```
//hoisting conceept---->default var type
message = 'Hello world'
console.log(message)//Print statement

const messagePrint = 'Hello world'
console.log(messagePrint)

let Message = 'Hello world'
console.log(Message)

var messageee = 'Hello world'
console.log("my data is:",messageee)

const name='Akshaya Natesan'
const rno=11;
console.log("My name is:",name,"and My rollno is:",rno)
//Template Literal
console.log(`My name is:${name} and my rollno is${rno}`)

=================================================================

//Reassigning
var myName='sai'
function disp(){
  myName='ram'
  console.log('im inside the fun:',myName)
  if(true){
    myName='nandhini'
    console.log('im inside the block',myName)
  }
}
 disp()
 console.log('im outside the block',myName)
=====================================================================

//Reassigning
const myName='sai'
function disp(){
  console.log('im inside the fun:',myName)
  if(true){
    const myName='nandhini'
    console.log('im inside the block',myName)
  }
}
 disp()
 console.log('im outside the block',myName)

=====================================================================

//Reassigning
var  myName='sai'
function disp(){
  console.log('im inside the fun:',myName)
  if(false){
    var myName='nandhini'
    console.log('im inside the block',myName)
  }
}
 disp()
```

```
 console.log('im outside the block',myName)
```
==========
Hoisting in JavaScript is a built-in behavior that moves the declarations of variables, functions, and classes to the top
==========
```
const person={
  rno:11,
  stuName:'Akshaya'
}
person.stuName="Akshaya Natesan"
console.log(person)
```
==========
Arrays allows different types of datatypes inside one array.....
==========
If want multiline text we can use template literal(``)
==========
```
a=12345678908765432123456789O
console.log(a)
```

Output:1.2345678908765432e+26

To get the output displayed completely for lengthy numbers use n at the end this represents BigInt
```
a=12345678908765432123456789On
console.log(a)
```

Output:12345678908765432123456789On
==========

```
a=10/0
console.log(a)
b="hai"*3
console.log(b)
```

Output:Infinity
nan

Note:Here we don't get ZeroDivisionError instead we get Infinity as the output....
When string multiplied with an integer we get NaN this means Not a number.....
==========

To check the type of the data (i.e.,) datatype we can use typeof(variablename)

```
var b=null
console.log(typeof(b))
```

Output: object

==========

Boolean
BigInt
Null
Number
Array
Object
Undefined ---> Variable with no value/Variable undeclared
String

==========

```
a=10
b='10'
console.log(a==b)
```

Output: True

if given:a===b then Output will be False since here it checks the datatype also...

```
a=10
b='10'
console.log(a!=b)
```

Output: False

if given:a!==b then Output will be True since here it checks the type of data also...

==========

```
a=2
b=3
console.log(a**b)
```

Output:8
//2*2*2
==========
DeStructuring:

```
const person={
   name:'Akshaya Natesan',
   rollno:11,
   gender:'female'
}

let {name,rollno,gender}=person
console.log(name,' ',rollno,' ',gender)
console.log(person)
```

Output: Akshaya Natesan 11 female
{ name: 'Akshaya Natesan', rollno: 11, gender: 'female' }

```
const person={
   n:'Akshaya Natesan',
   r:11,
   g:'female'
}

let {'n':name,'r':rollno,'g':gender}=person
console.log(name,' ',rollno,' ',gender)
console.log(person)
```

Output: Akshaya Natesan 11 female
{ n: 'Akshaya Natesan', r: 11, g: 'female' }

Note: To skip any specific value separate them by comma and leave a space(Eg: let{name, , , gender}=person

==========

```
const person={
   n:'Akshaya Natesan',
```

```
    r:11,
    g:'female'
}

let {'n':name,'r':rollno,'g':gender,city='Krr'}=person//Assigning default values
console.log(name,' ',rollno,' ',gender,city)
console.log(person)
```

Output:Akshaya Natesan 11 female Krr
{ n: 'Akshaya Natesan', r: 11, g: 'female' }

==========

```
const person={
    n:'Akshaya Natesan',
    r:11,
    g:'female'
}
person['city']='krr'
person.state='TN'

let {'n':name,'r':rollno,'g':gender,city='Krr'}=person//Assigning default values
console.log(name,' ',rollno,' ',gender,city)
console.log(person)
```

Output: Akshaya Natesan 11 female krr
{ n: 'Akshaya Natesan', r: 11, g: 'female', city: 'krr', state: 'TN' }

==========

Rest Operator:a set of three dots (...) that collects multiple elements into an array.....The rest operator can make i
r of arguments. It can also be used to create more versatile and flexible functions. .....

```
const person={
    name:'Akshaya Natesan',
    rollno:11,
    gender:'female'
}

let {name,...restdatas}=person//Destructuring syntax
console.log(name,restdatas)
console.log(person)
```

Output:Akshaya Natesan { rollno: 11, gender: 'female' }
{ name: 'Akshaya Natesan', rollno: 11, gender: 'female' }

Note:Rest operator cannot be given at the begining in a function as an argument it leads to an error....This concept is

==========

```
a=[1,2,3,4]
b=['hai','hello','welcome',...a]
console.log(b)
```

Output: ['hai', 'hello', 'welcome', 1, 2, 3, 4]

```
a=[1,2,3,4]
b=['hai','hello','welcome']
b.push(...a)
```

```
console.log(b)
b=['hai','hello','welcome',a]
console.log(b)
```

Output: ['hai', 'hello', 'welcome', 1, 2, 3, 4]
['hai', 'hello', 'welcome', [1, 2, 3, 4]]

==========

Functions: A JavaScript function is a block of code designed to perform a particular task.
A JavaScript function is executed when "something" invokes it (calls it).

```
function add(){
  console.log("Welcome")
}
add()//NANR
```

Output:Welcome

```
function add(a,b){
  a+b
}
console.log(add(1,2))//WANR
```

Output:undefined

```
function add(a,b){
  c=a+b
  return c
}
console.log(add(1,2))//WAWR
```

Output:3

```
function add(){
  return "Welcome"
}
console.log(add())//NAWR
```

Output:Welcome

==========

```
//a=[1,'sai',3,4.5,2,5]
function add(x,y,...a){
   s=x
   for(i=0;i<a.length;i++){

      s=s+a[i]

   }
   return s
}
res=add(1,'sai',3,4.5,2,5)
console.log(res)
```

Output: 15.5

==========

```
function out(){
   console.log("im inside the out func")
   return function(){
      console.log("im inside the inner func!!!!")
   }
}
inn=out()
inn()
```

Output: im inside the out func
im inside the inner func!!!!

```
function out(){
   console.log("im inside the out func")
   function inner(){
      console.log("im inside the inner func!!!!")
   }
   return inner
}
inn=out()
inn()
```

Output: im inside the out func
im inside the inner func!!!!

Note:Closures concept is used in the above example where an function is assigned to an variable...Returning an func

==========

Function Expression:

A function expression can be stored in a variable:

```
const x = function (a, b) {return a * b};
```

Note:If the function call is present above the definition in an function expression it leads to an error...But incase of fu
efinition...

==========

Arrow Functions:Arrow functions allow us to write shorter function syntax

```
let a=()=>{
   console.log("Welcome")//NANR
}
a()
```

Output:Welcome

```
et a=(x,y)=>{
   x+y
}
console.log(a())//WANR
```

Output:undefined

```
let a=(x,y)=>{
   z=x+y
   return z
```

```
}
console.log(a(1,2))//WAWR
```

Output:3

```
let a=()=>{
   return 'Welcome'
}
console.log(a())
```

Output: Welcome

```
let a=(x,y)=>x*y
console.log(a(1,2))
```

Output:2

Note:If want to specify return keyword use the curly braces{} or use the above example...

==========

```
function get(recCheckFun)
{
   name='sai1'
setTimeout( ()=>{
       if(name==='sai')
       {
          recCheckFun()
       }
   } ,2000)
}
function check()
{
   console.log('pass')
}
get(check)
```

==========

CallBack Functions:A callback is a function passed as an argument to another function
This technique allows a function to call another function
A callback function can run after another function has finished

Sync:

```
function get(recCheckFun)
{
   name='sai'
setTimeout( ()=>{
       if(name==='sai')
       {
          recCheckFun()
       }
   } ,2000)
}
function check()
{
   console.log('pass')
}
get(check)
```

Output:pass

Async:

```
function get(recCheckFun)
{
   name='sai'
setTimeout( ()=>{
       if(name==='sai')
       {
          recCheckFun()
       }
   } ,2000)
   console.log('im in get')
}
function check()
{
   console.log('pass')
}
get(check)
```

Output:im in get
pass

file:///C:/Users/AKSHAYA.N/Pictures/Screenshots/js.png

==========

Anonymous Functions/Nameless Functions:

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

greet(); // Output: Hello, world!
```

==========

Promises:

Pending
Resolve
Reject

"Producing code" is code that can take some time

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

A Promise is an Object that links Producing code and Consuming code

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

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

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

```
);



let res= new Promise((resolve,reject)=>{
    name= undefined
    setTimeout(()=>{
        if(name==='sai')
        {
            resolve(name)
        }
        else
        {
            reject('no data')
        }
    } ,2000)
})
res
.then((name)=>{
   console.log('received: ',name)
})
.catch((errr)=>{
     console.log('pb is : ',errr)
})
.finally(()=>{
    console.log('always i will print : ')
})

Output:pb is :  no data
always i will print :


let res= new Promise((resolve,reject)=>{
    name= 'sai'
    setTimeout(()=>{
        if(name==='sai')
        {
            resolve(name)
        }
        else
        {
            reject('no data')
        }
    } ,2000)
})
res
.then((name)=>{
   console.log('received: ',name)
})
.catch((errr)=>{
     console.log('pb is : ',errr)
})
.finally(()=>{
    console.log('always i will print : ')
})

Output:received:  sai
always i will print :
==========
```

Async/Await:

"async and await make promises easier to write"

async makes a function return a Promise

await makes a function wait for a Promise

```
const res = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const a = 'sai';
            if (a==='sai') {
                resolve(a);
            } else {
                reject(new Error('no data'));
            }
        }, 1000);
    });
};

const handleData = async () => {
    try {
        const name = await res(); // Wait for the promise to resolve
        console.log('received', name); // Handle resolved value
    } catch (err) {
        console.log(err.stack); // Handle error stack
    } finally {
        console.log('received'); // Final message
    }
};

handleData(); // Call the async function
```

Output:received sai
received

```
const res = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const a = 'sai1';
            if (a==='sai') {
                resolve(a);
            } else {
                reject(new Error('no data'));
            }
        }, 1000);
    });
};

const handleData = async () => {
    try {
        const name = await res(); // Wait for the promise to resolve
        console.log('received', name); // Handle resolved value
    } catch (err) {
        console.log(err.stack); // Handle error stack
    } finally {
        console.log('received'); // Final message
    }
```

```
};

handleData(); // Call the async function

Output:ERROR!
Error: no data
    at Timeout._onTimeout (/tmp/bkz7JzI4qb/main.js:17:24)
    at listOnTimeout (node:internal/timers:594:17)
    at process.processTimers (node:internal/timers:529:7)
received

If suppose given err.message then the output would be:
no data
received

================================================================================================

unshift() adds new items to the beginning of an array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.unshift("Lemon", "Pineapple");


Output:Lemon,Pineapple,Banana,Orange,Apple,Mango

shift() removes the first item of an array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.shift();
console.log(fruits)

Output:Orange,Apple,Mango

const fruits = ["Banana", "Orange", "Apple", "Mango"];
console.log(fruits.shift());

Output:Banana

const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Kiwi");

Output: Banana,Orange,Apple,Mango,Kiwi

const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Kiwi", "Lemon");

Output:Banana,Orange,Apple,Mango,Kiwi,Lemon

pop() removes the last element of an array.

const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.pop();
console.log(fruits)

Output:Banana,Orange,Apple

const fruits = ["Banana", "Orange", "Apple", "Mango"];
cosole.log(fruits.pop());

Output: Mango
```

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

Output: [ 'Orange', 'Lemon' ]

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

Output: [ 'Lemon', 'Apple' ]

// Create an Array
const fruits = ["Banana", "Orange", "Apple", "Mango"];

// At position 2, add "Lemon" and "Kiwi":
fruits.splice(2, 0, "Lemon", "Kiwi");

console.log(fruits);

Output: [ 'Banana', 'Orange', 'Lemon', 'Kiwi', 'Apple', 'Mango' ]

// Create an Array
const fruits = ["Banana", "Orange", "Apple", "Mango"];

// At position 2, remove 2 items
fruits.splice(2, 2);

console.log(fruits);

Output: [ 'Banana', 'Orange' ]

// Create an Array
const fruits = ["Banana", "Orange", "Apple", "Mango"];

// At position 2, remove 1 item, add "Lemon" and "Kiwi"
fruits.splice(2, 1, "Lemon", "Kiwi");

console.log(fruits)

Output: [ 'Banana', 'Orange', 'Lemon', 'Kiwi', 'Mango' ] □

================================================================================================

Operators:

Arithmetic: +, -, *, /, %, **
Assignment: =, +=, -=, *=, /=, %=
Comparison: ==, ===, !=, !==, >, <, >=, <=
Logical: &&, ||, !
Bitwise: &, |, ^, ~, <<, >>
Type: typeof
Ternary: condition ? expr1 : expr2
Spread/Rest: ...


let   reassign  redeclare   function   scope
var   reassign  no          block      scope
const no        no          block      scope
```