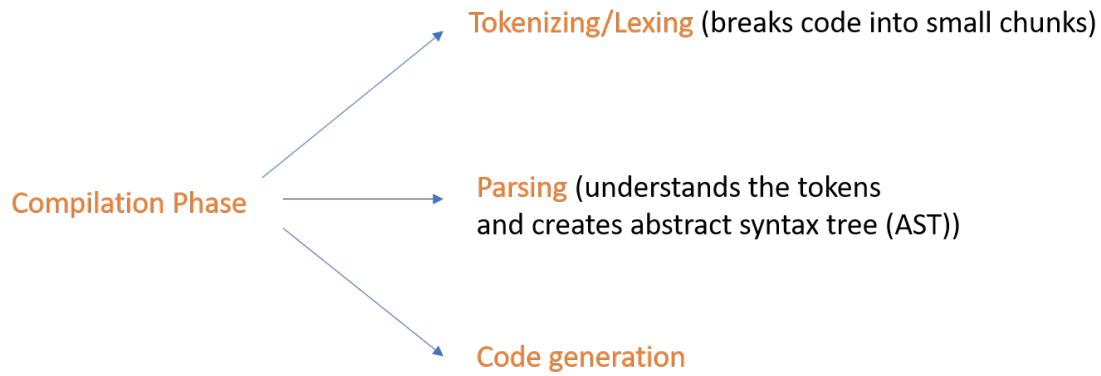
Some key features of JavaScript include its ability to manipulate web page content and styles in real time, handle user input and events, and interact with server-side resources via asynchronous requests such as XMLHttpRequests or the newer Fetch API. It also supports a wide range of programming paradigms, including object-oriented, functional, and procedural programming.

In recent years, JavaScript has also expanded beyond its original use case of client-side web development, and is now widely used for server-side development, desktop and mobile application development, and even for creating complex Internet of Things (IoT) devices.

JavaScript does the following steps sequentially:

1. Compile (first compile the code)/parse
2. Code execute

1.0.1 Compilation phase



Note: JS is also called lexical scoped language.

In original ES documentation, it is written that:

1. Early error checking needs to be done before code execution
2. Determining appropriate scope for variables

But to do these 2, we have to parse the code. So different browsers have different techniques for parsing the JS code or for compiling the code.

So, before the first code is executed, the code is first compiled.

Suppose our code is like below:

```
console.log(this);
console.log(window);
console.log(firstName);
var firstName = ."John";
```

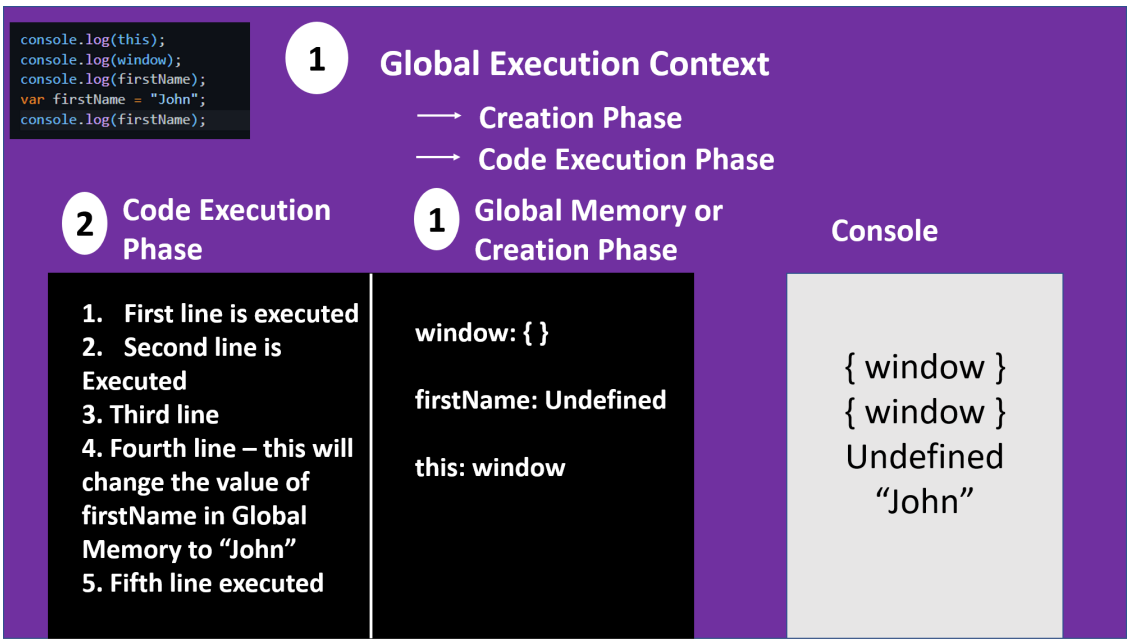
Now, even before executing the first line, this will throw error, as the last line has syntax error (extra `.`). Due to the parsing of all the code, this error is caught.

1.0.2 Code execution

In JS, code executes inside **execution context**.

Before executing first line, first Global Execution Context is created. It has two parts:

- Creation phase (sets some variables to global memory)
- Code execution phase



If we had used `let` to define variable `firstName`, then it would have the value `uninitialized` in Global Memory or in the Creation Phase of the Global Ececution Context. Thus, calling this or printing this before initialzation will give `Uncaught ReferenceError`.

JS is synchronous programming language. When first line is executed, the execution of second line will not start unless the first line gets finished.

Browser gives us the feature of asynchronous programming. It is single threaded.

1.0.3 Hoisting

Hoisting in JavaScript is a mechanism that allows variable and function declarations to be moved to the top of their respective scope at the compile time, regardless of where they were declared within the scope. This means that you can use a variable or a function before it has been declared, without getting a reference error.

In more technical terms, hoisting is the behavior of the JavaScript interpreter where it moves all variable and function declarations to the top of the current scope before executing any code. This is done to ensure that the code runs correctly, as JavaScript is an interpreted language that executes code line by line.

[1]:
```
console.log(a);
var a = 10;
```

```
undefined
```

At first glance, it might seem like this code would throw an error because a is being used before it has been declared. However, due to hoisting, the variable declaration is moved to the top of the scope, so the code is actually interpreted as:

```
var a;
console.log(a);
a = 10;
```

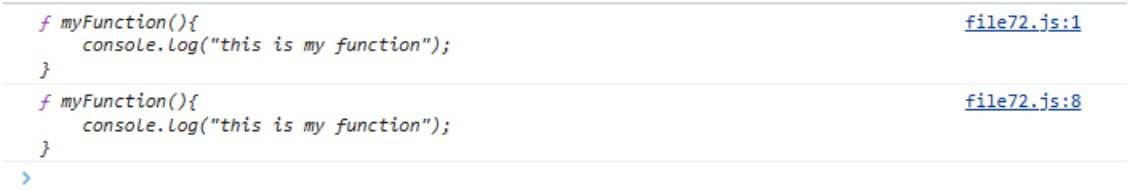
This means that `a` is defined as `undefined` when it is logged to the console, rather than throwing an error. It's important to note that only the declaration is hoisted, not the initialization, so if you try to use a variable that has been declared but not initialized, it will still be `undefined`.

It's also important to note that hoisting only affects variable and function declarations, not variable assignments or function expressions. So if you try to use a variable that has been declared with `let` or `const` before it has been initialized, you will get a `ReferenceError`. Similarly, function expressions are not hoisted, so if you try to use a function expression before it has been defined, you will also get a `ReferenceError`. Same error will also occur if we call variable declared with `const` before initialization.

For function declaration

```
console.log(myFunction);

// function declaration
function myFunction(){
    console.log("this is my function");
}

console.log(myFunction);
```

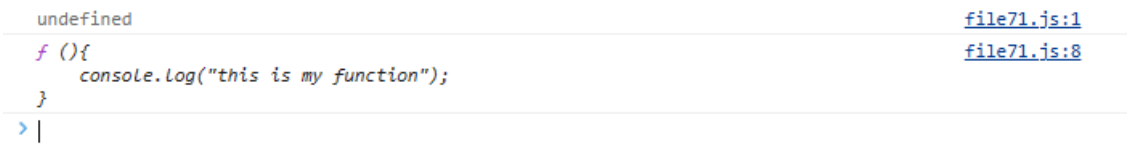
The output in the console will be



For function expression

```
console.log(myFunction);

// function expression
var myFunction = function(){
    console.log("this is my function");
}

console.log(myFunction);
```

The output in the console will be



[2]:
```
let foo = "foo";
console.log(foo);

function getFullName(firstName, lastName) {
    // arguents is array-like object which has index and length
    console.log(arguments); // we can also use arguments[0] or arguments[1]
    // we can also use arguments.length
    let myVar = "var inside func";
    console.log(myVar);
    const fullName = firstName + " " + lastName;
    return fullName;
}

const personName = getFullName("John", "Doe");
console.log(personName);
```

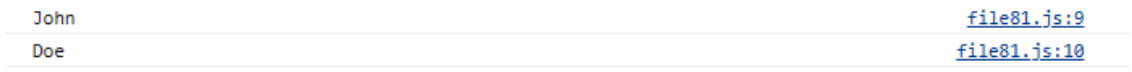
```
foo
[Arguments] { '0': 'John', '1': 'Doe' }
var inside func
John Doe
```

1.0.4 Lexical environment, scope chain

```
// lexical environment, scope chain
```

```
const lastName = "Doe";

const printName = function(){
    const firstName = "John";

    console.log(firstName);
    console.log(lastName);
}
printName();
```

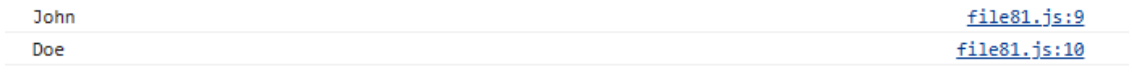
The output in the console will be



[3]:
```
// lexical environment, scope chain

const lastName = "Doe";

const printName = function(){
    const firstName = "John";

    function myFunction(){
        console.log(firstName);
        console.log(lastName);
    }
    myFunction();
}
printName();
```

```
John
Doe
```

The output in the console will again be the same



1.0.5 Closures

We need to know htat functions can return functions.

[4]:
```
function outerFunction(){
    function innerFunction(){
        console.log("innerFunction");
    }
    return innerFunction;
}
```

```
const ans = outerFunction();
ans();
```

```
innerFunction
```

[7]:
```
function printFullName3(firstName, lastName){
    function printName(){
        console.log(firstName, lastName);
    }
    return printName;
}

const ans3 = printFullName3("John", "Doe");
ans3();
```

```
John Doe
```

When a function inside a function is returned, here, when `printName()` is returned from inside `printFullName3()`, it gets returned with variables in the local memory, or the variables `firstName` and `lastName`. These variables `firstName` and `lastName` are in the closures.

Inner functions can access outer function elements.

[9]:
```
function hello4(x){
    const a = "varA";
    const b = "varB";
    return function(){
        console.log(a, b, x);
    }
}

const ans4 = hello4("arg");
ans4();
```

```
varA varB arg
```

[12]:
```
function myFunction5(power){
    return function(number){
        return number ** power;
    }
}

const cube5 = myFunction5(3);
const ans5 = cube5(2);
console.log(ans5);
```

```
8
```

[13]:
```
myFunction5(2)(8)
```

[13]: 64

Another application for closure – Working differently if called more than once or not even working

[17]:
```
function func7(){
    let counter = 0;
    return function(){
        if(counter < 1){
            console.log("Hi! Thanks for calling me the first time!");
            counter++;
        }
        else{
            console.log("I have already been called once!");
        }
    }
}

const myFunc7 = func7();
```

[18]:
```
myFunc7();
```

```
Hi! Thanks for calling me the first time!
```

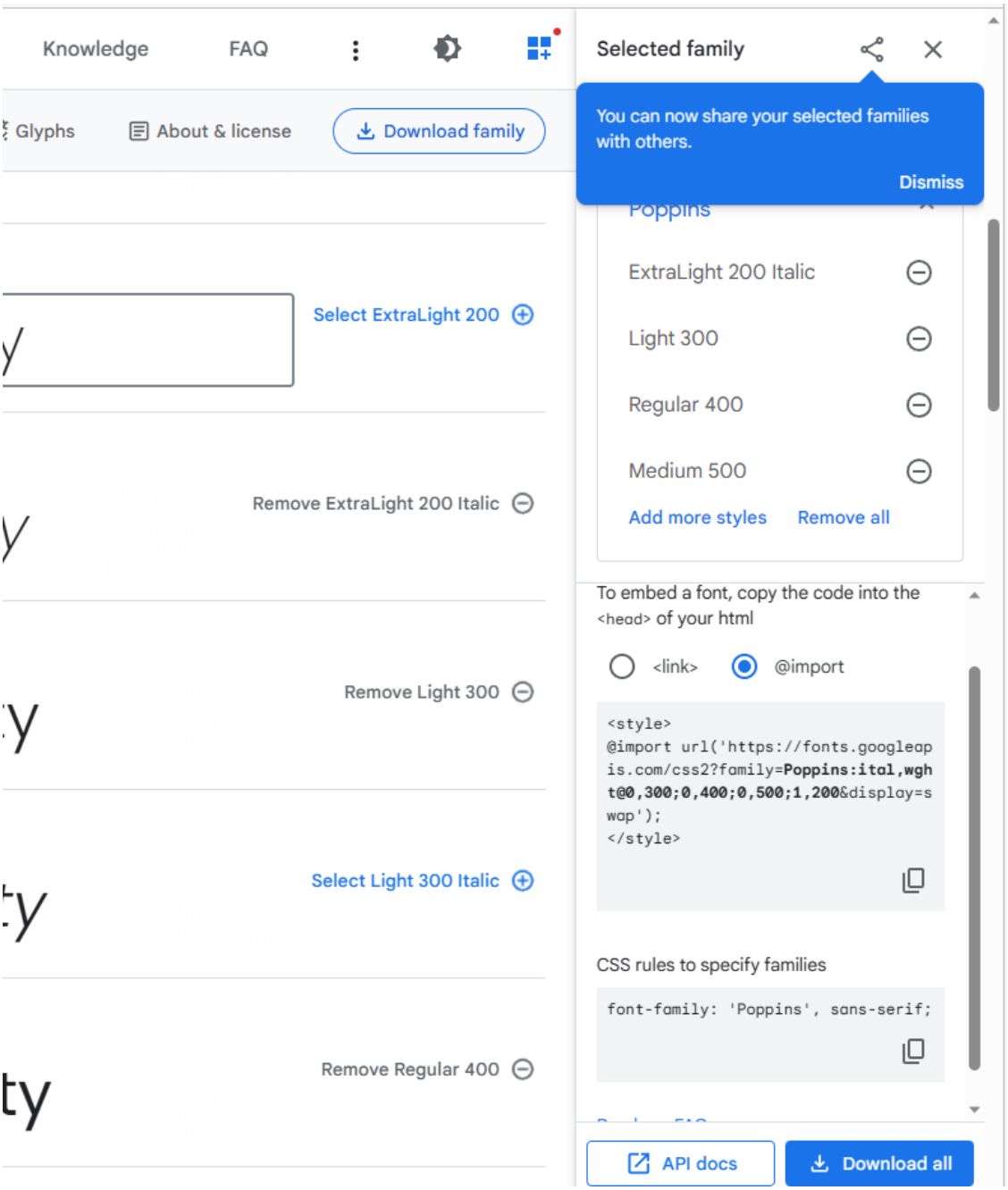
[19]:
```
myFunc7();
```

```
I have already been called once!
```

1.0.6 Working with DOM

For selecting different fonts, we can use this website: https://fonts.google.com/specimen/Poppins

After selecting different fonts, we can copy the `@import url(...` section of the code shown in the bottom right corner.



In the CSS file, after copying the `@import url(...` part, copy also the part written under the section **CSS rules to specify families**. Put this inside body{} tag as below:

```
body{
    font-family: 'Poppins', sans-serif;
}
```

For selecting nice background, we can use `pexels.com` and search for wallpaper.

VSCode shortcut To write the below html code:

```
<section class="section-signup"></section>
```

In VSCode, write `section.section-signup` and then press TAB to autocomplete this line.

1.0.7 Linking JS to HTML file

1st way If we link our js file inside `<head>` tag in HTML, it will throw error if the if the JS is using the HTML elements below as the HTML elements used in this file below are not read or loaded yet.

Don't do the following:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="./style.css">
    <!--
        We can link our js file here using below:
        <script src="./102.js"></script>
        But this will throw error if the JS is using the HTML elements below
        as the HTML elements used in this file below are not read or loaded yet

    -->
    <script src="./102.js"></script>
    <title>My Website</title>
```

2nd way A better approach will be to use it just before the end of `<body>` tag.

Do the following:

```
        </form>
    </section>
    <script src="./102.js"></script>
</body>
```

But even this latter option also has some issues. In this case, first HTML part will be parsed, then JS will load, and then JS will be executed. These will take some time.

3rd way Third approach to load JS is to put script inside `<head>` tag and before `<title>` tag as before but adding `async` like this: `<script src="./102.js" async></script>` .Have a look at example code below:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="./style.css">
    <!--
```

```
        We can link our js file here using below:
        <script src="./102.js"></script>
        But this will throw error if the JS is using the HTML elements below
        as the HTML elements used in this file below are not read or loaded yet

    -->
    <!--
        <script src="./102.js"></script>
    -->
    <script src="./102.js" async></script>

    <title>My Website</title>
</head>
```

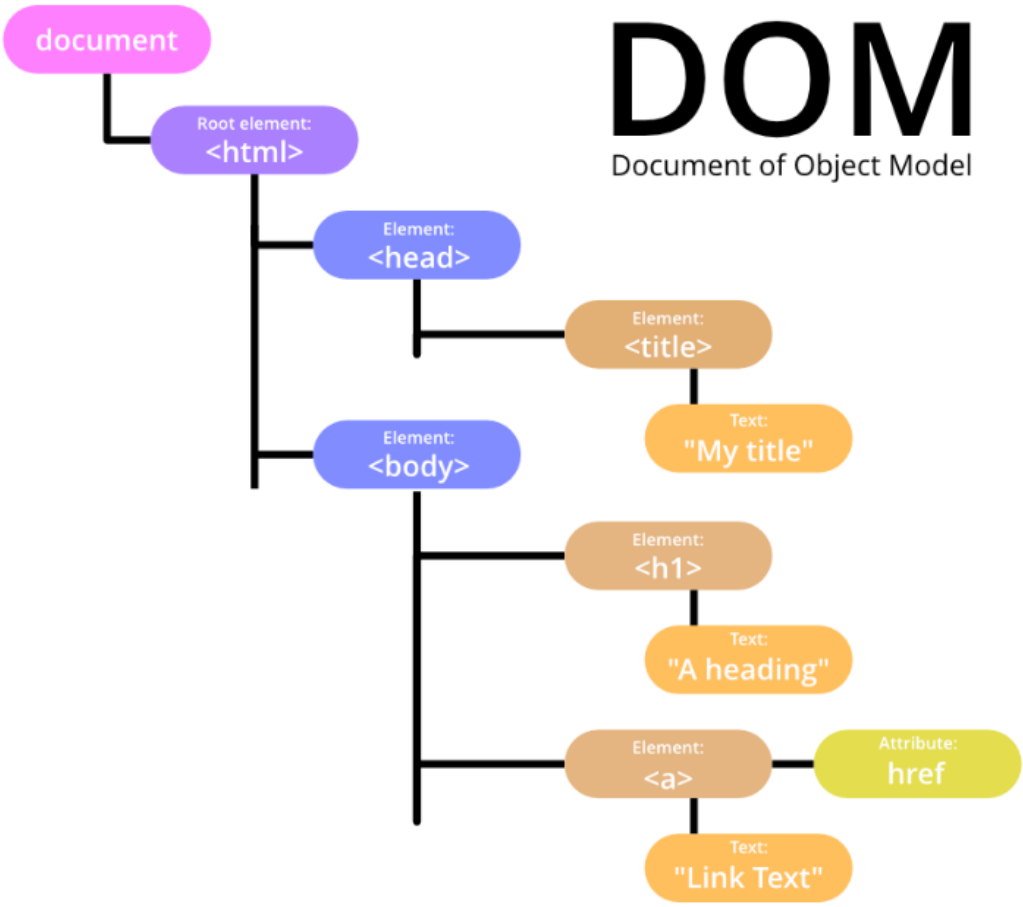
This will allow to continue parsing HTML even after reaching the `<script>` tag while also continuing the load of JS file. Parsing HTML and loading JS will go together in this case. But the problem is as soon as the JS file is loaded, parsing of HTML file will stop and the execution of JS file will start. The probability of error coming is very high as the full HTML file is not parsed yet.

4th way – the best approach Put script inside `<head>` tag and before `<title>` tag as before but adding `defer` like this: `<script src="./102.js" defer></script>`

This will allow to continue parsing HTML even after reaching the `<script>` tag while also continuing the load of JS file. Parsing HTML and loading JS will go together in this case. Compared to the previous approach, in this case as soon as the JS file is loaded, parsing of HTML file will not stop and the execution of JS file will start after the parsing of HTML file is completed.

This will increase the performance of the website. No chance of error in this case.

1.0.8 Document Object Model (DOM)



Source:

The above figure is taken from Geeks for Geeks website

Will create object of key-value pairs. This object is called `document`. Then browser will add `document` inside `window` object residing inside JS.

Now, we have `index.html` as following:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="./style.css">
    <!--
        We can link our js file here using below:
        <script src="./102.js"></script>
        But this will throw error if the JS is using the HTML elements below
        as the HTML elements used in this file below are not read or loaded yet
```

```
    -->
    <!--
        1st approach:
        <script src="./102.js"></script>
    -->
    <!--
        3rd approach:
        <script src="./102.js" async></script>
    -->

    <!--
        4th approach: best approach using defer
        <script src="./102.js" defer></script>
    -->
    <script src="./102.js" defer></script>

    <title>My Website</title>
</head>
<body>
    <header class="header">
        <nav class="nav container">
            <h1 class="log">Website</h1>
            <ul class="nav-items">
                <li><a href="">Home</a></li>
                <li><a href="">To Do</a></li>
                <li><a href="">Sign In</a></li>
            </ul>
        </nav>
        <div class="headline">
            <h2>Manage your tasks</h2>
            <button class="btn btn-headline">Learn More</button>
        </div>
    </header>
    <section class="section-todo container">
        <h2>What do you plan to do today?</h2>
        <form class="form-todo">
            <input type="text" name="" id="" placeholder="Add Todo">
            <input type="submit" value="Add Todo" class="btn">
        </form>
    </section>

    <section class="section-signup container">
        <h2>Sign Up</h2>
        <form class="signup-form">
            <div class="form-group">
                <label for="username">Username</label>
                <input type="text" name="username" id="username">
            </div>
```

```
            <div class="form-group">
                <label for="password">password</label>
                <input type="password" name="password" id="password">
            </div>
            <div class="form-group">
                <label for="confirmPassword">Confirm Password</label>
                <input type="password" name="confirmPassword" id="confirmPassword">
            </div>
            <div class="form-group">
                <label for="email">Email</label>
                <input type="email" name="email" id="email">
            </div>
            <div class="form-group">
                <label for="about">About Yourself</label>
                <textarea name="about" id="about" cols="30" rows="10"></textarea>
            </div>
            <button type="submit" class="btn signup-btn">Submit</button>

        </form>
    </section>
    <!--
        2nd approach: Putting before the end of <body> tag
        <script src="./102.js"></script>
    -->
</body>
</html>
```

We also have `style.css` as following:

```
@import url('https://fonts.googleapis.com/css2?family=Poppins:ital,wght@0,300;0,400;0,500;600;

*{
    box-sizing: border-box;
    padding: 0;
    margin: 0;
}

body{
    font-family: 'Poppins', sans-serif;
}

a{
    text-decoration: none;
    color: white;
}

.header{
    position: relative;
    min-height: 60vh;
```

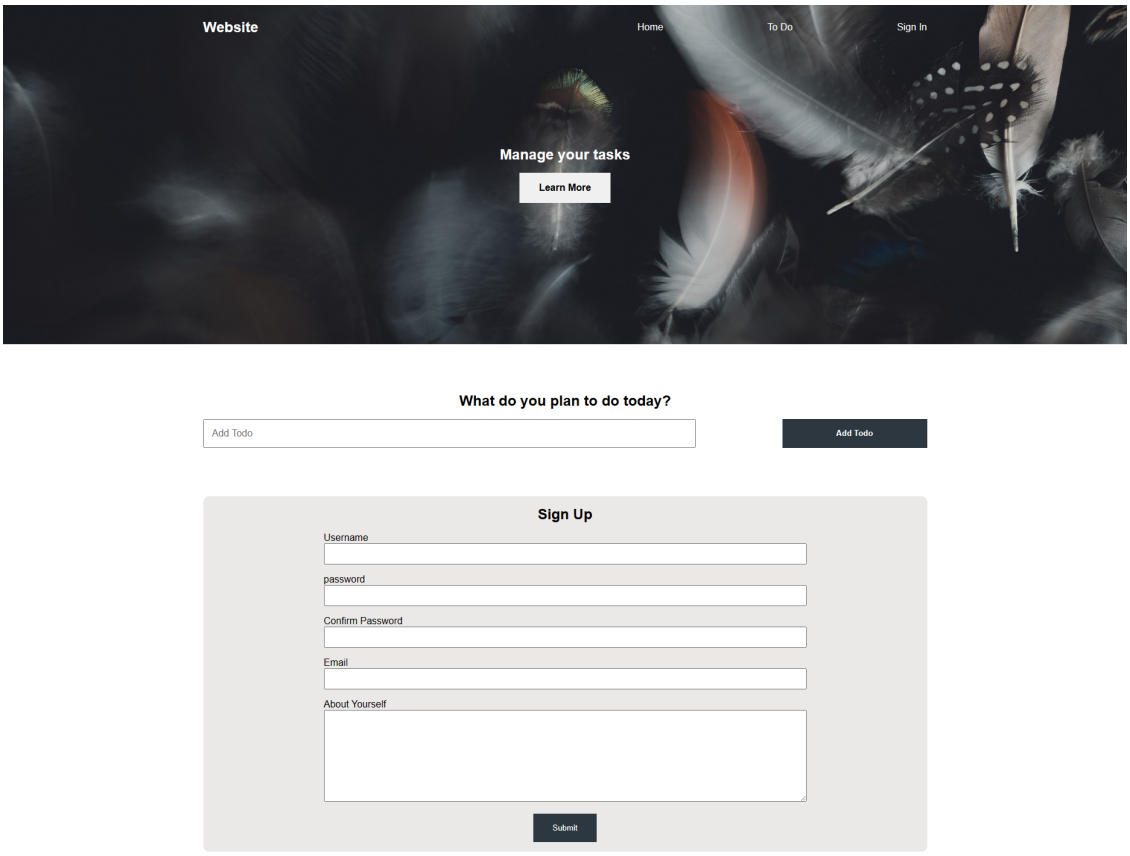
```
    background: url('./background.jpg');
    background-position: center;
    background-repeat: no-repeat;
    background-size: cover;
    color: white;
}

.container{
    max-width: 1200px;
    margin: auto;
    width: 90%;
}

.nav {
    min-height: 8vh;
    display: flex;
    justify-content: space-between;
    align-items: center;
  }

  .nav-items {
    width: 40%;
    display: flex;
    list-style-type: none;
    justify-content: space-between;
  }

  .headline {
    text-align: center;
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    /*
    transform: translate(-50%, -50%);" is used to move
    an element (typically used for centering an element)
    to the center of its parent container both vertically
    and horizontally.

    "transform: translate(-50%, -50%);" moves an element to
    the center of its parent container by moving it 50% of
    its own size to the left and 50% of its own size up. This
    technique is often used for centering elements vertically
    and horizontally in responsive web design, where the size
    of the container may change dynamically.
    */
  }
```

```css
.btn {
  display: inline-block;
  outline: none;
  border: none;
  cursor: pointer;
}

.btn-headline {
  padding: 1rem 2rem;
  font-size: 1rem;
  margin-top: 1rem;
  font-weight: 600;
  /*
  "padding: 1rem 2rem;" sets the amount of space between the content and the
  border of the "btn-headline" element. The "1rem" value specifies the top and
  bottom padding, while the "2rem" value specifies the left and right padding.

  "font-size: 1rem;" sets the size of the text inside the "btn-headline" element
  to 1 rem. The "rem" unit is relative to the font-size of the root element
  (usually the <html> element).

  "margin-top: 1rem;" sets the amount of space between the top edge of the
  "btn-headline" element and the preceding element(s). The "1rem" value specifies
  the amount of margin.

   margin refers to the space outside the border of an element. It is used to add
   space between elements, and affects the position of the element in relation to
   other elements on the page. When you apply margin-top to an element, it creates
   space above the element and pushes it down, which affects the layout of the page.

   the main difference between top and bottom padding vs margin-top is that padding
   affects the space inside the element, while margin affects the space outside the
   element and the position of the element in relation to other elements on the page.

  "font-weight": In CSS, the "font-weight" property is used to set the weight
  (or thickness) of the font. A font's weight is typically expressed as a numerical
  value ranging from 100 to 900, with certain values (e.g. 100, 400, 700) having
  specific names (e.g. "thin", "normal", "bold").

  A "font-weight" value of 500 means that the font is in the middle of the normal
  range, which is typically equivalent to "medium" or "semi-bold". It is thicker
  than "normal" (400) but lighter than "bold" (700). Some fonts may have additional
  weight values (e.g. 600, 800) that fall between these ranges.
   */
}

.section-todo {
  margin-top: 5rem;
```

```
  text-align: center;
}

.form-todo {
  min-height: 5vh;
  display: flex;
  justify-content: space-between;
  margin-top: 1rem;
}

.form-todo input {
  min-height: 100%;
}

.form-todo input[type="text"] {
  width: 68%;
  padding: 0.8rem;
  font-size: 1rem;
  font-weight: 400;
}

.form-todo input[type="submit"] {
  width: 20%;
  background: rgb(44, 55, 63);
  color: white;
  font-weight: bold;
}

.section-signup {
  /*
  The auto value for the left and right margins centers the element horizontally
  within its containing block.

  So, margin: 5rem auto; centers an element horizontally and sets a large top and
  bottom margin of 5 rems each.
  */
  margin: 5rem auto;
  text-align: center;
  background: rgb(235, 232, 232);
  border-radius: 10px;
  padding: 1rem;
}

.signup-form {
  max-width: 800px;
  width: 95%;
  text-align: left;
  margin: auto;
```

```
}

.signup-form label {
  display: block;
}

.signup-form input {
  display: block;
  width: 100%;
  padding: 0.5rem;
}

.form-group {
  margin-top: 1rem;
}

.signup-form textarea {
  width: 100%;
}

.signup-btn {
  background: rgb(44, 55, 63);
  color: white;
  padding: 1rem 2rem;
  display: block;
  margin: auto;
  margin-top: 1rem;
}
```

The HTML page looks like below:



Now, inside 102.js, if the code is only this:

```
console.log(window.document);
```

The output will be the code of this page itself:

```
▼ #document                                                                    102.js:2
    <!DOCTYPE html>
    <html lang="en">
    ▼ <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="./style.css">
        <!--
                We can link our js file here using below:
                <script src="./102.js"></script>
                But this will throw error if the JS is using the HTML elements below
                as the HTML elements used in this file below are not read or loaded yet

            -->
        <!--
                1st approach:
                <script src="./102.js"></script>
            -->
        <!--
                3rd approach:
                <script src="./102.js" async></script>
            -->
        <!--
                4th approach: best approach using defer
                <script src="./102.js" defer></script>
            -->
        <script src="./102.js" defer></script>
        <title>My Website</title>
      </head>
    ▼ <body>
      ▶ <header class="header">…</header>
      ▶ <section class="section-todo container">…</section>
      ▶ <section class="section-signup container">…</section>
        <!--
                2nd approach: Putting before the end of <body> tag
                <script src="./102.js"></script>
            -->
        <!-- Code injected by live-server -->
      ▶ <script>…</script>
      </body>
    </html>
```

In JavaScript, `console.dir()` is a method used to log an object to the console, displaying its properties in a hierarchical format, similar to expanding folders in a file system.

In the code `console.dir(window.document)`, we are using the `console.dir()` method to log the `document` object of the `window` object to the console, displaying its properties in a hierarchical format.

The `window` object is a global object in the browser environment that represents the current browser window or tab, while the `document` property of the `window` object represents the current HTML document that is loaded in the browser window.

By using `console.dir()` to log the document object, we can explore its properties and methods in a more detailed way, since `console.dir()` displays the object in a tree-like structure, allowing us to easily see its nested properties and methods.

This code logs the document object to the console using `console.dir()`, which can be helpful for debugging and understanding the structure of the HTML document and how it can be manipulated through the document object's properties and methods.

```
console.dir(window.document)
```

outputs the following:


```
console.dir(document)
```

this will also give the same output.

Same Id can't be assigned to multiple HTML elements. But classes can be assigned to multiple items.

```
// selects Id
const mainHeading = document.querySelector("#main-heading");
console.log(mainHeading);

// selects class
const header = document.querySelector(".header");
console.log(header);

// if there are multiple elements with the same class,
// it will select the first one only
const navItem = document.querySelector(".nav-item");
console.log(navItem);

// to select all elements with the same class
const navItems = document.querySelectorAll(".nav-item");
console.log(navItems);
```

```
// change text
// textContent and innerText

// print the text inside an element using Id
const mainHeading = document.getElementById("main-heading");
console.log(mainHeading.textContent)

// change the text inside an element using Id
// mainHeading.textContent = "This is changed now";
console.log(mainHeading.textContent);

// innerText will not give text inside span
console.log(mainHeading.innerText)
```

get multiple elements using `getElementsByClassName` – returns HTMLCollection which is array-like object

get multiple elements using `querySelectorAll` – returns NodeList

```
// the below gives HTMLCollection
const navItems = document.getElementsByClassName("nav-item");
console.log(navItems);

console.log(navItems[0]);

console.log(typeof navItems); // it's array-like object

console.log(Array.isArray(navItems)); // false


// returns NodeList
const navItems2 = document.querySelectorAll(".nav-item");
console.log(navItems2[1])
```

HTMLCollection is array-like object – can use index and has length property document.getElementsByTagName returns HTMLCollection or array-like object.

```
const navItems = document.getElementsByTagName("a");
console.log(navItems);
console.log(navItems.length);
```

With `HTMLCollection`, we can use simple for loop, for of loop but not forEach loop.

simple for loop

```
// simple for loop
for (let i = 0; i < navItems.length; i++) {
  console.log(navItems[i]);
}
```

```
for (let i = 0; i < navItems.length; i++) {
    const navItem = navItems[i];
    navItem.style.backgroundColor = "white";
    navItem.style.color = "green";
    navItem.style.fontWeight = "bold";
  }
```

for of loop

```
  // for of loop
  // change background color and color of the link
  for (let navItem of navItems){
    navItem.style.backgroundColor = "blue";
    navItem.style.color = "black";
    navItem.style.fontWeight = "bold";
  }
```

forEach – will not work

```
  // this will not work, will give error
  navItems.forEach((navItem)=>{
    navItem.style.backgroundColor = "blue";
    navItem.style.color = "black";
    navItem.style.fontWeight = "bold";
  })
```

But, we can still work with forEach loop by converting the `HTMLCollection` to `Array` using `Array.from`

```
    // the below works
    const navItemsArray = Array.from(navItems);
    console.log(navItemsArray);
    // Now, we can use forEach
    navItemsArray.forEach((navItem)=>{
        navItem.style.backgroundColor = "azure";
        navItem.style.color = "pink";
        navItem.style.fontWeight = "bold";
      })
```

1.0.9 querySelectorAll

querySelectorAll returns NodeList With this we can use all three loops: 1. simple for loop 2. for of loop 3. forEach loop

```
const navItems3 = document.querySelectorAll("a");

// simple for loop
for (let i = 0; i < navItems3.length; i++) {
    const navItem = navItems3[i];
    navItem.style.backgroundColor = "white";
```

```
    navItem.style.color = "green";
    navItem.style.fontWeight = "bold";
  }

// for of loop
for (let navItem of navItems3){
    navItem.style.backgroundColor = "blue";
    navItem.style.color = "black";
    navItem.style.fontWeight = "bold";
  }

// forEach loop
navItems3.forEach((navItem)=>{
      navItem.style.backgroundColor = "azure";
      navItem.style.color = "pink";
      navItem.style.fontWeight = "bold";
    })
```

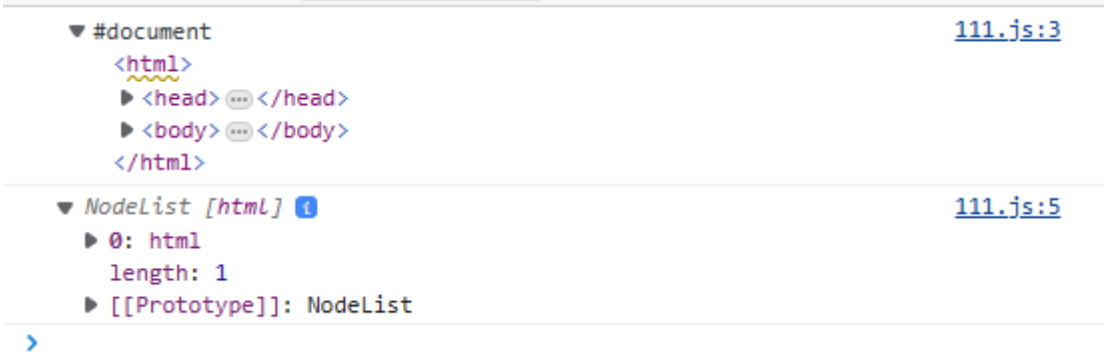
NodeList can also be converted to Array using `Array.from()`

1.0.10 More on DOM

This is the HTMl code we are working with now (`index2.html`):

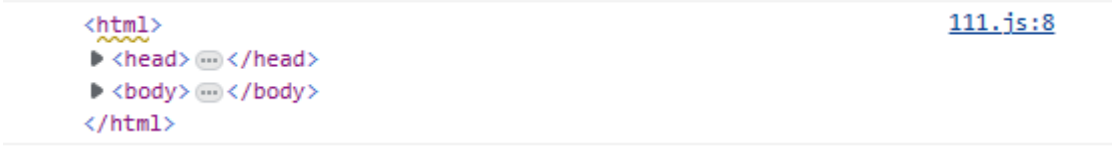
```
<html>
    <head>
        <title>DOM traversing</title>
        <script src="./111.js" defer></script>
    </head>
    <body>
        <div class="container">
            <h1>My heading</h1>
                <p>Some useful information</p>
        </div>
    </body>
</html>
```

```
const rootNode = document.getRootNode();
console.log(rootNode);

console.log(rootNode.childNodes);
```



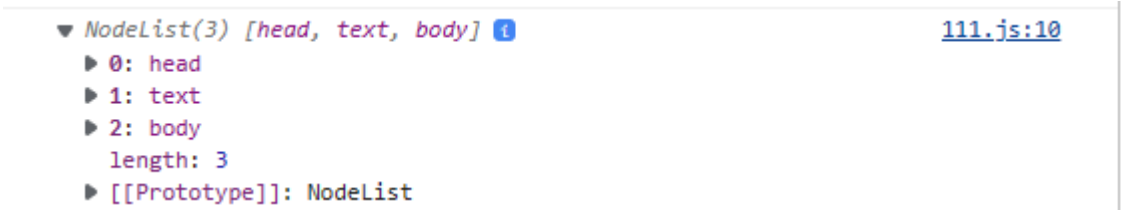
1.0.11 Root element corresponding to <HTML> tag

```
const htmlElementNode = rootNode.childNodes[0];
console.log(htmlElementNode);
```



If we want object representation, we can write `console.dir(htmlElementNode)`.

```
console.log(htmlElementNode.childNodes);
```



1.0.12 Child nodes under root element or HTML element or HTML node

```
const headElementNode = htmlElementNode.childNodes[0];
console.log(headElementNode);

const textNode = htmlElementNode.childNodes[1];
console.log(textNode);

const bodyElementNode = htmlElementNode.childNodes[2];
console.log(bodyElementNode);
```

```
▼ <head>
    <title>DOM traversing</title>
    <script src="./111.js" defer></script>
  </head>
▶ #text
▼ <body>
  ▶ <div class="container">…</div>
    <!-- Code injected by live-server -->
  ▶ <script>…</script>
  </body>
```

1.0.13 Parent node of head element

```
console.log(headElementNode.parentNode);
```

```
<html>                                                    111.js:22
▶ <head>…</head>
▶ <body>…</body>
</html>
```

1.0.14 Sibling relation

```
console.log(headElementNode.nextSibling);
```

```
▶ #text                                                   111.js:25
>
```

1.0.15 Two siblings next to head element

```
// two siblings next to head element
console.log(headElementNode.nextSibling.nextSibling);
```

```
▶ <body>…</body>
>
```

1.0.16 `nextElementSibling` – it will ignore textNode consisting of space and new line

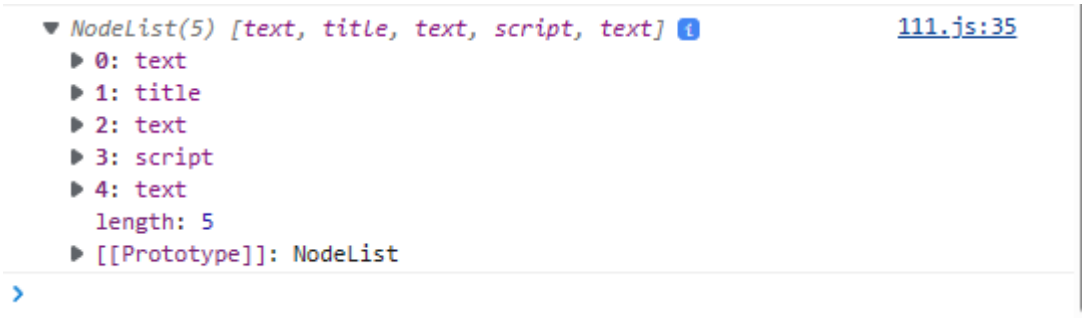
The line below will return body element:

```
console.log(headElementNode.nextElementSibling);
```

```
▶ <body>…</body>                                          111.js:33
```

1.0.17 childNodes

```
console.log(headElementNode.childNodes);
```



1.0.18 Select heading, go to its parent and then change the background color and text color of this parent

```
// Select heading, go to its parent and then change the
// background color and text color of this parent

const h1 = document.querySelector("h1");
const div = h1.parentNode;
div.style.color = "#efefef";
div.style.backgroundColor = "#333";
```

1.0.19 Reach body element from heading and then change the background color and text color of this parent

```
// reach body element from heading and then change the
// background color and text color of this parent
const h1 = document.querySelector("h1");
const body = h1.parentNode.parentNode;
body.style.color = "#efefef";
body.style.backgroundColor = "#333";
```

1.0.20 Direct way to select `body` element using `document.body`

```
// A different way to select body element and do the same tasks
// as before

const body2 = document.body;
body.style.color = "#efefef";
body.style.backgroundColor = "#333";
```

1.0.21 Nesting querySelector – selecting nested HTML element using `querySelector`

We can use querySelector to select an HTML element. We can again use querySelector to select HTML elements within the first selected element.

Selecting `title` element nested inside `head` element In the example below, we select `head` element first using `querySelector` and then select `title` element by again using `querySelector` on the selected `head` element.

```
const head = document.querySelector("head");
const title = head.querySelector("title");
console.log(title);
```

1.0.22 childNodes property gives child nodes including text nodes

```
// .childNodes property includes text nodes
console.log(container.childNodes); // includes text nodes
// representing new line and space
```

```
▶ NodeList(5) [text, h1, text, p, text]                    111.js:73
```

1.0.23 children property gives child nodes without text nodes

```
// .children property does not include text nodes
console.log(container.children); // does not include text nodes
// representing new line and space
```

```
▶ HTMLCollection(2) [h1, p]                                111.js:78
```

Now, again we will work with `index.html` file defined as following:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="./style.css">
    <!--
        We can link our js file here using below:
        <script src="./102.js"></script>
        But this will throw error if the JS is using the HTML elements below
        as the HTML elements used in this file below are not read or loaded yet

    -->
    <!--
        1st approach:
        <script src="./102.js"></script>
    -->
    <!--
        3rd approach:
        <script src="./102.js" async></script>
    -->

    <!--
        4th approach: best approach using defer
        <script src="./102.js" defer></script>
    -->
```

```
    <script src="./118.js" defer></script>

    <title>My Website</title>
</head>
<body>
    <header class="header">
        <nav class="nav container">
            <h1 class="log">Website</h1>
            <ul class="nav-items">
                <li class="nav-item"><a href="Home">Home</a></li>
                <li class="nav-item"><a href="">To Do</a></li>
                <li class="nav-item"><a href="">Sign In</a></li>
            </ul>
        </nav>
        <div class="headline">
            <h2 id="main-heading">Manage your tasks <span style="display: none">Hello</span></h
            <button class="btn btn-headline">Learn More</button>
        </div>
    </header>
    <section class="section-todo container">
        <h2>What do you plan to do today?</h2>
        <form class="form-todo">
            <input type="text" name="" id="" placeholder="Add Todo">
            <input type="submit" value="Add Todo" class="btn">
        </form>
        <ul class="todo-list">
            <!-- Uncomment the below line upto file 117.js -->                    //
            <!--<li class="first-todo">todo 1</li> -->
            <li>item 1</li>
            <li>item 2</li>
            <li>item 3</li>
            <li>item 4</li>
            <li>item 5</li>
        </ul>
    </section>

    <section class="section-signup container">
        <h2>Sign Up</h2>
        <form class="signup-form">
            <div class="form-group">
                <label for="username">Username</label>
                <input type="text" name="username" id="username">
            </div>
            <div class="form-group">
                <label for="password">password</label>
                <input type="password" name="password" id="password">
            </div>
            <div class="form-group">
```

```
                <label for="confirmPassword">Confirm Password</label>
                <input type="password" name="confirmPassword" id="confirmPassword">
            </div>
            <div class="form-group">
                <label for="email">Email</label>
                <input type="email" name="email" id="email">
            </div>
            <div class="form-group">
                <label for="about">About Yourself</label>
                <textarea name="about" id="about" cols="30" rows="10"></textarea>
            </div>
            <button type="submit" class="btn signup-btn">Submit</button>

        </form>
    </section>
    <!--
        2nd approach: Putting before the end of <body> tag
        <script src="./102.js"></script>
    -->
</body>
</html>
```

We also have `style.css` as before:

```
@import url('https://fonts.googleapis.com/css2?family=Poppins:ital,wght@0,300;0,400;0,500;600;

*{
    box-sizing: border-box;
    padding: 0;
    margin: 0;
}

body{
    font-family: 'Poppins', sans-serif;
}

a{
    text-decoration: none;
    color: white;
}

.header{
    position: relative;
    min-height: 60vh;
    background: url('./background.jpg');
    background-position: center;
    background-repeat: no-repeat;
    background-size: cover;
    color: white;
```

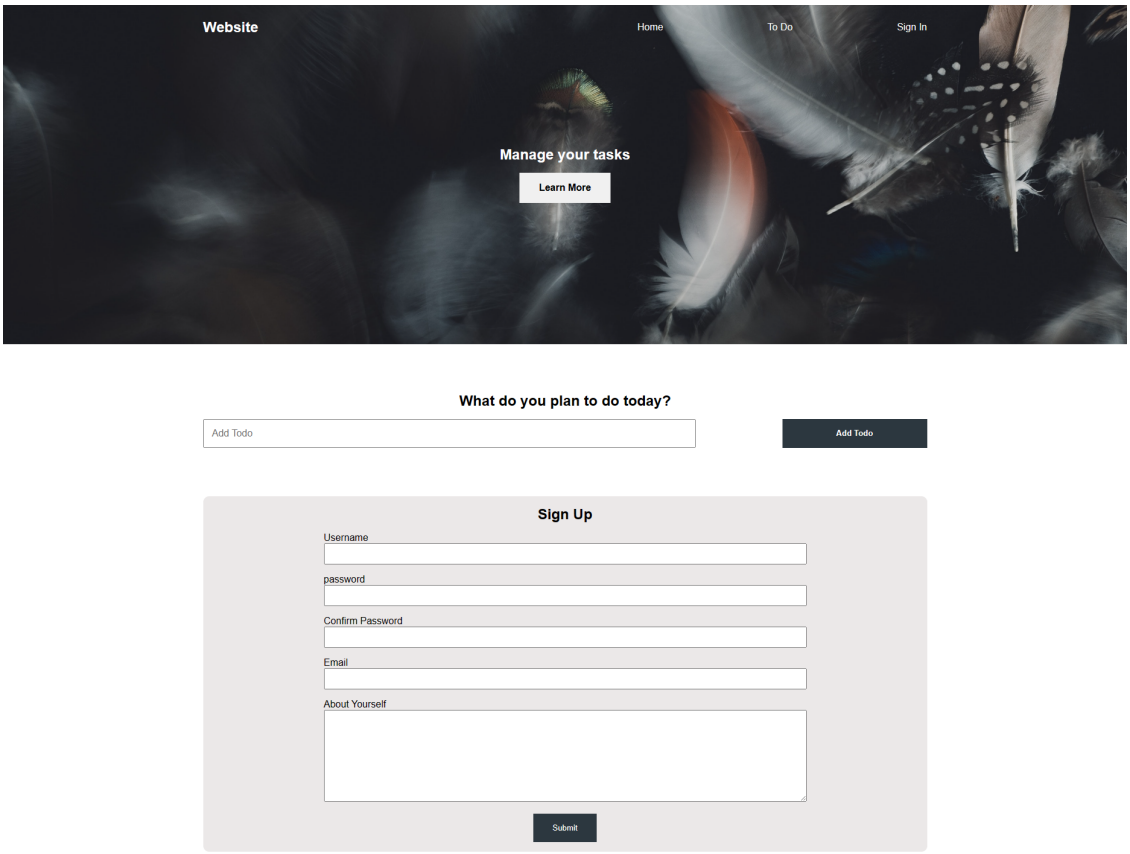
```
}

.container{
    max-width: 1200px;
    margin: auto;
    width: 90%;
}

.nav {
    min-height: 8vh;
    display: flex;
    justify-content: space-between;
    align-items: center;
  }

  .nav-items {
    width: 40%;
    display: flex;
    list-style-type: none;
    justify-content: space-between;
  }

  .headline {
    text-align: center;
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    /*
    transform: translate(-50%, -50%);" is used to move
    an element (typically used for centering an element)
    to the center of its parent container both vertically
    and horizontally.

    "transform: translate(-50%, -50%);" moves an element to
    the center of its parent container by moving it 50% of
    its own size to the left and 50% of its own size up. This
    technique is often used for centering elements vertically
    and horizontally in responsive web design, where the size
    of the container may change dynamically.
    */
  }

  .btn {
    display: inline-block;
    outline: none;
    border: none;
    cursor: pointer;
```

```
}

.btn-headline {
  padding: 1rem 2rem;
  font-size: 1rem;
  margin-top: 1rem;
  font-weight: 600;
  /*
  "padding: 1rem 2rem;" sets the amount of space between the content and the
  border of the "btn-headline" element. The "1rem" value specifies the top and
  bottom padding, while the "2rem" value specifies the left and right padding.

  "font-size: 1rem;" sets the size of the text inside the "btn-headline" element
  to 1 rem. The "rem" unit is relative to the font-size of the root element
  (usually the <html> element).

  "margin-top: 1rem;" sets the amount of space between the top edge of the
  "btn-headline" element and the preceding element(s). The "1rem" value specifies
  the amount of margin.

   margin refers to the space outside the border of an element. It is used to add
   space between elements, and affects the position of the element in relation to
   other elements on the page. When you apply margin-top to an element, it creates
   space above the element and pushes it down, which affects the layout of the page.

   the main difference between top and bottom padding vs margin-top is that padding
   affects the space inside the element, while margin affects the space outside the
   element and the position of the element in relation to other elements on the page.

  "font-weight": In CSS, the "font-weight" property is used to set the weight
  (or thickness) of the font. A font's weight is typically expressed as a numerical
  value ranging from 100 to 900, with certain values (e.g. 100, 400, 700) having
  specific names (e.g. "thin", "normal", "bold").

  A "font-weight" value of 500 means that the font is in the middle of the normal
  range, which is typically equivalent to "medium" or "semi-bold". It is thicker
  than "normal" (400) but lighter than "bold" (700). Some fonts may have additional
  weight values (e.g. 600, 800) that fall between these ranges.
   */
}

.section-todo {
  margin-top: 5rem;
  text-align: center;
}

 .bg-dark {
    background: #000;
```

```
    color: #eee;
 }

 .form-todo {
  min-height: 5vh;
  display: flex;
  justify-content: space-between;
  margin-top: 1rem;
}

.form-todo input {
  min-height: 100%;
}

.form-todo input[type="text"] {
  width: 68%;
  padding: 0.8rem;
  font-size: 1rem;
  font-weight: 400;
}

.form-todo input[type="submit"] {
  width: 20%;
  background: rgb(44, 55, 63);
  color: white;
  font-weight: bold;
}

.section-signup {
  /*
  The auto value for the left and right margins centers the element horizontally
  within its containing block.

  So, margin: 5rem auto; centers an element horizontally and sets a large top and
  bottom margin of 5 rems each.
  */
  margin: 5rem auto;
  text-align: center;
  background: rgb(235, 232, 232);
  border-radius: 10px;
  padding: 1rem;
}

.signup-form {
  max-width: 800px;
  width: 95%;
  text-align: left;
  margin: auto;
```

```
}

.signup-form label {
  display: block;
}

.signup-form input {
  display: block;
  width: 100%;
  padding: 0.5rem;
}

.form-group {
  margin-top: 1rem;
}

.signup-form textarea {
  width: 100%;
}

.signup-btn {
  background: rgb(44, 55, 63);
  color: white;
  padding: 1rem 2rem;
  display: block;
  margin: auto;
  margin-top: 1rem;
}
```

To refresh the memory, the HTML page looks like below:



1.0.24 Listing class

```
const sectionToDo = document.querySelector('.section-todo');

// listing class
console.log(sectionToDo.classList);
```

```
▼ DOMTokenList(2) i                                    112.js:4
    0: "section-todo"
    1: "container"
    length: 2
    value: "section-todo container"
  ▶ [[Prototype]]: DOMTokenList
```

1.0.25 Adding class

```
sectionToDo.classList.add('bg-dark');
console.log(sectionToDo.classList);
```

```
                                                       112.js:8
▶ DOMTokenList(3) ['section-todo', 'container', 'bg-dark', value: 'section-to
  do container bg-dark']
```

1.0.26 Removing class

```
sectionToDo.classList.remove("container");
console.log(sectionToDo.classList);
```

```
                                                                    112.js:12
  ▶ DOMTokenList(2) ['section-todo', 'bg-dark', value: 'section-todo bg-dark']
>
```

1.0.27 Checking the existence of a class

```
console.log(sectionToDo.classList.contains('container'));
```

```
  false                                                             112.js:15
```

1.0.28 Toggle class

If a class is already there, remove it; if not, add it In the example, we are working with `bg-dark` class defined in CSS

```
sectionToDo.classList.toggle('bg-dark');
```

Some more code on the already covered topics.

```
const header = document.querySelector('.header');
console.log(header.classList);
header.classList.add("bg-dark");
```

1.0.29 Emnet abbreviation

Writing `ul.todo-list>li{todo 1}*1` in VSCode and then pressing Tab will give the following HTML code:

```
<ul class="todo-list">
     <li>todo 1</li>
</ul>
```

1.1 Add new HTML elements to page

1.1.1 Add HTML element using innerHTML

We want to select the class in this section:

```
    <ul class="todo-list">
        <li>todo 1</li>
    </ul>
```

The JS code to select the innerHTML inside the class `todo-list`

```
const todoList = document.querySelector(".todo-list");
console.log(todoList.innerHTML);
```

```
                                                                    113.js:6
            <li>todo 1</li>
```

1.1.2 Change the innerHTML – if we are not adding new element, we can use it

```
todoList.innerHTML = "<li>New Todo</li>"
console.log(todoList.innerHTML);
```

```
   <li>New Todo</li>                                                113.js:10
```

1.1.3 Add to the existing HTML – don't use this approach

```
todoList.innerHTML += "<li>New Todo</li>";
console.log(todoList.innerHTML);
```

```
                                                                    113.js:14
            <li>todo 1</li>
          <li>New Todo</li>
```

```
// Add to the existing HTML
todoList.innerHTML += "<li>New Todo</li>";
todoList.innerHTML += "<li>Teaching ML</li>";
todoList.innerHTML += "<li>Learning DS and Algo</li>";
console.log(todoList.innerHTML);
```

```
                                                                    113.js:16
            <li>todo 1</li>
          <li>New Todo</li><li>Teaching ML</li><li>Learning DS and Algo</li>
```

But never do what we have shown above. It gives rise to performance issues. Only use when we need to change all the code inside innerHTML, then use it. Don't use for adding new elements.

To create new element, a better approach is using `createElement`

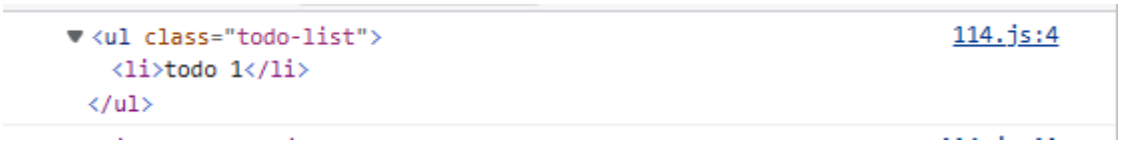
1.1.4 Create HTML element using `document.createElement`

Some of the steps are:

- document.createElement
- append
- prepend
- remove

Adding new list item Let's have a look at the list items before appending anything new.

```
// todo-list before appending
const todoList = document.querySelector(".todo-list");
console.log(todoList);
```



Now, ;et's create new list element, create a new text node, add text to it and then add this to the list item.

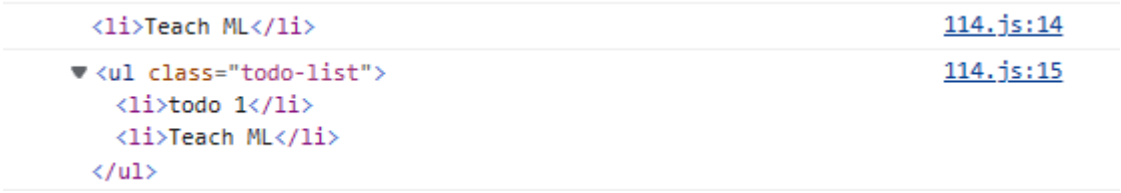
```
const newTodoItem = document.createElement("li");
const newTodoItemNext = document.createTextNode("Teach ML");
newTodoItem.append(newTodoItemNext);
```

Now, add this new list item to the todo list item:

```
const newTodoItem = document.createElement("li");
const newTodoItemNext = document.createTextNode("Teach ML");
newTodoItem.append(newTodoItemNext); // Add text node to the new list node
todoList.append(newTodoItem); // add new list to the todo list
```

Now, have a look at the newly created list and the updated todo list:

```
console.log(newTodoItem);
console.log(todoList);
```



1.1.5 A more efficient way to add list element – `textContent`

Using `textContent`, we don't even need to create a new text node (using `createTextNode`), and then add it to the newly created list node; rather, we can add text to the newly created list node using `textContent` property of this newly created list.

```
// Adding text to new list item in a more efficient way
const todoList = document.querySelector(".todo-list");
const newTodoItem = document.createElement("li");
newTodoItem.textContent = "Teach ML";
todoList.append(newTodoItem);
// we can also use appendChild() instead of append()
console.log(todoList);
```

Now, we also look at `prepend` which adds at the beginning:

```
// Adding text to new list item in a more efficient way
const todoList = document.querySelector(".todo-list");
```

```
const newTodoItem = document.createElement("li");
newTodoItem.textContent = "Teach ML";
todoList.prepend(newTodoItem);
console.log(todoList);
```

```
▼ <ul class="todo-list">                                    114 3.js:6
    <li>Teach ML</li>
    <li>todo 1</li>
  </ul>
```

1.1.6 Removing a list item

```
// selects first list item under todo-list class
const todo1 = document.querySelector(".todo-list li");
todo1.remove();
console.log(todo1);
```

1.1.7 Inserting before and after an element

Inserting before using `before()`

```
// Inserting before
const newTodoItem = document.createElement("li");
newTodoItem.textContent = "Teach ML";
const todoList = document.querySelector(".todo-list");
todoList.before(newTodoItem);
```

Inserting after using `after()`

```
// Inserting after
const newTodoItem = document.createElement("li");
newTodoItem.textContent = "Teach ML";
const todoList = document.querySelector(".todo-list");
todoList.after(newTodoItem);
```

1.1.8 An alternative way to insert element using `insertAdjacentHTML`

- beforebegin
- afterbegin
- beforeend
- afterend

elem.insertAdjacentHTML(where, html) Insert before the end of the element with class `todo-list`

```
// Add one more list item under unordered list
const todoList = document.querySelector(".todo-list");
console.log(todoList);
```

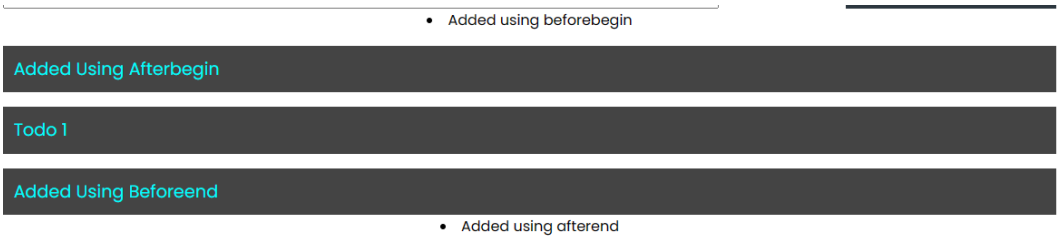
```
todoList.insertAdjacentHTML("beforeend", "<li>Teach ML</li>");
console.log(todoList);
```

Similarly, `afterbegin` can used for prepending

```
todoList.insertAdjacentHTML("afterbegin", "<li>Added using afterbegin</li>");
console.log(todoList);
```

Similarly, we can use `beforrebegin` and `afterend`

```
todoList.insertAdjacentHTML("beforebegin", "<li>Added using beforebegin</li>");
console.log(todoList);

// It will be added outside the unordered list or class with the value todo-list
todoList.insertAdjacentHTML("afterend", "<li>Added using afterend</li>");
console.log(todoList);
```



1.1.9 Clone nodes

Suppose we want to both append and prepend a new item to the list, we can clone and then append/prepend the cloned node

```
const ul = document.querySelector(".todo-list");
const li = document.createElement("li");
li.textContent = "Teach ML";
const li2 = li.cloneNode(true); // true means deep clone
ul.prepend(li);
ul.append(li2);
```

1.1.10 Some old methods to support poor IE

- `appendChild` (instead of `append`)

```
const ul = document.querySelector(".todo-list");
const li = document.createElement("li");
li.textContent = "New todo";
ul.appendChild(li);
```

- `insertBefore` (instead of `prepend`)

```
// use of insertBefore
const ul = document.querySelector(".todo-list");
const li = document.createElement("li");
const referenceNode = document.querySelector(".first-node");
```

```
li.textContent = "Added with insertBefore";
ul.insertBefore(li, referenceNode);
```

- replaceChild;

```
const ul = document.querySelector(".todo-list");
const li = document.createElement("li");
li.textContent = "Added with insertBefore";
const referenceNode = document.querySelector(".first-node");
ul.replaceChild(li, referenceNode);
```

- removeChild (instead of remove)

```
const ul = document.querySelector(".todo-list");
const referenceNode = document.querySelector(".first-node");
ul.removeChild(referenceNode);
```

1.1.11 More Emnet shortcut

`li{item $}*5` will give

```
<li>item 1</li>
<li>item 2</li>
<li>item 3</li>
<li>item 4</li>
<li>item 5</li>
```

1.1.12 static list vs live list

`querySewlectorAll` gives us static list, gives us NodeList

`getElementBySomething` gives us live list, gives us HTML Element

static list

```
const listItems = document.querySelectorAll(".todo-list li");
const sixthListItem = document.createElement("li");
sixthListItem.textContent = "Sixth todo -- added with static list";
const ul = document.querySelector(".todo-list");
ul.append(sixthListItem);
console.log(listItems); // this will not include the new item
// although the new item is added to the DOM, the static
// list is not updated
```

live list

```
const ul = document.querySelectorAll(".todo-list");
const listItems = ul.getElementsByTagName("li");

const sixthListItem = document.createElement("li");
sixthListItem.textContent = "Sixth todo -- added with live list";
const ul = document.querySelector(".todo-list");
```

```
ul.append(sixthListItem);
console.log(listItems); // this will include the new item
```

1.1.13 Dimensions of an element

getBoundingClientRect()

```
const sectionTodo = document.querySelector('.section-todo');
const info = sectionTodo.getBoundingClientRect();
console.log(info);
```

```
                                                                    119.js:8
▼ DOMRect {x: 64.453125, y: 365.09375, width: 1160.09375, height: 415.46875,
  top: 365.09375, …}
    bottom: 780.5625
    height: 415.46875
    left: 64.453125
    right: 1224.546875
    top: 365.09375
    width: 1160.09375
    x: 64.453125
    y: 365.09375
  ▶ [[Prototype]]: DOMRect
```

```
const height = sectionTodo.getBoundingClientRect().height;
console.log(height);
```

```
443.46875                                                           119.js:11
```

```
const width = sectionTodo.getBoundingClientRect().width;
console.log(width);
```

```
217.796875                                                          119.js:14
```

```
const top = sectionTodo.getBoundingClientRect().top;
console.log(top);
```

```
96.5                                                                119.js:17
```

1.1.14 Events

In JavaScript, an event is a signal that is generated by the browser or by the user's interaction with a web page. Events can be triggered by various actions, such as clicking a button, submitting a form, moving the mouse, or typing on the keyboard.

When an event occurs, the browser creates an event object that contains information about the event, such as its type, target element, and any additional data related to the event.

You can use JavaScript to listen for events and respond to them in different ways. This is often done by attaching event listeners to elements in the web page using the `addEventListener` method. Here's an example:

```
const button = document.querySelector('#my-button');

button.addEventListener('click', function(event) {
  // do something when the button is clicked
});
```

In this example, an event listener is attached to a button element with the ID `my-button`. The listener responds to the `click` event, which is triggered when the button is clicked. When the event occurs, the function passed as the second argument to `addEventListener` is called, and the event object is passed as an argument to the function.

Overall, events in JavaScript allow you to create dynamic and interactive web pages by responding to user interactions and other actions in the browser.

There are 3 ways to bind events to elements

1. Using onclick attribute inside the element
2. Select the element and specify what to do when a specific event happens
3. Select the element and use addEventListener method (more efficient and flexible)

1.1.15 1. using onclick attribute inside the HTML element inside the HTML file

```
<button class="btn btn-headline" onclick="console.log('You clicked me!')">Learn More</button>
```

1.1.16 2. select the element and specify what to do when the event happens

```
const btn = document.querySelector(".btn-headline");
btn.onclick = function(){
    console.log("You clicked me!");
}
```

1.1.17 3. Select the element and use/add `addEventListener` method

Below all three approaches give the same result.

Approach 1

```
const btn = document.querySelector(".btn-headline");
function clickMe(){
    console.log("You clicked me!");
}
btn.addEventListener("click", clickMe);
```

Approach 2

```
const btn = document.querySelector(".btn-headline");
btn.addEventListener("click", function(){
    console.log("You clicked me!");
})
```

Approach 3

```
const btn = document.querySelector(".btn-headline");
btn.addEventListener("click", ()=>{
    console.log("Arrow function -- You clicked me!");
});
```

1.1.18 this keyword

```
const btn = document.querySelector(".btn-headline");
btn.addEventListener("click", function(){
    console.log("you clicked me");
    console.log(this);
    this.style.backgroundColor = "red";
})
```

```
you clicked me                                                    121.js:6
                                                                  121.js:7
    <button class="btn btn-headline" style="background-color: red;">Learn More
    </button>
>
```

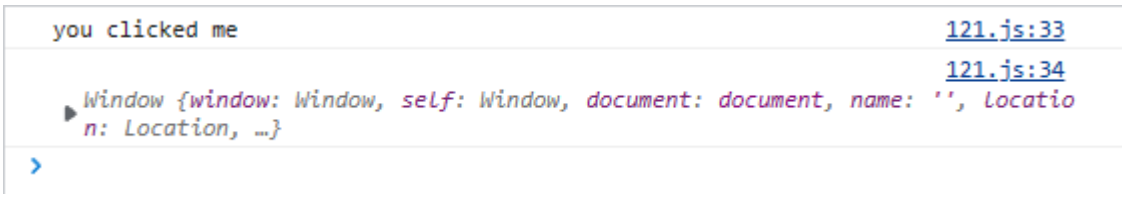
Even if we declare function outside of the `addEventListener` method, the this keyword will still refer to the element that we are listening to the event on.

```
const btn = document.querySelector(".btn-headline");
function clickMe(){
    console.log("you clicked me");
    console.log(this);
    this.style.backgroundColor = "red";
}
btn.addEventListener("click", clickMe);
```

```
you clicked me                                                    121.js:6
                                                                  121.js:7
    <button class="btn btn-headline" style="background-color: red;">Learn More
    </button>
>
```

In case of arrow function, the value of this keyword will be window or will be the value of this keyword in the parent scope

```
const btn = document.querySelector(".btn-headline");
btn.addEventListener("click",()=>{
    console.log("you clicked me");
    console.log(this);
    //this.style.backgroundColor = "red";
})
```



1.1.19 More Emnet shortcuts

! + Tab **! + Tab** gives the following HTML template code

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
</body>
</html>
```

script:src + Tab **script:src + Tab** gives the following `script` tag:

```
<script src=""></script>
```

.className + Tab **.className + Tab** gives the following `div` with the class `className`

For example, **.mybuttons + Tab** gives the following `div`

```
<div class="mybuttons">
```

button*3 + Tab **button*3 + Tab** gives 3 `button` elements

```
<button></button><button></button><button></button>
```

h1.heading$*7{Hello World}** ** h1.heading$*7{Hello World} will give 7 h1 with class heading and text Hello World

```
<h1 class="heading1">Hello World</h1>
<h1 class="heading2">Hello World</h1>
<h1 class="heading3">Hello World</h1>
<h1 class="heading4">Hello World</h1>
<h1 class="heading5">Hello World</h1>
<h1 class="heading6">Hello World</h1>
<h1 class="heading7">Hello World</h1>
```

1.1.20 Adding click event to multiple elements (all the buttons)

I have the following HTML file and trying to incoporate click events to all the buttons

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="./122.js" defer></script>
    <title>Click event</title>
</head>
<body>
    <div class="my-buttons">
        <button id="one">My button one</button>
        <button id="two">My button two</button>
        <button id="three">My button three</button>
    </div>
</body>
</html>
```

Now, in the **122.js** file, we can write the loops in different ways to add click events to multiple elements

1. **Using simple for loop**

```javascript
const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

for(let i=0; i<allButtons.length; i++){
    allButtons[i].addEventListener("click", function(){
        console.log("You clicked me!");
        console.log(this); // will show the button element
    })
}
```

Clicking on the first button will show this:

```
You clicked me!                                              122.js:12
    <button id="one">My button one</button>                  122.js:13
```

Clicking on the second button will show this:

```
You clicked me!                                              122.js:12
    <button id="two">My button two</button>                  122.js:13
```

Clicking on the third button will show this:

```
You clicked me!                                              122.js:12
    <button id="three">My button three</button>              122.js:13
>
```

We can also see the text content of these different buttons:

```
for(let i=0; i<allButtons.length; i++){
    allButtons[i].addEventListener("click", function(){
        console.log(this.textContent);  // will show the button text
    })
}
```

Clicking on the second button will now show the text content of this button:

```
My button two                                                    122.js:21
>
```

Note: Don't use arrow function with the above code as this will refer to the window object and not the button element. But also remember that most of the time we will be using arrow functions. We will see how to get the button element using arrow functions later.

But how do I use arrow function then – use `event.target` If we just use arrow function here, `this` will give the window object. Using event object, we can get the same result. Event object is discussed later.

```
const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

for(let i=0; i<allButtons.length; i++){
    allButtons[i].addEventListener("click", (e)=>{
        console.log("You clicked me!");
        console.log(e.target.textContent); // will show the button element
    })
}
```

Now, clicking on the first button will show this:

```
You clicked me!                                                  122.js:12
    <button id="one">My button one</button>                      122.js:13
```

2. **Using for of loop**

```
// Using for of loop

const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

for(let button of allButtons){
    button.addEventListener("click", function(){
        //console.log("You clicked me!");
        //console.log(this);
        console.log(this.textContent);
    })
}
```

Similar to before, if we click on the third button, we will see the following in the Console:

```
My button three                                    122.js:43
>
```

Using arrow function – `event.target`

```
const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

for(let button of allButtons){
    button.addEventListener("click", (e)=>{
        //console.log("You clicked me!");
        //console.log(this);
        console.log(e.target.textContent);
    })
}
```

Similar to before, if we click on the third button, we will see the following in the Console:

```
My button three                                    122.js:43
>
```

3. **Using forEach method**

```
// using forEach method
const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

allButtons.forEach(function(button){
    button.addEventListener("click", function(){
        // console.log("You clicked me!");
        // console.log(this);
        console.log(this.textContent);
    })
});
```

Similar to before, if we click on the third button, we will again see the following in the Console:

```
My button three                                    122.js:57
>
```

Using arrow function – `event.target`

```
const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

allButtons.forEach(function(button){
    button.addEventListener("click", (e)=>{
        // console.log("You clicked me!");
        // console.log(this);
        console.log(e.target.textContent);
```

```
    })
});
```

If clicked on button three

1.1.21 Event object

Whenever browser encounters an event it is listening to 1. Gives callback function to JS Engine 2. With the callback function, it also gives information on the event happened or an object called event object

```
const firstButton = document.querySelector("#one");

firstButton.addEventListener("click", function(abc){
    console.log(abc); // will show the event object
});
```

Now, if we click on button one with id “one”, we get the following in the Console:

2 important proiperties of event:

1. **target** Which element triggered the event

```
▼ target: button#one
    accessKey: ""
    ariaAtomic: null
    ariaAutoComplete: null
    ariaBrailleLabel: null
    ariaBrailleRoleDescription: null
    ariaBusy: null
    ariaChecked: null
    ariaColCount: null
    ariaColIndex: null
    ariaColSpan: null
    ariaCurrent: null
    ariaDescription: null
    ariaDisabled: null
    ariaExpanded: null
    ariaHasPopup: null
    ariaHidden: null
    ariaInvalid: null
    ariaKeyShortcuts: null
    ariaLabel: null
    ariaLevel: null
    ariaLive: null
    ariaModal: null
    ariaMultiLine: null
    ariaMultiSelectable: null
    ariaOrientation: null
    ariaPlaceholder: null
    ariaPosInSet: null
    ariaPressed: null
    ariaReadOnly: null
    ariaRelevant: null
    ariaRequired: null
    ariaRoleDescription: null
    ariaRowCount: null
    ariaRowIndex: null
    ariaRowSpan: null
    ariaSelected: null
    ariaSetSize: null
    ariaSort: null
    ariaValueMax: null
    ariaValueMin: null
    ariaValueNow: null
    ariaValueText: null
```

2. **currentTarget** to which element we attached the event listener

```
    currentTarget: null
```

Now using `event.target` in the arrow function will address the issue of not being able to use for loop to add key events to buttons.

```
const firstButton = document.querySelector("#one");

firstButton.addEventListener("click", function(abc){
```

```
    console.log(abc.target); // will show the event object
});
```

Clicking on the first button gives:

```
    <button id="one">My button one</button>                    123.js:26
```

The following code will attach click event to all the buttons

```
const allButtons = document.querySelectorAll(".my-buttons button");
// console.log(allButtons);

for(let button of allButtons){
    button.addEventListener("click", (e)=>{
        console.log(e.currentTarget);
    })
}
```

Now, if we click on the three buttons:

```
    <button id="one">My button one</button>                    123.js:46
    <button id="two">My button two</button>                    123.js:46
    <button id="three">My button three</button>                123.js:46
 >
```

For the code example we used, it doesn't matter whether we use `event.target` or `event.currentTarget`. But, later we will see where they actually differ.

JS Engine of **Chrome** is **v8**

JS Engine of Firefox is **Spider Monkey**

1.1.22 Event loop, callback queue

The event loop and callback queue are two important concepts in JavaScript that help manage the execution of asynchronous code.

In JavaScript, the event loop is a mechanism that ensures the code runs in a non-blocking way. The event loop continuously checks for tasks in the callback queue and executes them one by one in a sequential order. The event loop keeps running until there are no more tasks in the callback queue.

When an asynchronous operation is started in JavaScript, it is placed in the callback queue when it completes, along with any associated callback functions. These callbacks are executed in the order they are received in the callback queue.

Here's an example to illustrate the event loop and callback queue:

[2]:
```
console.log('Start');
setTimeout(() => {
  console.log('Middle');
```

```
}, 1000);
console.log('End');
```

```
Start
End
Middle
```

In this example, we have a `setTimeout` function that takes a callback function as its first argument and a time delay of 1000ms as its second argument. When the `setTimeout` function is called, it starts an asynchronous operation that waits for the specified time and then adds the callback function to the callback queue.

As you can see, the console.log statements are executed in the order they appear in the code. However, the callback function added to the callback queue by the `setTimeout` function is executed after the main code has finished executing.

In summary, the event loop and callback queue are essential for managing the execution of asynchronous code in JavaScript. The event loop continuously checks for tasks in the callback queue and executes them one by one in a sequential order, ensuring that the code runs in a non-blocking way.

1.1.23 Event bubbling, capturing

Event bubbling and capturing are two mechanisms in JavaScript that describe the order in which event handlers are executed when an event occurs on an element with nested child elements.

Event capturing is the first phase of event propagation, where the event is captured by the outermost element first and then propagated inward to the target element. This means that if you have a nested structure of HTML elements, such as a div inside another div, and an event occurs on the inner div, the event will first be captured by the outer div before it is handled by the inner div. The capturing phase occurs before the bubbling phase.

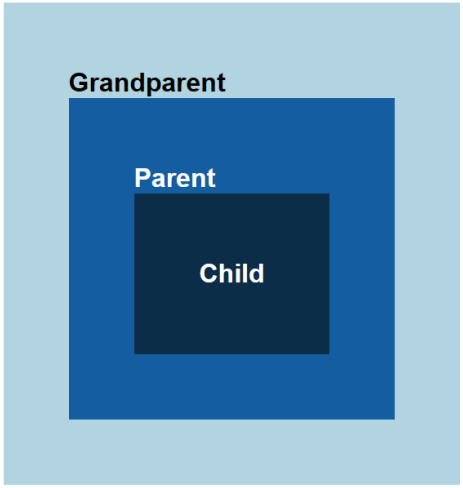
Event bubbling is the second phase of event propagation, where the event is handled by the target element first and then propagated outward to the outermost element. This means that if you have a nested structure of HTML elements and an event occurs on the innermost element, the event will first be handled by the innermost element and then propagated up to the outermost element. The bubbling phase occurs after the capturing phase.

Event bubbling is the default mechanism in JavaScript, and it can be useful for handling events that are triggered by child elements. However, in some cases, you may want to use event capturing instead. To use event capturing, you can set the third parameter of the `addEventListener` method to true.

We have the following HTMl file:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="./evt-bcd.css">
```

```
    <script src="./128.js" defer></script>
    <title>Event Bubbling | Capturing | delegation</title>
</head>
<body>
    <main>
        <div class="grandparent box">
            Grandparent
            <div class="parent box">
                Parent
                <div class="child box">Child</div>
            </div>
        </div>
    </main>
</body>
</html>
```



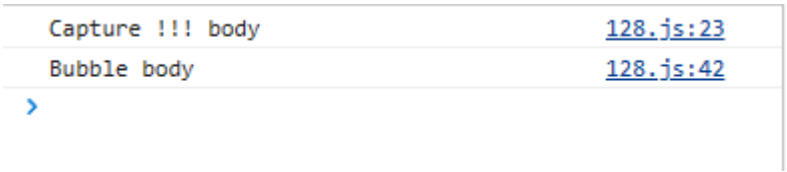
The page looks like below:

The JavaScript code we are using is in **128.js**.

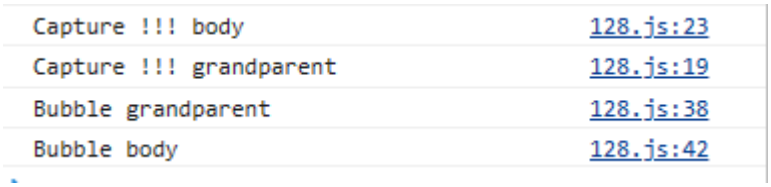
```
// console.log("Hello world!");

const grandparent = document.querySelector(".grandparent");
const parent = document.querySelector(".parent");
const child = document.querySelector(".child");

// capturing events
// from the top to the bottom
child.addEventListener("click", ()=> {
    console.log("Capture !!! child");
```

```
}, true);

parent.addEventListener("click", ()=> {
    console.log("Capture !!! parent");
}, true);

grandparent.addEventListener("click", ()=> {
    console.log("Capture !!! grandparent");
}, true);

document.body.addEventListener("click", ()=> {
    console.log("Capture !!! body");
}, true);


// not capture -- bubbling
// from the bottom to the top
child.addEventListener("click", ()=> {
    console.log("Bubble child");
});

parent.addEventListener("click", ()=> {
    console.log("Bubble parent");
});

grandparent.addEventListener("click", ()=> {
    console.log("Bubble grandparent");
});

document.body.addEventListener("click", ()=> {
    console.log("Bubble body");
});
```

Now, if we click outside the `grandparent` portion inside the body, we get the following output:



Now, if we click inside the `grandparent` portion, we get the following output:



Now, if we click inside the `parent` portion, we get the following output:

```
Capture !!! body                    128.js:23
Capture !!! grandparent             128.js:19
Capture !!! parent                  128.js:15
Bubble parent                       128.js:34
Bubble grandparent                  128.js:38
Bubble body                         128.js:42
```

Now, if we click inside the `child` portion, we get the following output:

```
Capture !!! body                    128.js:23
Capture !!! grandparent             128.js:19
Capture !!! parent                  128.js:15
Capture !!! child                   128.js:11
Bubble child                        128.js:30
Bubble parent                       128.js:34
Bubble grandparent                  128.js:38
Bubble body                         128.js:42
```

Now, let's change our js file (128_2.js) to below where for capturing events, we keep only grandparent and body events and for bubbling, we keep only parent and child events:

```
const grandparent = document.querySelector(".grandparent");
const parent = document.querySelector(".parent");
const child = document.querySelector(".child");

// capturing events
// from the top to the bottom
grandparent.addEventListener("click", ()=> {
    console.log("Capture !!! grandparent");
}, true);

document.body.addEventListener("click", ()=> {
    console.log("Capture !!! body");
}, true);


// not capture
// from the bottom to the top
child.addEventListener("click", ()=> {
    console.log("Bubble child");
});

parent.addEventListener("click", ()=> {
    console.log("Bubble parent");
});
```

Now, if we click inside child portion, we will see this:

Now, if we change the JS code (128_3.js) as such that event is associated with only grandparent

```
// event delegation

const grandparent = document.querySelector(".grandparent");

grandparent.addEventListener("click", (e)=> {
    console.log("You clicked something!");
});
```

Now, even if we click inside child element which doesn't have any click event associated, we will see `"You clicked something!"` in the console.

It starts looking for event starting from body, then goes to check grandparent, then parent, then child, then it again bubbles to see if there is any click event in parent, if not if there is any click event associated with Grandparent, then it called the callback function of the grandparent and not child. This means child, parent and grandparent don't need separate event listeners due to event delegation.

If the js code is updated (128_4.js),

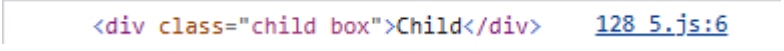
```
const grandparent = document.querySelector(".grandparent");

grandparent.addEventListener("click", (e)=> {
    console.log(e);
});
```

We will see that `PointerEvent` is printed and the value of `target` is `div.child.box`.

We can also print the HTML tag for the portion where it is clicked,

```
const grandparent = document.querySelector(".grandparent");

grandparent.addEventListener("click", (e)=> {
    console.log(e.target);
});
```

Now, if we click inside child portion, we will see the HTML tag for child:



Similarly, if we click at the parent portion, we will see the HTML tag for parent. This will include the HTML tag for child as it is nested inside parent.

```
▼ <div class="parent box">                    128_5.js:6
    " Parent "
    <div class="child box">Child</div>
  </div>
```

Similarly, if we click at the grandparent portion, we will see the HTML tag for grandparent. This will include the HTML tag for parent and child as thesy are nested inside grandparent.

```
                                              128_5.js:6
▼ <div class="grandparent box">
    " Grandparent "
  ▼ <div class="parent box">
      " Parent "
      <div class="child box">Child</div>
    </div>
  </div>
```

Now, if we click in the body part, nothing will happen, as we have not associated any event with body and so event capture doesn't happen.

```
const grandparent = document.querySelector(".grandparent");

grandparent.addEventListener("click", (e)=> {
    console.log(e.target.textContent);
});
```

Now, clicking on parent will give "Parent Child", clicking on grandparent will give "Grandparent Parent Child" and clicking on child will only give "Child".

[ ]:

JS_Beginning_to_Mastery_Part3_1

April 21, 2023

0.1 Synchronous programming vs asynchronous programming

0.1.1 JS is synchronous programming language and single threaded.

0.1.2 Synchronous programming: One line of code is executed after the other. Synchronous programming is single threaded meaning that only one line of code is executed at a time

0.1.3 Asynchronous programming: One line of code is executed after the other, but not necessarily in order. Asynchronous programming is used to prevent the browser from freezing. Asynchronous programming is multi threaded meaning that multiple lines of code can be executed at the same time

Synchronous programming and asynchronous programming are two different programming paradigms in JavaScript.

In synchronous programming, each task is executed in sequence, one after the other. This means that if a task takes a long time to complete, the program will be blocked until that task is finished.

In asynchronous programming, on the other hand, multiple tasks can be executed concurrently, and the program doesn't wait for a task to finish before moving on to the next one. As a result, the program is not blocked, and it can continue to execute other tasks while waiting for a task to complete. Asynchronous programming is commonly used for tasks that involve I/O operations or network requests, which can take a long time to complete.

In JavaScript, asynchronous programming is typically implemented using callbacks, promises, or async/await syntax. Overall, asynchronous programming allows for more efficient and responsive programs, especially when dealing with tasks that involve waiting for external resources. However, it can be more complex to implement than synchronous programming, and it requires a good understanding of JavaScript's asynchronous mechanisms.

All the code we have written so far are synchronous.

```
// synchronous programming example
console.log("script start");

for(let i=0; i<10000; i++) {
    console.log("inside for loop");
}

console.log("script end");
```

In the code example above, the for loop section acts as a blocking section as it prevents `script end` to be printed in the console. This gets printed only after the for loop is finished. Here asynchronous concept will help. If an API takes time in getting loaded, then it will prevent other parts of the page to load in synchronous proggramming, here also asynchronous programming is of great help.

0.1.4 Asynchronous programming example

Call a function after 1 second

```
console.log("script start");
function hello(){
    console.log("inside setTimeout");
}
setTimeout(hello, 1000);
console.log("script end");
```

We can also pass an arrow function to `setTimeout`

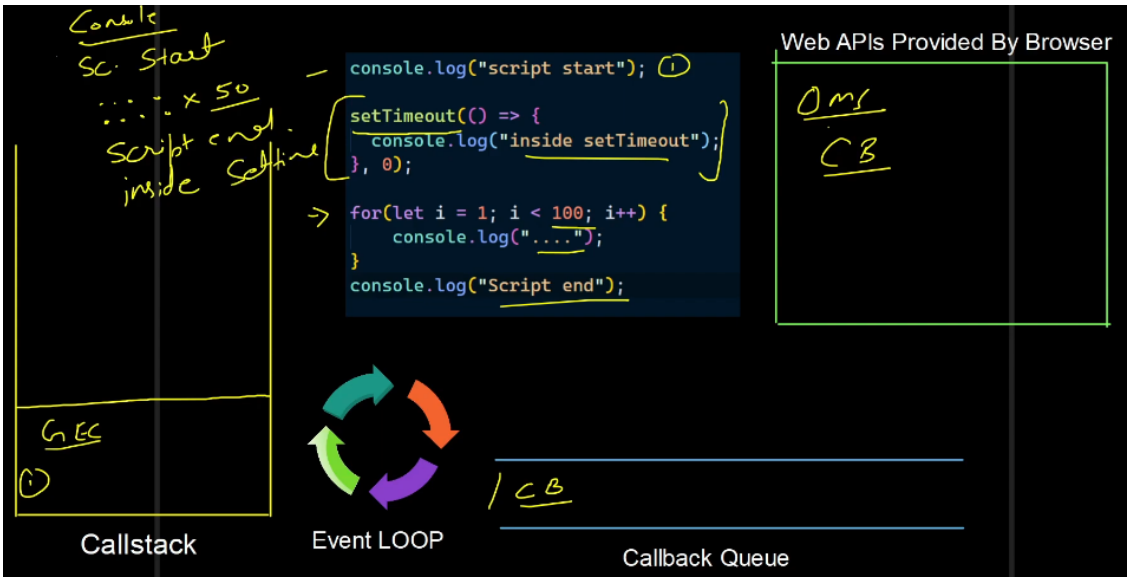
```
console.log("script start");
setTimeout(()=>{
    console.log("inside setTimeout");
}, 1000);
console.log("script end");
```

```
script start                                    130.js:30
script end                                      130.js:34
inside setTimeout                               130.js:32
```

We can see that `hello` gets printed after `script end` is printed.

```
console.log("script start");
setTimeout(()=>{
    console.log("inside setTimeout");
}, 0);

for(let i=0; i<100; i++){
    console.log("....");
}
console.log("script end");
```

```
script start                                    130.js:39
100 ....                                        130.js:45
script end                                      130.js:47
inside setTimeout                               130.js:41
```



`setTimeout()` returns an ID and we can store it in a variable.

```
console.log("script start");
const id = setTimeout(()=>{
    console.log("inside setTimeout");
}, 0);

for(let i=0; i<100; i++){
    console.log("....");
}
console.log("setTimeout id is: ", id);
console.log("script end");
```

```
script start                                    130.js:50
100 ....                                        130.js:56
setTimeout id is:  3                            130.js:58
script end                                      130.js:59
inside setTimeout                               130.js:52
```

Using the id of the setTimeout

```
console.log("script start");
const id = setTimeout(()=>{
    console.log("inside setTimeout");
}, 0);

for(let i=0; i<100; i++){
    console.log("....");
}
console.log("setTimeout id is: ", id);
console.log("clearing the timeout");
clearTimeout(id); // using id
console.log("script end");
```

0.1.5 setInterval

In JavaScript, `setInterval()` is a built-in method that allows you to repeatedly execute a specified function at a set interval of time. It takes two arguments: the function to be executed and the time interval in milliseconds.

Here's an example of how to use `setInterval()` to display a message every 1 second:

```
setInterval(() => {
  console.log("Hello, world!");
}, 1000);
```

In this example, the `setInterval()` function is used to call an anonymous function that logs the message `"Hello, world!"` to the console every 1 second (`1000` milliseconds).

`setInterval()` returns an ID value that can be used to stop the execution of the function using the `clearInterval()` method. For example, the following code stops the execution of the `setInterval()` function defined above after 5 seconds:

```
const intervalId = setInterval(() => {
  console.log("Hello, world!");
}, 1000);

setTimeout(() => {
  clearInterval(intervalId);
}, 5000);
```

In this example, the `setTimeout()` function is used to call the `clearInterval()` method after 5 seconds, passing in the `intervalId` returned by `setInterval()` as its argument. This stops the execution of the `setInterval()` function after 5 seconds.

`setInterval()` is commonly used for animations, real-time updates, and polling for new data from a server. However, it's important to use it with caution, as it can consume a lot of resources if the interval is set too low or if the function being executed takes a long time to complete.

0.1.6 callback function

callback function is a function that is passed as an argument to another function

In JavaScript, a callback is a function that is passed as an argument to another function, and is intended to be executed after some specific task or event has occurred.

Callbacks are commonly used in asynchronous programming, where a function takes some time to complete and does not block the execution of other parts of the program. In this scenario, a callback function is used to specify what should happen after the asynchronous task has completed.

For example, in the following code, the setTimeout function takes two arguments: a callback function and a time delay in milliseconds. The `setTimeout` function schedules the execution of the

callback function after the specified delay:

```
setTimeout(function() {
  console.log('This will be printed after 1000ms');
}, 1000);
```

In this case, the anonymous function passed as the first argument is the callback function that will be executed after 1000 milliseconds.

Callbacks can also be used to handle events, such as mouse clicks or keyboard presses. For instance, the `addEventListener` method allows you to register an event listener that executes a callback function when the event occurs:

```
document.addEventListener('click', function() {
  console.log('The document was clicked');
});
```

In this example, the anonymous function passed as the second argument is the callback function that will be executed when the `click` event occurs on the `document` object.

Callbacks are a fundamental concept in JavaScript and are widely used in modern web development, especially in combination with other asynchronous programming techniques such as promises and async/await.

[1]:
```
function myFunc(callback){
    console.log("Function is doing task 1");
    callback();
}

function myFunc2(){
    console.log("Function is doing task 2");
}

myFunc(myFunc2);
```

```
Function is doing task 1
Function is doing task 2
```

We can also write function inside function like the following:

```
myFunc(function(){
    console.log("function is doing task 2");
});
```

We can also use arrow function:

```
myFunc(()=> {
    console.log("Function inside is doing task 2");
})
```

[2]:
```
function getTwoNumbersAndAdd(number1, number2, callback){
    console.log(number1, number2);
    callback(number1, number2);
```

```
}

function addTwoNumbers(number1, number2){
    console.log(number1 + number2);
}

getTwoNumbersAndAdd(4, 5, addTwoNumbers);
```

```
4 5
9
```

0.1.7 Callback hell, pyramid of doom

we want first to change the text of the heading1 to “Heading 1” and then change the color of the heading1 to “violet” after 1 second, after that to change the text of the heading2 to “Heading 2” and then change the color of the heading2 to “purple” after 3 seconds, and so on.

```
const heading1 = document.querySelector('.heading1');
const heading2 = document.querySelector('.heading2');
const heading3 = document.querySelector('.heading3');
const heading4 = document.querySelector('.heading4');
const heading5 = document.querySelector('.heading5');
const heading6 = document.querySelector('.heading6');
const heading7 = document.querySelector('.heading7');
const heading8 = document.querySelector('.heading8');
const heading9 = document.querySelector('.heading9');
const heading10 = document.querySelector('.heading10');

setTimeout(()=>{
    heading1.textContent = "Heading1";
    heading1.style.color = "violet";
    setTimeout(()=>{
        heading2.textContent = "Heading2";
        heading2.style.color = "purple";
        setTimeout(()=>{
            heading3.textContent = "Heading3";
            heading3.style.color = "red";
            setTimeout(()=>{
                heading4.textContent = "Heading4";
                heading4.style.color = "pink";
                setTimeout(()=>{
                    heading5.textContent = "Heading5";
                    heading5.style.color = "green";
                    setTimeout(()=>{
                        heading6.textContent = "Heading6";
                        heading6.style.color = "blue";
                        setTimeout(()=>{
                            heading7.textContent = "Heading7";
                            heading7.style.color = "brown";
```

```
                    }, 1000);
                }, 3000);
            }, 2000);
        }, 1000);
    }, 2000);
}, 1000);
```

The above can also be done using function.

```
function changeText(element, text, color, time, onSuccessCallback, onFailureCallback){
    setTimeout(()=>{
        if(element){
            element.textContent = text;
            element.style.color = color;
            if (onSuccessCallback){
                onSuccessCallback();
            };
        } else{
            if(onFailureCallback){
                onFailureCallback();
            }
        }
    }, time)
}

// Pyramid of doom

changeText(heading1, "one", "green", 1000, ()=>{
    changeText(heading2, "two", "purple", 2000, ()=>{
        changeText(heading3, "three", "red", 1000, ()=>{
            changeText(heading4, "four", "pink", 1000, ()=>{
                changeText(heading5, "five", "green", 2000, ()=>{
                    changeText(heading6, "six", "blue", 1000, ()=>{
                        changeText(heading7, "seven", "brown", 1000, ()=>{
                            changeText(heading8, "eight", "cyan", 1000, ()=>{
                                changeText(heading9, "nine", "#cda562", 1000, ()=>{
                                    changeText(heading10, "ten", "#dca652", 1000, ()=>{

                                    }, ()=>{
                                        console.log("Heading9 doesn't exist")
                                    });
                                }, ()=>{
                                    console.log("Heading9 doesn't exist")
                                });
                            }, ()=>{
                                console.log("Heading8 doesn't exist")
                            });
```

```
                    }, ()=>{
                        console.log("Heading7 doesn't exist")
                    });
                }, ()=>{
                    console.log("Heading6 doesn't exist")
                });
            }, ()=>{
                console.log("Heading5 doesn't exist")
            });
        }, ()=>{
            console.log("Heading4 doesn't exist")
        });
    }, ()=>{
        console.log("Heading3 doesn't exist")
    });
}, ()=>{
    console.log("Heading2 doesn't exist")
});
}, ()=>{
    console.log("Heading1 doesn't exist")
});
```

Promise help us to avoid this nested structure.

0.1.8 Promise

In JavaScript, a Promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to handle asynchronous operations in a more structured way, making your code more readable and maintainable.

Promises have three states:

- Pending: The initial state. The promise is neither fulfilled nor rejected.
- Fulfilled: The operation completed successfully, and the promise has a resulting value.
- Rejected: The operation failed, and the promise has a reason for the failure. When you create a new Promise, you pass a function that defines the asynchronous operation to be performed. This function takes two arguments: `resolve` and `reject`. If the operation is successful, you call `resolve` and pass the result value. If the operation fails, you call `reject` and pass an error object or message.

Here's an example of how to create and use a Promise:

```
const myPromise = new Promise((resolve, reject) => {
  // Perform an asynchronous operation, such as an API call
  // If the operation is successful, call resolve with the result
  // If the operation fails, call reject with an error object or message
});

myPromise.then(result => {
  // Handle the successful completion of the operation
```

```
}).catch(error => {
  // Handle the failure of the operation
});
```

In this example, `myPromise` is a new Promise that you create by passing a function with `resolve` and `reject` arguments. You can then use the `then()` method to handle the successful completion of the operation, and the `catch()` method to handle any errors that occur.

Promise can both be saved as a variable and returned as a function. We can rewrite the heading change code using Promise in the following way:

```
function changeText(element, text, color, time) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (element) {
        element.textContent = text;
        element.style.color = color;
        resolve();
      } else {
        reject("Element doesn't exist");
      }
    }, time);
  });
}

changeText(heading1, "one", "green", 1000)
  .then(() => changeText(heading2, "two", "purple", 2000))
  .then(() => changeText(heading3, "three", "red", 1000))
  .then(() => changeText(heading4, "four", "pink", 1000))
  .then(() => changeText(heading5, "five", "green", 2000))
  .then(() => changeText(heading6, "six", "blue", 1000))
  .then(() => changeText(heading7, "seven", "brown", 1000))
  .then(() => changeText(heading8, "eight", "cyan", 1000))
  .then(() => changeText(heading9, "nine", "#cda562", 1000))
  .then(() => changeText(heading10, "ten", "#dca652", 1000))
  .catch((error) => {
    console.log(error);
  });
```

Simple Promise example We promise to prepare fried rice if we have all the ingredients in the bucket

Produce promise

```
const friedRicePromise = new Promise((resolve, reject)=>{
    if (bucket.includes("vegetables") && bucket.includes("rice") && bucket.includes("salt") &&
        resolve("Fried Rice")
    } else{
        reject("No Fried Rice");
```

```
    }
});
```

Consume promise the argument passed to the resolve function is passed to the then function as we passed `Fried Rice` to the `resolve` function, the value of `myFriedRice` is `"Fried Rice"`.

```
friedRicePromise.then((myFriedRice)=>{
    // the argument passed to the resolve function is passed to the then function
    // as we passed "Fried Rice" to the resolve function, the value of myFriedRice
    // is "Fried Rice"
    console.log("let's eat ", myFriedRice);
})
```

Now, if we don't have all the elements, for example, missing rice, the promise will not be kept.

```
const bucket = ['coffee', 'chips', 'vegetables', 'salt']
```

We will see this error message: **Uncaught (in promise) No Fried Rice**

So, we can also catch this error by passing another callback function inside then in the following way:

```
friedRicePromise.then((myFriedRice)=>{
    console.log("let's eat ", myFriedRice);
}, (error)=>{console.log(error)})
```

Now, we will see no error, and a message **"No Fried Rice"** in the console.

```
const friedRicePromise = new Promise((resolve, reject)=>{
    if (bucket.includes("vegetables") && bucket.includes("rice") && bucket.includes("salt") &&
        resolve({value: "Fried Rice"}) // we can also pass an object or an array
    } else{
        reject(new Error("something missing from bucket"));
    }
});

friedRicePromise.then((myFriedRice)=>{
    console.log("let's eat ", myFriedRice);
}).catch((error)=>{console.log(error)})
```

will show the following if not all elements are not in the bucket:

```
Error: something missing from bucket                                    134.js:70
    at 134.js:64:16
    at new Promise (<anonymous>)
    at 134.js:60:26
```

Instead of this `reject(new Error("something missing from bucket"));`, we can also write `reject("something missing from bucket");` in the following way:

```
const friedRicePromise = new Promise((resolve, reject)=>{
    if (bucket.includes("vegetables") && bucket.includes("rice") && bucket.includes("salt") &&
        resolve({value: "Fried Rice"}) // we can also pass an object or an array
    } else{
```

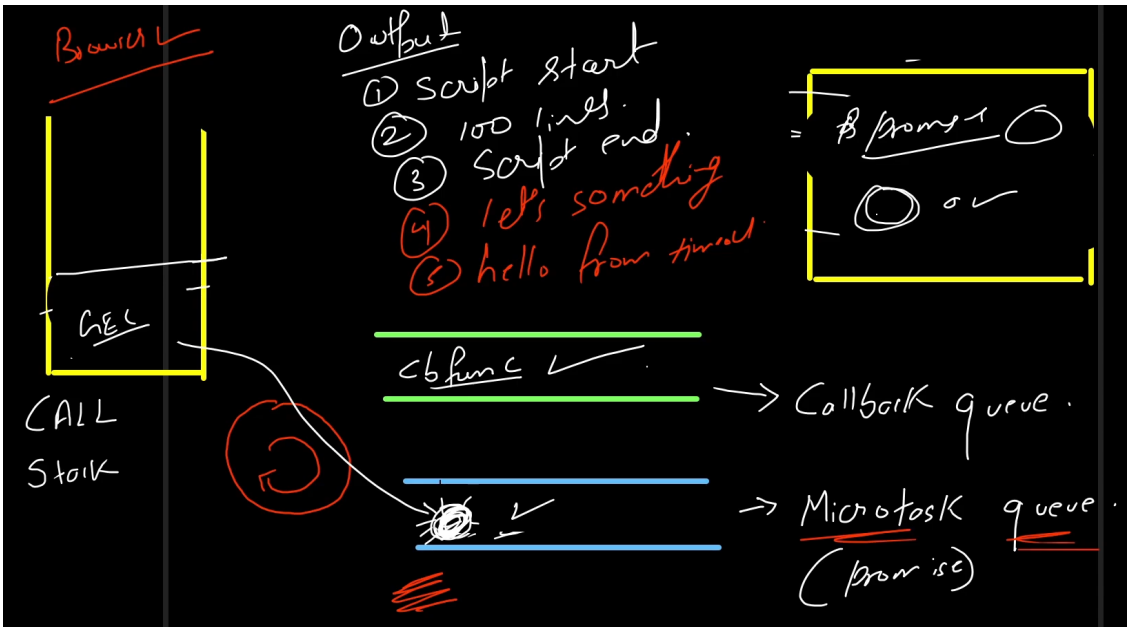
```
        reject("something missing from bucket"); // this is different
    }
});

friedRicePromise.then((myFriedRice)=>{
    console.log("let's eat ", myFriedRice);
}).catch((error)=>{console.log(error)})
```

```
something missing from bucket                                    134.js:86
>
```

Promise is not a JS feature, rather a feature of browser. Promise is executed by the browser. Promise is done asynchronously. Browser will consume promise. Promise is added in microtask queue. Steps under promises are sent to microtask queue and other asynchronous steps are sent for execution by callback queue. If there are actions to be performed both stacked in callback queue and microtask queue, microtask queue gets more priority and the task in microstask queue is moved to call stack queue first.

```
// Promise
console.log("script start");
const bucket = ['coffee', 'chips','vegetables','salt','rice'];



const friedRicePromise = new Promise((resolve,reject)=>{
    if(bucket.includes("vegetables")&& bucket.includes("salt") && bucket.includes("rice")){
        resolve({value:"friedrice"});
    }else{
        reject("could not do it");
    }
})

// produce



// consume
// how to consume

friedRicePromise.then(
    // jab promise resolve hoga
    (myfriedRice)=>{
    console.log("lets eat ", myfriedRice);
    }
    ).catch(
    (error)=>{
        console.log(error)
    })
```

```
setTimeout(()=>{
    console.log("hello from settimeout")
},0)

for(let i = 0; i <=100; i++){
    console.log(Math.random(), i);
}

console.log("script end!!!!")
```




```
0.802859518886889 87                                    134 2.js:40
0.2336602490976154 88                                   134 2.js:40
0.9806019129789179 89                                   134 2.js:40
0.3658045071582523 90                                   134 2.js:40
0.5858031750440631 91                                   134 2.js:40
0.7739992209078201 92                                   134 2.js:40
0.3761961700469083 93                                   134 2.js:40
0.8493964071839006 94                                   134 2.js:40
0.06786330434862986 95                                  134 2.js:40
0.13516532383643498 96                                  134 2.js:40
0.9130042616742517 97                                   134 2.js:40
0.3608363380175552 98                                   134 2.js:40
0.2137275353668675 99                                   134 2.js:40
0.6877394741870324 100                                  134 2.js:40
script end!!!!                                          134 2.js:43
lets eat ▶ {value: 'friedrice'}                         134 2.js:27
hello from settimeout                                   134 2.js:36
```

0.1.9 Function returning promise

```
// function returning promise
```

```
function friedRicePromise(){
    const bucket = ['coffee', 'chips', 'vegetables', 'salt', 'rice'];
    return new Promise((resolve, reject) => {
        if (bucket.includes("vegetables") && bucket.includes("rice") && bucket.includes("salt")
            resolve({value: "Fried Rice"});
        } else{
            reject("Not enough ingredients");
        }
    })
}

friedRicePromise().then((myFriedRice)=>{
    console.log("let's eat ", myFriedRice);
}).catch((error)=>{
    console.log(error);
});
```

0.1.10 AJAX

AJAX (Asynchronous JavaScript and XML) is a technique used in JavaScript programming to send and receive data from a web server without reloading the entire page. This allows for faster, more interactive and dynamic web applications.

To implement AJAX in JavaScript, you need to use the XMLHttpRequest (XHR) object, which is built into most modern browsers. Here's an example of how to create an XHR object:

```
var xhr = new XMLHttpRequest();
```

Once you have created an XHR object, you can use it to make an HTTP request to the server using the `open()` method. Here's an example of how to make a `GET` request to a server:

```
xhr.open('GET', '/data.json', true);
```

The first parameter of the open() method specifies the HTTP method (`GET`, `POST`, `PUT`, `DELETE`, etc.), the second parameter specifies the URL of the server, and the third parameter specifies whether the request should be asynchronous (`true`) or synchronous (`false`).

Next, you need to define a callback function that will be called when the server responds to the request. This function is usually defined using the onreadystatechange event. Here's an example:

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4 && xhr.status == 200) {
    var data = JSON.parse(xhr.responseText);
    // Do something with the data
  }
};
```

The readyState property of the XHR object indicates the current state of the request, which can have the following values:

- 0: uninitialized
- 1: loading

- 2: loaded
- 3: interactive
- 4: complete The status property indicates the HTTP status code returned by the server.

Finally, you can send the request to the server using the `send()` method:

```
xhr.send();
```

Here's a complete example that demonstrates how to use AJAX to load data from a server and display it on a web page:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/data.json', true);
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4 && xhr.status == 200) {
    var data = JSON.parse(xhr.responseText);
    var list = document.getElementById('list');
    for (var i = 0; i < data.length; i++) {
      var item = document.createElement('li');
      item.innerHTML = data[i].name + ': ' + data[i].value;
      list.appendChild(item);
    }
  }
};
xhr.send();
```

In this example, the code loads data from a JSON file called '`data.json`', creates a list of items from the data, and appends the list to an HTML element with an id of `'list'`.

Here are some more examples and use cases for AJAX in JavaScript:

1. **Form submission:** You can use AJAX to submit a form to the server without reloading the entire page. This allows for a smoother user experience and can save time by not requiring the user to wait for the page to reload. Here's an example:

```
var form = document.getElementById('myForm');
form.addEventListener('submit', function(event) {
  event.preventDefault();
  var xhr = new XMLHttpRequest();
  xhr.open('POST', '/submit', true);
  xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && xhr.status == 200) {
      var response = JSON.parse(xhr.responseText);
      // Do something with the response
    }
  };
  xhr.send(new FormData(form));
});
```

In this example, the code prevents the default form submission behavior, creates an XHR object, sets the HTTP method to POST, sets the content type header to 'application/x-www-form-urlencoded',

and sends the form data as a FormData object. The server can then process the form data and return a response that can be handled in the callback function.

2. Dynamic content loading: You can use AJAX to load dynamic content into a web page without reloading the entire page. This allows for a more interactive and dynamic user experience. Here's an example:

```
var button = document.getElementById('loadButton');
button.addEventListener('click', function() {
  var xhr = new XMLHttpRequest();
  xhr.open('GET', '/data.json', true);
  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && xhr.status == 200) {
      var data = JSON.parse(xhr.responseText);
      var content = document.getElementById('content');
      content.innerHTML = data.content;
    }
  };
  xhr.send();
});
```

In this example, the code creates an XHR object, sets the HTTP method to GET, and sends a request to the server to load data from a JSON file. When the server responds with the data, the callback function replaces the content of an HTML element with an id of 'content' with the new data.

3. Autocomplete search: You can use AJAX to implement an autocomplete search feature that suggests search terms as the user types. This can improve the user experience and help the user find what they are looking for more quickly. Here's an example:

```
var input = document.getElementById('searchInput');
input.addEventListener('input', function() {
  var xhr = new XMLHttpRequest();
  xhr.open('GET', '/search?q=' + encodeURIComponent(input.value), true);
  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && xhr.status == 200) {
      var suggestions = JSON.parse(xhr.responseText);
      // Display the suggestions to the user
    }
  };
  xhr.send();
});
```

In this example, the code creates an XHR object, sets the HTTP method to GET, and sends a request to the server with a query parameter 'q' that contains the user's search term. The server can then return a list of suggested search terms based on the user's input, which can be displayed to the user in real-time.

`encodeURIComponent()` is a built-in JavaScript function that encodes a string as a valid URI (Uniform Resource Identifier) component by replacing all special characters with their corresponding encoded values.

In the context of this AJAX example, `encodeURIComponent(input.value)` is used to encode the user's search term before it is included in the URL query parameter. This is necessary because URL query parameters must be encoded to avoid any special characters or reserved characters that can disrupt the query.

For example, if the user enters the search term "pizza toppings", the resulting URL with the encoded query parameter would be `/search?q=pizza%20toppings`. The `%20` represents a space character that has been encoded.

Without encoding the search term, the resulting URL could look like `/search?q=pizza toppings`, which is not a valid URL and would result in errors. By encoding the search term with `encodeURIComponent()`, we ensure that the resulting URL is valid and can be sent to the server for processing.

0.1.11 AJAX : asynchronous javascript and XML

- HTTP request: is a set of "web development techniques" using many web technologies on the "client-side" to create asynchronous web applications. Browser is considered as client.

With Ajax, web applications can send and retrieve data from a server asynchronously (in the background) without interfering with the display and behaviour of the existing page.

We don't use data in XML format anymore, we use JSON now.

We have 3 most common ways to create and send request to server

1. xmlHTTPRequest (old way of doing)
2. fetch API (new way of doing)
3. axios (this is third party library)

XHR — XMLHttpRequest JSON is different from JS object in these ways: a. JSON key has to be within " " and there can't be any methods.

In AJAX, the onreadystatechange attribute is a callback function that gets called every time the readyState property of the XMLHttpRequest object changes. The readyState property indicates the state of the request, and can have the following values:

- 0: UNSENT - the XMLHttpRequest object has been created, but open() method has not been called yet.
- 1: OPENED - the open() method has been called, but send() method has not been called yet.
- 2: HEADERS_RECEIVED - the send() method has been called, and the server has responded with the headers of the response.
- 3: LOADING - the response body is being received. This value is usually not used in practice.
- 4: DONE - the entire response has been received and is available.

So, the values of 2, 3, and 4 for the `onreadystatechange` attribute correspond to the readyState values of `HEADERS_RECEIVED`, `LOADING`, and `DONE`, respectively.

When the `readyState` value changes to 2 (i.e., `HEADERS_RECEIVED`), the `onreadystatechange` callback function can access the response headers using the `getAllResponseHeaders()` or `getResponseHeader()` methods of the XMLHttpRequest object.

When the `readyState` value changes to 3 (i.e., `LOADING`), the `onreadystatechange` callback function can access the partial response using the `responseText` or `responseXML` properties of the XMLHttpRequest object.

When the `readyState` value changes to 4 (i.e., `DONE`), the `onreadystatechange` callback function can access the entire response using the `responseText` or `responseXML` properties of the XMLHttpRequest object.

HTTP status code All HTTP response status codes are separated into five classes or categories. The first digit of the status code defines the class of response, while the last two digits do not have any classifying or categorization role. There are five classes defined by the standard:

- 1xx informational response – the request was received, continuing process
- 2xx successful – the request was successfully received, understood, and accepted
- 3xx redirection – further action needs to be taken in order to complete the request
- 4xx client error – the request contains bad syntax or cannot be fulfilled
- 5xx server error – the server failed to fulfil an apparently valid request

```
const URL = "https://jsonplaceholder.typicode.com/posts";
const xhr = new XMLHttpRequest();
// console.log(xhr);
// console.log(xhr.readyState);
// console.log(xhr.readyState); // 0
xhr.open("GET", URL); // browser will do this asynchronously
// console.log(xhr.readyState); // 1

xhr.onreadystatechange = function(){
    // console.log(xhr.readyState);
    if(xhr.readyState === 4){
        // console.log(xhr.response);
        // console.log(typeof xhr.response);
        console.log(xhr.status); // gives 200
        const response = xhr.response;
        const data = JSON.parse(response); // converts to JS object
        console.log(typeof data); // object
    }
}

xhr.send();
```

Instead of the above code, we can use the below code using `onload` which runs only when the readyState is 4

```
const URL = "https://jsonplaceholder.typicode.com/posts";
const xhr = new XMLHttpRequest();
// console.log(xhr);
// console.log(xhr.readyState);
// console.log(xhr.readyState); // 0
xhr.open("GET", URL); // browser will do this asynchronously
// console.log(xhr.readyState); // 1
```

```
xhr.onload = function(){
    const response =  xhr.response;
    const data = JSON.parse(response);
    console.log(data);
}

xhr.send();
```

Example of using XHR

```
const URL = "https://jsonplaceholder.typicode.com/posts";

const xhr = new XMLHttpRequest();

xhr.open("GET", URL);

xhr.onload = function(){
    if (xhr.status >= 200 && xhr.status < 300){
        const data = JSON.parse(xhr.response);
        console.log(data);
        const id = data[3].id;
        const xhr2 = new XMLHttpRequest();
        const URL2 = `${URL}/${id}`;
        console.log(URL2);
        xhr2.open("GET", URL2);
        xhr2.onload = ()=>{
            const data2 = JSON.parse(xhr2.response);
            console.log(data2);
        }
        xhr2.send();
    }
    else{
        console.log("something went wrong");
    }
}
```

Now, we will do the same as before using `Promise`

```
const URL = "https://jsonplaceholder.typicode.com/posts";

function sendRequest(method, url) {
    return new Promise(function(resolve, reject) {
        const xhr  = new XMLHttpRequest();
        xhr.open(method, url);
        xhr.onload = function() {
            if(xhr.status >= 200 && xhr.status < 300){
                resolve(xhr.response);
            }
            else{
```

```
                reject(new Error("Something Went wrong"));
            }
        }

        xhr.onerror = function() {
            reject(new Error("Something went wrong"));
        }

        xhr.send();
    })
}

sendRequest("GET", URL)
    .then(response => {
        const data = JSON.parse(response);
        // console.log(data)
        return data;
    })
    .then(data=>{
        const id = data[3].id;
        return id;
    })
    .then(id=>{
        const url = `${URL}/${id}ssss`;
        return sendRequest("GET", url);
    })
    .then(newResponse => {
        const newData = JSON.parse(newResponse);
        console.log(newData);
    })
    .catch(error =>{
        console.log(error);
    })
```

0.1.12 fetch – it does the GET request by default

fetch returns Promise

In JavaScript, `fetch()` is a built-in function that is used to make network requests to a server and retrieve data asynchronously. It returns a Promise that resolves to the Response object representing the response to the request.

Here is an example of how to use `fetch()` to make a GET request and retrieve data from a server:

```
fetch('https://example.com/data.json')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

In this example, `fetch()` is called with the URL of a JSON file on a server. The Promise returned

by `fetch()` resolves to a Response object, which is passed to the first `.then()` method. The `json()` method of the Response object is called to parse the response as JSON, and the resulting data is passed to the second `.then()` method. Finally, any errors that occur during the request are caught by the `.catch()` method and logged to the console.

`fetch()` can also be used to make other types of requests, such as POST or PUT requests, and to include additional options such as headers or authentication credentials.

```
// fetch

const URL = "https://jsonplaceholder.typicode.com/posts";

fetch(URL,{
    method: 'POST',
    body: JSON.stringify({
        title: 'foo',
        body: 'bar',
        userId: 1,
    }),
    headers: {
        'Content-type': 'application/json; charset=UTF-8',
    },
    })
    .then(response => {
        if(response.ok){
            return response.json();
        } else{
            throw new Error("Something went wrong!!!");
        }
    })
    .then(data=>{
        console.log(data);
    })
    .catch(error=>{
        console.log("inside catch");
        console.log(error);
    })
```

0.1.13 `async await`

fetch followed by multiple then can be replaced by using `async await`

`async/await` is a modern approach for handling asynchronous operations in JavaScript. It provides a way to write asynchronous code that looks like synchronous code and is easier to read and understand.

The `async` keyword is used to define a function that returns a Promise. When a function is marked as `async`, it allows us to use the `await` keyword inside it to wait for the resolution of a Promise.

Here's an example:

```
async function getData() {
  const response = await fetch('https://example.com/data.json');
  const data = await response.json();
  return data;
}

getData()
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

In this example, `getData()` is an async function that fetches data from a server using fetch() and returns it as a Promise. The `await` keyword is used to wait for the response to be returned from the server before parsing it as JSON using the `json()` method. The parsed data is then returned from the function.

Note that the `await` keyword can only be used inside an async function.

Using `async/await` can make code more readable and easier to understand, especially when dealing with complex asynchronous operations that involve multiple requests and callbacks.

One thing to keep in mind when using `async/await` is error handling. If an error occurs in an `async function`, it will automatically be caught and converted into a rejected Promise. To handle errors, we can use a `try/catch` block like this:

```
async function getData() {
  try {
    const response = await fetch('https://example.com/data.json');
    const data = await response.json();
    return data;
  } catch (error) {
    console.error(error);
    throw new Error('Failed to fetch data');
  }
}
```

In this example, we use a `try/catch` block to catch any errors that occur during the execution of the function. If an error occurs, it is logged to the console and a new `Error` object is thrown with a custom error message. The `catch` block can also be used to perform other error handling tasks, such as displaying an error message to the user or logging the error to a remote service.

```
const URL = "https://jsonplaceholder.typicode.com/posts";

// writing async before function makes it return a promise
async function getPosts(){

}

const returned = getPosts();
console.log(returned);
```

```
▼ Promise i                                                    144.js:10
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: undefined
```

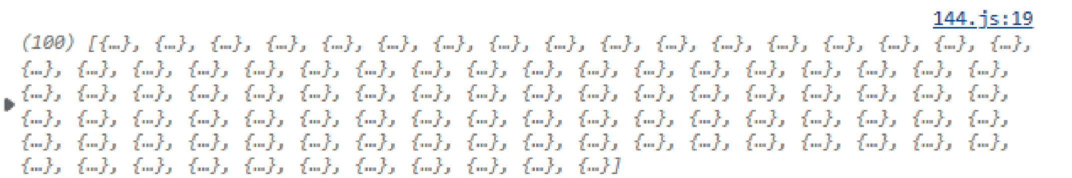
```
const URL = "https://jsonplaceholder.typicode.com/posts";

// writing async before function makes it return a promise
async function getPosts(){
    // fetch(URL) returns a promise
    // we can use the term await before a promise to wait for it to resolve
    const response = await fetch(URL);
    if(!response.ok){
        throw new Error("Something went wrong!");
    }
    // as json method also returns a promise, we can use await before it
    const data = await response.json();
    return data; // we are returning a promise here
}

getPosts()
.then(myData => {
    console.log(myData);
})
.catch(error=>{
    console.log("inside catch");
    console.log(error);
})
```

```
                                                               144.js:19
(100) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
▶ {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
```

Writing async function using arrow function

```
// Writing async function using arrow function

const URL = "https://jsonplaceholder.typicode.com/posts";

// writing async before function makes it return a promise
const getPosts = async()=>{
    // fetch(URL) returns a promise
    // we can use the term await before a promise to wait for it to resolve
    const response = await fetch(URL);
    if(!response.ok){
        throw new Error("Something went wrong!");
    }
```

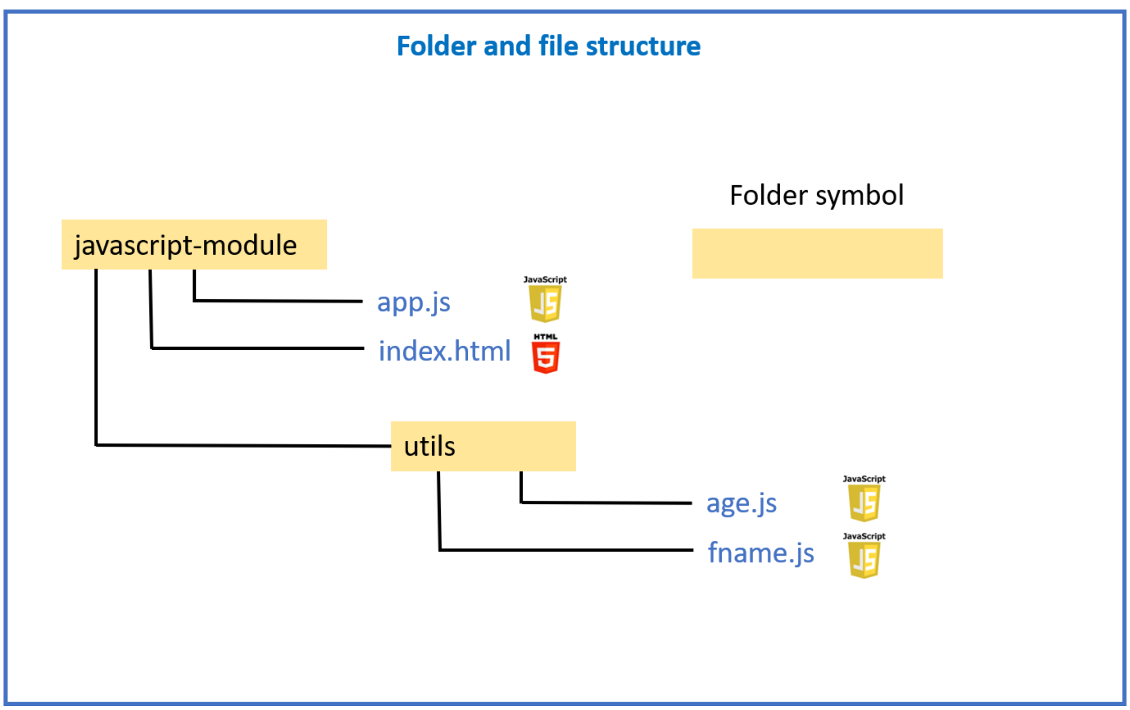
```
    // as json method also returns a promise, we can use await before it
    const data = await response.json();
    return data; // we are returning a promise here
}

getPosts()
.then(myData => {
    console.log(myData);
})
.catch(error=>{
    console.log("inside catch");
    console.log(error);
})
```



We can split our code to multiple files using ES6.

0.1.14 Importing variables defined in multiple JS files

Suppose the folder and file structure we have looks like the following:



The code of **index.html** is like the following:

Note the use of `type="module"` inside `script` tag and dropping defer term.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="./app.js" type="module"></script>
    <title>Document</title>
</head>
<body>
    <h1>JavaScript Module</h1>
</body>
</html>
```

The code of **age.js** is the following. Note the use of `export` before the variable declaration.

```
export const age = 31;
```

The code of **fname.js** is the following:

```
export const firstName = "John";
```

In the **app.js**, we import `firstName` variable from **fname.js** and `age` variable from **age.js** in the following way:

```
// we want to import firstName from fname.js and age from age.js

import {firstName} from './utils/fname.js';
import {age} from './utils/age.js';

console.log(firstName, age);
```

We can also use `export` after declaring a variable. The code of **age.js** is the following. Note the use of `export` after the variable declaration in a separate line.

```
const age = 31;
export {age};
```

We can also shorten the variable name after importing in the JS file and use this shortened name for variable. The code for **app.js** can be also written as:

```
import {firstName as fname} from './utils/fname.js';
import {age} from './utils/age.js';

console.log(fname, age);
```

We can export anything, not only variables.

Suppose, inside **utils** subfolder, we have another JS file named **Person.js**.

```
export class Person{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
```

```
        this.age = age;
    }

    info(){
        console.log(this.firstName, this.lastName, this.age)
    }
}
```

Now, we change the file **app.js** to the following:

```
import {firstName} from './utils/fname.js';
import {age} from './utils/age.js';
import {Person} from './utils/Person.js';

console.log(firstName, age);

const person = new Person("John", "Doe", 31);
person.info();
console.log(person);
```

If we open **index.html** and look at the console, we will see the following:

Inside the **Person.js**, if we write `export default class Person{ ... }` instead of `export class Person{ ... }`, we can import class `Person` without curly braces inside **app.js**.

In this case, the code for **Person.js** looks like the following:

```
export default class Person{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    info(){
        console.log(this.firstName, this.lastName, this.age)
    }
}
```

In this case, the code for **app.js** will look like the following:

```
import {firstName} from './utils/fname.js';
import {age} from './utils/age.js';
import Person from './utils/Person.js';
```

```
console.log(firstName, age);

const person = new Person("John", "Doe", 31);
person.info();
console.log(person);
```

Remember that more than one class or variable can't use export default in a single JS file. One file allows only one `export default`.

Suppose, we have more than one functions to export and we want at least one to have `export default`, then the rest have to use `export` only like the following (**Person.js**):

```
export default class Person{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    info(){
        console.log(this.firstName, this.lastName, this.age)
    }
}

export class Person2{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.fullName = `${firstName} ${lastName}`;
        this.age = age;
    }

    info(){
        console.log(this.fullName, this.age)
    }
}
```

Now, the file **app.js** will look like this:

```
// we want to import firstName from fname.js and age from age.js

import {firstName} from './utils/fname.js';
import {age} from './utils/age.js';
import Person from './utils/Person.js';
import { Person2 } from './utils/Person.js';

console.log(firstName, age);

const person = new Person("John", "Doe", 31);
person.info();
```

```
console.log(person);

const person2 = new Person2("Milon", "Shah", 21);
person2.info();
console.log(person2);
```

```
John 31                                                                    app.js:8
John Doe 31                                                                Person.js:9
▶ Person {firstName: 'John', lastName: 'Doe', age: 31}                     app.js:12
Milon Shah 21                                                              Person.js:22
▶ Person2 {firstName: 'Milon', lastName: 'Shah', fullName: 'Milon Shah', age: 21}   app.js:16
```

Instead of writing {Person} and {Person2} in two lines as shown before:

```
import Person from './utils/Person.js';
import { Person2 } from './utils/Person.js';
```

We can also write them in a single line like this `import Person, {Person2} from './utils/Person.js';`

Now, suppose the file **Person.js** has three classes like the following:

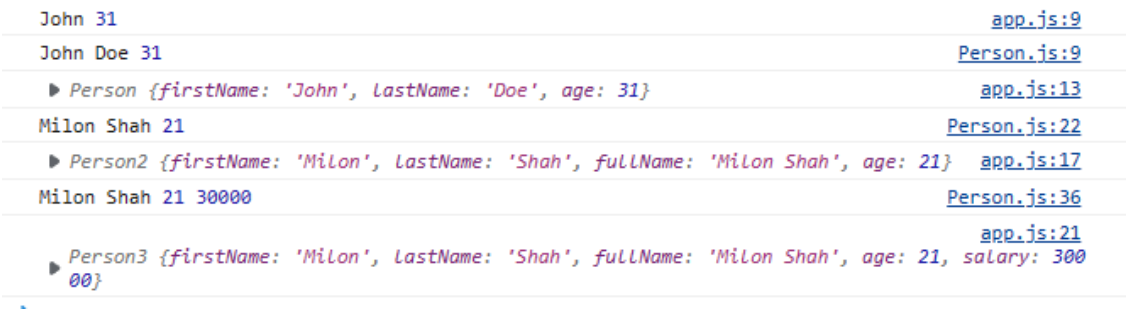
```
export default class Person{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    info(){
        console.log(this.firstName, this.lastName, this.age)
    }
}

export class Person2{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.fullName = `${firstName} ${lastName}`;
        this.age = age;
    }

    info(){
        console.log(this.fullName, this.age)
    }
}

export class Person3{
    constructor(firstName, lastName, age, salary){
        this.firstName = firstName;
```

```
        this.lastName = lastName;
        this.fullName = `${firstName} ${lastName}`;
        this.age = age;
        this.salary = salary
    }

    info(){
        console.log(this.fullName, this.age, this.salary)
    }
}
```

Now, to use these all 3 classes, we can rewrite **app.js** in the following way:

```
// we want to import firstName from fname.js and age from age.js

import {firstName} from './utils/fname.js';
import {age} from './utils/age.js';
// import Person from './utils/Person.js';
// import { Person2 } from './utils/Person.js';
import Person, {Person2, Person3} from './utils/Person.js';

console.log(firstName, age);

const person = new Person("John", "Doe", 31);
person.info();
console.log(person);

const person2 = new Person2("Milon", "Shah", 21);
person2.info();
console.log(person2);

const person3 = new Person3("Milon", "Shah", 21, 30000);
person3.info();
console.log(person3);
```

Now, the output for **index.html** in the console will be:



Now, suppose for the **Person.js** file, we now want to export a variable as `default` in addition to the three classes we already have defined. In the **app.js** we don't necessarily need to name the variable as it is and can import with any arbitrary name that we want.

The code for **Person.js** now:

```
export class Person{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    info(){
        console.log(this.firstName, this.lastName, this.age)
    }
}

export class Person2{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.fullName = `${firstName} ${lastName}`;
        this.age = age;
    }

    info(){
        console.log(this.fullName, this.age)
    }
}

export class Person3{
    constructor(firstName, lastName, age, salary){
        this.firstName = firstName;
        this.lastName = lastName;
        this.fullName = `${firstName} ${lastName}`;
        this.age = age;
        this.salary = salary
    }

    info(){
        console.log(this.fullName, this.age, this.salary)
    }
}

const hello = "Hello World";
export default hello;
```

The code for **app.js** is modified as follwos. Please take a note of how we have imported default variable and other functions using this line `import something, {Person, Person2, Person3} from './utils/Person.js';`

```
// we want to import firstName from fname.js and age from age.js
```

```
import {firstName} from './utils/fname.js';
import {age} from './utils/age.js';
// import Person from './utils/Person.js';
// import { Person2 } from './utils/Person.js';
import something, {Person, Person2, Person3} from './utils/Person.js';

console.log(firstName, age);

const person = new Person("John", "Doe", 31);
person.info();
console.log(person);

const person2 = new Person2("Milon", "Shah", 21);
person2.info();
console.log(person2);

const person3 = new Person3("Milon", "Shah", 21, 30000);
person3.info();
console.log(person3);

console.log(something);
```

Now, the output for **index.html** in the console will be:

```
John 31                                                                    app.js:9
John Doe 31                                                                Person.js:9
▶ Person {firstName: 'John', lastName: 'Doe', age: 31}                     app.js:13
Milon Shah 21                                                              Person.js:22
▶ Person2 {firstName: 'Milon', lastName: 'Shah', fullName: 'Milon Shah', age: 21}   app.js:17
Milon Shah 21 30000                                                        Person.js:36
                                                                           app.js:21
▶ Person3 {firstName: 'Milon', lastName: 'Shah', fullName: 'Milon Shah', age: 21, salary: 300
00}
Hello World                                                                app.js:23
```

[ ]: