1. Asynchronous

```
/*
relationship between the now and later parts of your program is at the heart of asynchronous
programming.
*/


function now() {
      return        21;
    }

function later() {
         answer = answer * 2;
         console.log("Meaning of  life:",     answer)
      }

var       answer =           now();
var call = setTimeout(later, 1000)

/*
The       now      chunk runs right away, as soon as you execute your program. But setTimeout(..)
also      sets
up an event (a timeout) to happen later(1000 ms from now).
*/


/*
event loop
Browser have a mechanism in    them that handles executing multiple chunks      of your
          program           over
time, at each moment invoking  the       JS engine, called the "event loop."

scheduled "events"
The       browser           is then  set       up to listen for the response from the network, and
          when it has
something to give you, it schedules the callback function to be, executed by inserting it into
the event loop.
*/

//        `eventLoop`     is        an        array    that     acts     as       a         queue   (first-in,
          first-out)
var       eventLoop =[];
var       event;
```

```
//while (true)
      {
   if       (eventLoop.length >       0) {
          event =   eventLoop.shift();

   try     {
     event();
         }
          catch      (err){
          // reportError(err);
         }
      }
   }


/*
Stander Ajax request dont complate synchronously, the simplest way of "waiting" from now
until later is to use a
function, called callback function.

ajax.("https://googlr.com", function myFunction(data){
   console.log(data);
})
*/
```

2. Parrallel

```
/*
async is about the gap between now and later.But parallel is about things being  able to  occur
simultaneously
*/
import React, { Component } from 'react';


class Parrallel extends Component {
   state = {

   }
   render(){
      return(
         <div>
           {addNumbers(21, 21)}<br/>
           {addNumbers(21, "21")}<br/>                {/*Error: "Bad parameters"*/}
         </div>
      )
```

```
  }
}


export default Parrallel;


//2
var     a =     20;
function foo(){
    a = a +     1;
   }

function bar(){
    a = a *     2;
   }

//      ajax(..)  is      some    arbitrary       Ajax    function        given   by      a
        library

//ajax( "http://some.url.1",    foo     );
//ajax( "http://some.url.2",    bar     );


/*
job queqe
the     asynchronous behavior of Promises is based on Jobs,     so it's important to keep clear
        how     that
relates to      event loop behavior.
*/

console.log("A");
setTimeout(     function(){
  console.log("B");
   },   0);

setTimeout( function(){
  console.log("C");
        setTimeout( function(){
    console.log("D");
  });
 });
 //4
 function          addNumbers(x,y)          {
                   if         (typeof x  != "number" || typeof y != "number"){

              //throw          Error(    "Bad     parameters"      );
             }
          return     x +        y;
      }

addNumbers(21, 21)
addNumbers(21, "21")               /*Error: "Bad parameters"*/
```

3.clouser

```
/*
Most of the JavaScript Developers use closure consciously or unconsciously. Even if they do
unconsciously it works fine in most of the cases. But knowing closure will provide a better
control over the code when using them. And another reason for learning closure is that it is the
most frequently asked question in the interview for the JavaScript developers.
*/

function foo(outer_arg) {

   function inner(inner_arg) {
      return outer_arg + inner_arg;
   }
   return inner;
}
var get_func_inner = foo(5);

console.log(get_func_inner(4));
console.log(get_func_inner(3));


function outer()
{
   function create_Closure(val)
   {
      return function()
      {
         return val;
      }
   }
```

```
    var arr = [];
    var i;
    for (i = 0; i < 4; i++)
    {
        arr[i] = create_Closure(i);
    }
    return arr;
}
var get_arr = outer();
console.log(get_arr[0]());
console.log(get_arr[1]());
console.log(get_arr[2]());
console.log(get_arr[3]());
```

4.Compare Two’s Array

```
var array1 = [1, 2, 3, 4, 5, 6];
    var array2 = [1, 2, 3, 4, 5, 6, 7, 8, 9];

    var result = [];

    var len = array2.length;
    for (var i = 0; i <= len - 1; i++) {
        if (array1.indexOf(array2[i]) == -1) {
            result.push(array2[i]);
        }
    }
    console.log(" the difference is " + result);
```

5.Fibinacci

```
var t1 = 0, t2 = 1, nextTerm = 0;
    console.log(t1);
    console.log(t2);
    for (var i = 0; i <= 100; i++) {
        nextTerm = t1 + t2;
        t1 = t2;
        t2 = nextTerm;
        console.log(nextTerm);
    }
```

6.Loop

```
/*
 Create a for loop that iterates up to 100 while outputting "fizz" at multiples of 3, "buzz"
 at multiples of 5 and "fizzbuzz" at multiples of 3 and 5
```

```
*/


for (var i = 1; i <= 100; i++) {
        if (i % 3 === 0 && i % 5 === 0)
            console.log(i + "== fizzbuzz");
        else if (i % 3 === 0)
            console.log(i + "== fizz");
        else if (i % 5 === 0)
            console.log(i + "== buzz");
            }
```

7. Map

```
/*
Map is a collection of elements where each element is stored as a Key, value pair. Map object
can hold both objects and primitive values as either key or value. When we iterate over the map
object it returns the key,value pair in the same order as inserted.
*/

var map1 = new Map();

map1.set("first name", "sumit");
map1.set("last name", "ghosh");
map1.set("website", "geeksforgeeks")
    .set("friend 1","gourav")
    .set("friend 2","sourav");

console.log(map1);

console.log("map1 has website ? "+
                    map1.has("website"));

console.log("map1 has firend 3 ? " +
                    map1.has("friend 3"));


console.log("get value for key website "+
                    map1.get("website"));

console.log("get value for key friend 3 "+
                    map1.get("friend 3"));

console.log("delete element with key website "
                    + map1.delete("website"));
```

```
console.log("map1 has website ? "+
                    map1.has("website"));

console.log("delete element with key website " +
                    map1.delete("friend 3"));


map1.clear();

console.log(map1);
```

8. Prime

```
var pr = 0;
    for (var p = 2; p <= 100; p++) {
        for (var i = 2; i <= 9; i++) {
            if (i != p) {
                if (p % i == 0) {
                    pr = 0;
                    break;
                } else {
                    pr = 1;
                }
            }
        }
        if (pr == 1)
            console.log(p);
    }
```

9. Scope

```
var globalVar = "This is a global variable";

function fun() {
  var localVar = "This is a local variable";

  console.log(globalVar);
  console.log(localVar);
}

console.log(fun())
```

10. Seconds Largest Element

```
var secondMax = function (arr) {
    var max = Math.max.apply(null, arr);
```

```
  arr.splice(arr.indexOf(max), 1);
  return Math.max.apply(null, arr);
};

var arr = [20, 120, 111, 215, 54, 78];
var max2 = secondMax(arr);
console.log(max2);
```

11. Sets

```
//A set is a collection of items which are unique

Set.prototype.subSet = function(otherSet) {
  if(this.size > otherSet.size)
    return false;
  else
  {
    for(var elem of this)
    {
      if(!otherSet.has(elem))
        return false;
    }
    return true;
  }
}

var setA = new Set([10, 20, 30]);
var setB = new Set([50, 60, 10, 20, 30, 40]);
var setC = new Set([10, 30, 40, 50]);

console.log(setA.subSet(setB));
console.log(setA.subSet(setC));
console.log(setC.subSet(setB));
```

12. Create Objects

```
function copyClass(name, age) {
    this.name = name;
    this.age = age;
    this.printInfo = function() {
      console.log(this.name);
      console.log(this.age);
    }
  }

// Creating the object of copyClass
// and initializing the parameters.
var obj = new copyClass("Vineet", 20);

// Calling the method of copyClass.
obj.printInfo();
```

12.1. Get and Set

```
/*
The get property of the property descriptor is a function that will be called to retrieve the value from the property.
The set property is also a function, it will be called when the property has been assigned a value, and the new value will be passed as an argument.
*/

var person = { name: "John", surname: "Doe"};
Object.defineProperty(person, 'fullName', {
 get: function () {
             return this.name + " " + this.surname;
               },
                set: function (value) {
                 [this.name, this.surname] = value.split(" ");
                  }
                 });
 console.log(person.fullName);
person.surname = "Hill";
 console.log(person.fullName);
person.fullName = "Mary Jones";
 console.log(person.name)

 // Dynamic / variable property names
var dictionary = {
  lettuce: 'a veggie',    banana: 'a fruit',    tomato: 'it depends on who you ask',    apple: 'a fruit',
  Apple: 'Steve Jobs rocks!'
   }
var word = prompt('What word would you like to look up today?')
var definition = dictionary[word]

alert(word + '\n\n' + definition)
console.log(dictionary.word)
```

12.2. Object Literals

```
var obj = {
       name : "",
       age : "",
        printInfo : function() {
            console.log(this.name);
            console.log(this.age);
        }
    }

// Initializing the parameters.
obj.name = "Vineet";
obj.age = 19;

// Using method of the object.
obj.printInfo();
```

12.3. Objects

```
/*
.assign() function can be used to copy all of the enumerable properties from an existing Object
instance to a new one.

*/

const existing = { a: 1, b: 2, c: 3 };
const clone = Object.assign({d:4}, existing)

console.log(clone)

//second

var obj = { 0: 'a', 1: 'b', 2: 'c' };
Object.keys(obj).map(function(key) {
   console.log(key); });

/*
 The Object.assign() method is used to copy the values of all enumerable own properties from
 one or more source objects to a target object. It will return the target object.
*/
var user = {    firstName: "John" };
Object.assign(user, {lastName: "Doe", age:39});
console.log(user);
```

12.4. Singleton using a function

```
var obj = new function() {
        this.name = "";
        this.age = "";
        this.printInfo = function() {
            console.log(this.name);
            console.log(this.age);
        };
    }

// Initializing object.
obj.name = "Vineet";
obj.age = 20;

// Calling method of the object.
obj.printInfo();
```

13. HOF

```
/*
A higher order function is a function either:
1. Accept a function as an argument.
2.Return a function.

HOF are
forEach
map
filter
sort
reduce
*/


document.addEventListener("click", otherFunction);

function otherFunction() {
    alert("calling and using hof");
}
```

13.1. HOF array

```
const companies= [
    {name: "Company One", category: "Finance", start: 1981, end: 2004},
    {name: "Company Two", category: "Retail", start: 1992, end: 2008},
    {name: "Company Three", category: "Auto", start: 1999, end: 2007},
    {name: "Company Four", category: "Retail", start: 1989, end: 2010},
    {name: "Company Five", category: "Technology", start: 2009, end: 2014},
    {name: "Company Six", category: "Finance", start: 1987, end: 2010},
    {name: "Company Seven", category: "Auto", start: 1986, end: 1996},
    {name: "Company Eight", category: "Technology", start: 2011, end: 2016},
    {name: "Company Nine", category: "Retail", start: 1981, end: 1989}
];

const ages = [33, 12, 20, 16, 5, 54, 21, 44, 61, 13, 15, 45, 25, 64, 32];

for(let i = 0; i < companies.length; i++) {
  console.log(companies[i]);
}

// forEach

companies.forEach(function(company) {
  console.log(company.name);
});

// filter

// Get 21 and older

let canDrink = [];
for(let i = 0; i < ages.length; i++) {
  if(ages[i] >= 21) {
    canDrink.push(ages[i]);
  }
}

const canDrink2 = ages.filter(function(age) {
  if(age >= 21) {
    return true;
  }
});

const canDrink3 = ages.filter(age => age >= 21);

// Filter retail companies

const retailCompanies = companies.filter(function(company) {
  if(company.category === 'Retail') {
    return true;
  }
});

const retailCompanies2 = companies.filter(company => company.category === 'Retail');

// Get 80s companies
```

```
  const eightiesCompanies = companies.filter(company => (company.start >= 1980 &&
company.start < 1990));

  // Get companies that lasted 10 years or more

  const lastedTenYears = companies.filter(company => (company.end - company.start >= 10));

  // map

  //Create array of company names
  const companyNames = companies.map(function(company) {
    return company.name;
  });

  const testMap = companies.map(function(company) {
    return `${company.name} [${company.start} - ${company.end}]`;
  });

  const testMap2 = companies.map(company => `${company.name} [${company.start} -
${company.end}]`);

  const ageMap = ages
    .map(age => Math.sqrt(age))
    .map(age => age * 2);



  // sort

  // Sort companies by start year

  const sortedCompanies  = companies.sort(function(c1, c2) {
    if(c1.start > c2.start) {
      return 1;
    } else {
      return -1;
    }
  });

  const sortedCompanies2 = companies.sort((a, b) => (a.start > b.start ? 1 : -1));

  // Sort ages
  const sortAges = ages.sort((a, b) => a - b);
  console.log(sortAges);


  // reduce

  let ageSum = 0;
  for(let i = 0; i < ages.length; i++) {
    ageSum += ages[i];
  }

  const ageSum2 = ages.reduce(function(total, age) {
    return total + age;
  }, 0);

  const ageSum3 = ages.reduce((total, age) => total + age, 0);

  // Get total years for all companies

  const totalYears = companies.reduce(function(total, company) {
    return total + (company.end - company.start);
  }, 0);

  const totalYears2 = companies.reduce((total, company) => total + (company.end -
company.start), 0);

  // Combine Methods

  const combined = ages
    .map(age => age * 2)
    .filter(age => age >= 40)
    .sort((a, b) => a - b)
    .reduce((a, b) => a + b, 0);

  console.log(combined);
```

13.2. HOF array coppy

```
const companies= [
    {name: "Company One", category: "Finance", start: 1981, end: 2004},
    {name: "Company Two", category: "Retail", start: 1992, end: 2008},
    {name: "Company Three", category: "Auto", start: 1999, end: 2007},
    {name: "Company Four", category: "Retail", start: 1989, end: 2010},
    {name: "Company Five", category: "Technology", start: 2009, end: 2014},
    {name: "Company Six", category: "Finance", start: 1987, end: 2010},
  {name: "Company Seven", category: "Auto", start: 1986, end: 1996},
  {name: "Company Eight", category: "Technology", start: 2011, end: 2016},
  {name: "Company Nine", category: "Retail", start: 1981, end: 1989}
];

const ages = [33, 12, 20, 16, 5, 54, 21, 44, 61, 13, 15, 45, 25, 64, 32];

// for(let i = 0; i < companies.length; i++) {
//   console.log(companies[i]);
// }

// forEach

// companies.forEach(function(company) {
//   console.log(company.name);
// });

// filter

// Get 21 and older

// let canDrink = [];
// for(let i = 0; i < ages.length; i++) {
//   if(ages[i] >= 21) {
//     canDrink.push(ages[i]);
//   }
// }

// const canDrink = ages.filter(function(age) {
//   if(age >= 21) {
//     return true;
//   }
// });

const canDrink = ages.filter(age => age >= 21);

// Filter retail companies

// const retailCompanies = companies.filter(function(company) {
//   if(company.category === 'Retail') {
//     return true;
//   }
// });
```

```
  const retailCompanies = companies.filter(company => company.category === 'Retail');

  // Get 80s companies

  const eightiesCompanies = companies.filter(company => (company.start >= 1980 && company.start < 1990));

  // Get companies that lasted 10 years or more

  const lastedTenYears = companies.filter(company => (company.end - company.start >= 10));

  // map

  // Create array of company names
  // const companyNames = companies.map(function(company) {
  //   return company.name;
  // });

  // const testMap = companies.map(function(company) {
  //   return `${company.name} [${company.start} - ${company.end}]`;
  // });

  // const testMap = companies.map(company => `${company.name} [${company.start} - ${company.end}]`);

  // const ageMap = ages
  //   .map(age => Math.sqrt(age))
  //   .map(age => age * 2);



  // sort

  // Sort companies by start year

  // const sortedCompanies  = companies.sort(function(c1, c2) {
  //   if(c1.start > c2.start) {
  //     return 1;
  //   } else {
  //     return -1;
  //   }
  // });

  // const sortedCompanies = companies.sort((a, b) => (a.start > b.start ? 1 : -1));
  // Sort ages
  // const sortAges = ages.sort((a, b) => a - b);

  // console.log(sortAges);


  // reduce

  // let ageSum = 0;
  // for(let i = 0; i < ages.length; i++) {
  //   ageSum += ages[i];
  // }

  // const ageSum = ages.reduce(function(total, age) {
  //   return total + age;
  // }, 0);

  // const ageSum = ages.reduce((total, age) => total + age, 0);

  // Get total years for all companies

  // const totalYears = companies.reduce(function(total, company) {
  //   return total + (company.end - company.start);
  // }, 0);

  const totalYears = companies.reduce((total, company) => total + (company.end - company.start), 0);

  // Combine Methods

  const combined = ages
    .map(age => age * 2)
    .filter(age => age >= 40)
    .sort((a, b) => a - b)
    .reduce((a, b) => a + b, 0);

  console.log(combined);
```

14. OOPS (Dynamic Methods)

```
let METADATA = Symbol('metadata');
class Car {
              constructor(make, model) {
                this.make = make;
```

```
        this.model = model;
       }
       [METADATA]() {
              return {
               make: this.make,
               model: this.model
              };
        }
        ["add"](a, b) {
          return a + b;
           }
        [1 + 2]() {
          return "three";
          }
        }
let MazdaMPV = new Car("Mazda", "MPV");
 MazdaMPV.add(4, 5);
 MazdaMPV[3]();
 console.log(MazdaMPV[METADATA]())
```

14.1. Encapsulation

```
/*
The process of wrapping property and function within a single unit is known as encapsulation.
*/

class Person {
  constructor(name, id) {
    this.name = name;
    this.id = id;
  }
  add_Address(add) {
    this.add = add;
  }
  details() {
    return "Name is = " + this.name +
      ", Student id = " + this.id + ", Address = " + this.add;
  }
}
var person = new Person("Sunny", "14783");
person.add_Address("Delhi");
console.log(person.details());
```

14.2. Inheritance

```
/*
It is a concept in which some property and methods of an Object is being used by another
Object
*/

class Person {
    // Initializing the name
    constructor(name) {
        this.name = name;;
    }

    // toString method returns the name
    toString_Person() {
        return "Name of person = " + this.name;
    }
}

// Defining the student class
// It is the derived class
// It extends Person
class Student extends Person {
    // Initializing the name and id
    constructor(name, Sid) {
        // calling the super class constructor
        super(name);

        // Initializing Sid
        this.Sid = Sid;
    }

    // toString method retuns the student detail
    // Overriding the toString method from base
    // class
    toString_Student() {
        // Calling the toString method of the base
        // class to get the name
        return super.toString_Person() + ", Student Id = "
            + this.Sid
    }
}

// creating Object
var Student_1 = new Student("Sumit", "GFG_123");
```

```
// Printing the name and Sid of Student_1
console.log(Student_1.toString_Student());
```

14.3. Methods

```
class Something {
   constructor(data) {
      this.data = data
        }
   doSomething(text) {
        return {
             data: this.data,           text
              }
             } }
var s = new Something({})
 s.doSomething("hi")
 console.log(s.doSomething("hi"))
```

14.4. Object Constructor

```
function person(first, last) {
   this.firstName = first;
   this.lastName = last;
}

// using prototype to define methods
person.prototype.getDetails = function () {
   return "Person name is " + this.firstName +
      " " + this.lastName;
}


var P1 = new person("Sumit", "Ghosh");
console.log(P1.firstName);
console.log(P1.getDetails());
```

14.5. Object Literals

```
class Employee {
   // Defining connstructor
   // to initialize the property
   constructor(Ename, Eid) {
      this.Ename = Ename;
      this.Eid = Eid;
   }

   // Method returns employee details
```

```
  getDetails() {
    return "Employee name = " + this.Ename +
      ", Employee id = " + this.Eid;
  }
}

// Creating an Employee Object
var Emp1 = new Employee("Sumit", "1234");

// Printing the Employee Details
console.log(Emp1.getDetails());
```

15. Promises

```
/*
Promises sre a time-independent wrapper around a "future value".

Generator can be paused at "yield" point and be resumed asynchronously later.
*/


//1
var promise = new Promise(function(resolve, reject) {
  const x = "geeksforgeeks";
  const y = "geeksforgeeks"
  if(x === y) {
   resolve();
  } else {
   reject();
  }
 });

 promise.
   then(function () {
     console.log('Success, You are a GEEK');
   }).
   catch(function () {
     console.log('Some error has occured');
   });

//2
var promise = new Promise(function(resolve, reject) {
  resolve('Geeks For Geeks');
})
```

```
promise
    .then(function(successMessage) {
      //success handler function is invoked
        console.log(successMessage);
    }, function(errorMessage) {
        console.log(errorMessage);
    })


//3
var promise = new Promise(function(resolve, reject) {
    reject('Promise Rejected')
})

promise
    .then(function(successMessage) {
        console.log(successMessage);
    })
    .catch(function(errorMessage) {
      //error handler function is invoked
        console.log(errorMessage);
    });


//Promise Rejected
var promise = new Promise(function(resolve, reject) {
    throw new Error('Some error has occured')
})

promise
    .then(function(successMessage) {
        console.log(successMessage);
    })
    .catch(function(errorMessage) {
      //error handler function is invoked
        console.log(errorMessage);
    });


//5
```

15.1. Promises2

```
var     foo = Promise.resolve(21)
```

```
 .then(function(v){
  return(v);
});
```

16. Prompt

```
class Popup extends Component {
  render() {
   return (
    <div>
      <h1>{this.props.text}</h1>
     <button onClick={this.props.closePopup}>close me</button>
    </div>
   );
  }
 }



 class Prompt extends React.Component {
  constructor() {
   super();
   this.state = {
    showPopup: false,
    age:''
   };
  }
  togglePopup() {
   this.setState({
    showPopup: !this.state.showPopup
   });
  }
  render() {
   return (
    <div className='app'>
     <h1>Prompt</h1>
     <button onClick={this.togglePopup.bind(this)}>show popup</button>
     <button onClick={() => {alert('javascript aler box?');}}>Alert</button>
     <button onClick={() => {prompt('enter age?');}}>Prompt</button>

     {this.state.showPopup ?
      <Popup
       text='Close Me'
       closePopup={this.togglePopup.bind(this)}
```

```
          />
          : null
        }
      </div>
    );
  }
 };

export default Prompt;
```

17. Proptotypes (Constructor)

```
/*
Functions themselves are not constructors. However,     when you put the new keyword in
          front of
a normal function call, that makes          that function call a "constructor call". In fact, new sort
of hijacks any
normal function and calls          it in a     fashion that constructs  an object.

.constructor is    not       a magic immutable property.     It is non-enumerable, but its value
          is
writable          (can be changed),and     moreover,you can add or          overwrite
(intentionally oraccidentally) a   property of       the       name constructor
on any object in any [[Prototype]] chain, with any value you see fit.
*/

//2
function NothingSpecial() {
  console.log("Don't     mind     me!");
}
var a = new NothingSpecial();
//a;

/*
NothingSpecial          is        just      a         plain     old       normal  function,         but
          when      called    with                new       ,         it        constructs an    object,  almost
          as        a         side-effect,        which     we        happen  to        assign    to
          a         .         The       call      was       a constructor       call,     but
          NothingSpecial                is        not,      in        and       of        itself,   a
          constructor.
Functions          aren't    constructors,       but       function            calls     are       "constructor
          calls"    if        and       only      if                  new                 is used
*/

function Foo(name) {
  this.name = name;
}

Foo.prototype.myName = function () {
  return this.name;
};

var a = new Foo("a");
var b = new Foo("b");


a.myName()
b.myName()
```

17.1. Create Objects

```
var anotherObject = {
  a: 2
};
//     create  an     object  linked  to     `anotherObject`
var myObject = Object.create(anotherObject);

myObject.a
```

17.2. Non- javascript

```
/*
most common non-javascript encounter is the DOM API
document.getElementById('ap)
alert() is provided to your JS program by the browser not by the JS engine itself.

*/
```

18. Scope (Block Scope)

```
var foo = true, baz = 10;
if(foo){
  let bar = 3;

  if(baz > bar){
    console.log(baz);
  }
 }


 var foo2 = true;
```

```
  if(foo){
      var a = 2;
      const b = 3;

      a = 3;
//    b = 4;
  }
```

18.1. Lexical Scope

```
/*
Hide these private details inside the scope of doSomething()
Lecxical scope s the set of rules about how the js engine can look-up a variable and where it will find it.
Lexical scope is defined authore-time, we can't cheat with eval() or with().
*/

function foo(str, a){
    eval(str) //cheating;
     return a + b;
  }

  var b = 2;

  foo("var b = 3;", 1);  //1, 3


  function foo2(str, a){
      "use strict";
    eval(str);
     console.log(a)  //Reference Error: a is not defined
  }

  var b = 2;

  foo2("var b = 3;", 1);



function doSomething(a){
    function doSomethingElse(a){
        return a - 1;
    }
     var b;
     b = a + doSomethingElse(a * 2);
     return ( b * 3);
}

doSomething(2);


  function infiniteLoop() {
      function foo(a) {
          i = 3;
          return a + i;
      }

      for(var i =0; i<10; i++) {
          foo(i * 2)
      }
  }

  //infiniteLoop()



  var obj = {
      count:0,
      cool: function coolFn(){
          if(this.count<3){
              setTimeout(function timer(){
                  this.count++;
                  console.log("awesome");
              }, 1000);
          }
      }
  }

  obj.cool();


  /*
    Arrow-function do not behave at like normal function when it comes to their this binding.
    they discard all the normal rules for this binding, and instead take on the this value of
    their immediate lexical enclosing scope.
  */

  var obj2 = {
    count:0,
    cool: function coolFn(){
        if(this.count<3){
            setTimeout(() => {
                this.count++;
                console.log("awesome");
            }, 1000);
        }
    }
}

obj2.cool();
```

19.String

```
/*
String are just array of character. while the implementation under the cover may or may not use array.
*/

    var a = "foo";
    var b = ["f", "o", "o"];

    //var a2 = a.concate("bar")      //foobar
    //var b2 = b.concat(["b", "a", "r"])     //["f", "o", "o","b", "a", "r"]


    //1
    //.split("")
    //.reverse()
    //.join(          ""          );


    //2
    var    a          =          42.59;
 a.toFixed(        0          );          //          "43"
 a.toFixed(        1          );          //          "42.6"
 a.toFixed(        2          );          //          "42.59"
 a.toFixed(        3          );          //          "42.590"
 a.toFixed(        4          );          //          "42.5900"


 //3
 var       a          =          42.59;
```

```
a.toPrecision(   1        );       //       "4e+1"
a.toPrecision(   2        );       //       "43"
a.toPrecision(   3        );       //       "42.6"
a.toPrecision(   4        );       //       "42.59"
a.toPrecision(   5        );       //       "42.590"
a.toPrecision(   6        );       //       "42.5900"

// 42.toFixed(   3        );  //   SyntaxError
(42).toFixed(    3        );                               //       "42.000"
0.42.toFixed(    3        );                               //       "0.420" 42..toFixed(     3
         );                                 //       "42.000"


//4
var      onethousand    =         1E3;                     //       means  1        *        10^3
var      onemilliononehundredthousand       =        1.1E6;  //       means  1.1      *
         10^6

//0xf3;  //       hexadecimal    for:     243
//0Xf3;  //       ditto
//0363;  //       octal    for:   243

//5
//0.1    +        0.2      ===     0.3;     //       false
//It's   really   close:           0.30000000000000004

var      a        =        1       /        0;                                 //       Infinity
var      a        =        0       /        -3;      //       -0


20. Array
/*
one key difference between Arrays and Array-like Objects is that Array-like objects inherit from
Object.prototype
instead of Array.prototype. This means that Array-like Objects can't access common Array
prototype methods like
forEach(), push(), map(), filter(), and slice():

*/

var house = ["1BHK", 1000, "2BHK", 5000, "RENT", true];

var len=house.length;
for(var i=0;i<len;i++)
```

```
        console.log(house[i])

//Convert Array-like Objects to Arrays in ES6
```

21. Call

```
/*
multiple inheritance. That’s when an object or a class can inherit characteristics from more than one parent. This can be done using one of these 3 methods: call / apply / bind.
*/

let obj = { things: 3 };
let addThings = function (a, b, c) {
   return this.things + a + b + c;
};

console.log(addThings.call(obj, 1, 4, 6));


/*
Apply
 We can pass them as an array.
*/

let obj2 = { things: 3 };
let addThings2 = function (a, b, c) {
   return this.things + a + b + c;
};

let arr = [10, 14, 16];
console.log(addThings2.apply(obj2, arr));


/*
Bind
 Bind works by returning a copy of the function, but with a different context.
*/

let obj3 = { things: 3 };
let addThings3 = function (a, b, c) {
   return this.things + a + b + c;
};

console.log(addThings3.bind(obj3, 1, 4, 6));
```

```
//work
console.log(addThings.bind(obj, 1, 4, 6)());

//We can also pass the arguments like this
console.log( addThings.bind(obj)(1,4,60) )
```

22. CallBack

```
setTimeout(function () {
  console.log("Iwaited  1         second!");
}, 1000)



//2
function wait(message) {
  setTimeout(function timer() {
    console.log(message);
  }, 1000);
}
wait(    "Hello,  closure!"          );
```

23. CallBack Asynchronous

```
/*
In JavaScript, almost anything that has to pull data into your app or push data out will always
be asynchronous because it’s not going to be running in the same thread.

callbacks do not work with try-catch.
*/

const throwError = () => {
  throw "Who made this function?"
}

const someAsyncListener = (callback, ) => {
  setTimeout(callback)
}

// THIS DOES NOT CATCH!
try {
  someAsyncListener(throwError)
}
catch (error) {
  console.log(error)
}
```

```
console.log("I'm alive!");


//To catch an error, you have to move your try-catch to the callback function itself.
const throwError2 = () => {
  try {
    throw "Who made this function?"
  }
  catch (error) {
    console.log(error)
  }
}


//Although, if your callback is synchronous, then you can catch errors using try-catch
const someSyncListener = (callback, ) => {
  callback()
}

try {
  someSyncListener(throwError)
}
catch (error) {
  console.log(error)
}
console .log("I'm alive!")
```

24. CallBack Hell

```
function one() {
  setTimeout(function() {
   console.log('1. First thing setting up second thing');
   setTimeout(function() {
    console.log('2. Second thing setting up third thing');
    setTimeout(function() {
     console.log('3. Third thing setting up fourth thing');
     setTimeout(function() {
      console.log('4. Fourth thing');
     }, 2000);
    }, 2000);
   }, 2000);
  }, 2000);
};

one();
```

25. CallBack Synchronous

```
const callbackSynchronously = (callback,) => {
        callback()
    }

callbackSynchronously( () => {
    console.log("Callback Synchronously");
})


//2
const logToConsole = (data, ) => {
    console.log(data)
}
const logDataSynchronously = () => {
    // Do some processing...
    //logToConsole(data)
}


//3
const getDataSynchronously = () => {
    return "getDataSynchronously data";
}

console.log(getDataSynchronously());


//4
const giveDataSynchronously = (callback3,) => {
    callback3();
}

giveDataSynchronously((data,) => {
    console.log("giveDataSynchronously data");
})

//synchronously callback usally return values and asynchronous callback don't.
```

26. Class

/*

The constructor is a special method that initializes an object created by a class automatically, so each time we need to make a new User, we would have to pass in their username, age and address. One important aspect of classes is, unlike function declarations, classes are hoisted.

This means that you cannot create an object before accessing it, otherwise the code will throw a ReferenceError.

A constructor is a special method for creating and initializing objects that have been created with a specific class. There can only be one constructor. If a class does not have a constructor, a default one will be assigned and used.

The super keyword is used in JavaScript to access and call functions on an object's parent. The super.prop and super[expression] expressions are valid in defining methods for both classes and object literals

If the sub class has a constructor, you will have to call super() first before using the this keyword.

It's also good to remember that classes can only inherit regular objects using the Object.setPrototypeOf() method.

```
*/

class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }

  getArea() {
    return this.width * this.height;
  }
}

class Square extends Rectangle {
  constructor(length) {
    super(length, length);
  }
}
```

27. Clouser

```
function makeAdder(x) {
  function add(y) {
    return y + x;
  };
  return add;
}

var plusOne = makeAdder(1);
var        plusTen=           makeAdder(10);

plusOne(3);
plusOne(41);
plusTen(13);


//2
for (var i = 1; i <= 5; i++){
   setTimeout(function timer() {
      console.log(i);
   }, i * 1000);
}
```

28. Corcion

```
Var a     = "42";
Var b     = Number(a);
a;
b;
```

29. Convert a String to an Array

```
/*
The .split() method splits a string into an array of substrings. By default .split() will break
the string into
substrings on spaces (" "), which is equivalent to calling .split(" ").
*/

var strArray = "StackOverflow".split("");
console.log(strArray)

/*
.splice() to remove a series of elements from an array .splice() accepts two parameters, the starting index, and an optional number of elements to delete. If the second parameter is left out .splice() will remove all elements from the starting index through the end of the array.
*/

var array = [1, 2, 3, 4];
  array.splice(1, 2);
  console.log(array)

//Joining array elements in a string
 console.log(["Hello", " ", "world"].join(""));
```

30. Eval

```
function foo(str, a) {
    eval(str);                      // cheating!
    console.log(a, b);
}
Var b =  2;
foo("var b = 3;", 1);         // 1  3



//2
function foo(str) {
    "use  strict";
    eval(str);
    console.log(a);             // ReferenceError: a is not defined
}
foo("var a = 2"   );
```

31. Filter

```
/*
The filter() method accepts a test function, and returns a new array containing only the
elements
of the original array that pass the test provided.
*/

var people = [
{  id: 1,  name: "John",  age: 28 },
{  id: 2,  name: "Jane",  age: 31 },
{  id: 3,  name: "Peter",  age: 55 }
];

let young = people.filter(person => person.age < 35);
console.log(young)

//seconds
var young2 = people.filter((obj) => {
  var flag = false;
  Object.values(obj).forEach((val) => {
     if(String(val).indexOf("J") > -1) {
          flag = true;
          return;
           }
       });
  if(flag) return obj;
});
console.log(young2)


//thired
var numbers = [5, 32, 43, 4];
let odd = numbers.filter(n => n % 2 !== 0);
console.log(odd)
```

32. Function

```
function printAmount() {
   console.log(amount.toFixed(2));
}

Var amount = 99.99;
printAmount();  //         "99.99"
amount= amount * 2;
printAmount();  //         "199.98
```

33. functional programming

JavaScript has the most important features needed for functional programming:

1. First class functions: The ability to use functions as data values: pass functions as arguments, return functions, and assign functions to variables and object properties. This property allows for higher order functions, which enable partial application, currying, and composition.

2. Anonymous functions and concise lambda syntax: x => x * 2 is a valid function expression in JavaScript. Concise lambdas make it easier to work with higher-order functions.

3. Closures: A closure is the bundling of a function with its lexical environment. Closures are created at function creation time. When a function is defined inside another function, it has access to the variable bindings in the outer function, even after the outer function exits. Closures are how partial applications get their fixed arguments. A fixed argument is an argument bound in the closure scope of a returned function. In add2(1)(2), 1 is a fixed argument in the function returned by add2(1).

2.es6 features

1. Default Parameters in ES6
2. Template Literals in ES6
3. Multi-line Strings in ES6
4. Destructuring Assignment in ES6
5. Enhanced Object Literals in ES6
6. Arrow Functions in ES6

7. Promises in ES6
8. Block-Scoped Constructs Let and Const
9. Classes in ES6
10. Modules in ES6

34. Hosting

```
var a = 2;
foo();
function foo() {
   a = 3;
   console.log(a);
   var a;
}
console.log(a);
```

35. IIFE

```
var x = (function IIFE() {
   return 42;
})();
x;
```

36. Interpolation

```
/*
String Interpolation using Template Literal.
interpolated with these brackets: ${}. And yes, the $ is required.

*/

const names = ['Curly', 'Moe', 'Larry'];
const interpolation = `The Three Stooges were ${names.slice(0, 2).join(', ')} and ${names[2]}.`
console.log(interpolation);
```

37. JSON

```
/*
Parse a string (written in JSON format) and return a JavaScript object:
JSON.stringify(..) utility  to serialize a value to a JSON-compatible string value.
*/

var obj = JSON.parse('{"firstName":"John", "lastName":"Doe"}');
console.log(obj);


//2
```

```
JSON.stringify(undefined); //      undefined
JSON.stringify(function(){}); //    undefined
JSON.stringify(   [1,undefined,function(){},4]); //  "[1,null,null,4]"
JSON.stringify({a:2, b:function(){}}); //     "{"a":2}"


//3
var        a={
   val:[1,2,3],
   toJSON:         function(){
      return        this.val.slice(1);
         }
      };

Var b={
   val:[1,2,3],
   //      probably           incorrect!
   toJSON: function(){
      return        "["        +
         this.val.slice(1).join() +
         "]";
      }
    };

    JSON.stringify( a );     //          "[2,3]"
    JSON.stringify( b );//  ""[2,3]""
```

38. LexicalScope

```
/*
Hide these private details inside the scope of doSomething()
Lecxical scope s the set of rules about how the js engine can look-up a variable and where it will find it.
Lexical scope is defined authore-time, we can't cheat with eval() or with().
*/


function foo(str, a){
   eval(str) //cheating;
    return a + b;
  }

  var b = 2;

  foo("var b = 3;", 1);  //1, 3
  function foo2(str, a){
      "use strict";
    eval(str);
     console.log(a)  //Reference Error: a is not defined
  }

  var b = 2;

  foo2("var b = 3;", 1);



function doSomething(a){
    function doSomethingElse(a){
        return a - 1;
    }
     var b;
     b = a + doSomethingElse(a * 2);
     return ( b * 3);
}

doSomething(2)


  function infiniteLoop() {
      function foo(a) {
          i = 3;
          return a + i;
      }

      for(var i =0; i<10; i++) {
          foo(i * 2)
      }
  }

  //infiniteLoop()



  var obj = {
      count:0,
      cool: function coolFn(){
```

```
     if(this.count<3){
       setTimeout(function timer(){
         this.count++;
         console.log("awesome");
       }, 1000);
     }
   }
 }

 obj.cool();


 /*
  Arrow-function do not behave at like normal function when it comes to their this binding.
  they discard all the normal rules for this binding, and instead take on the this value of
  their immediate lexical enclosing scope.
 */

 var obj2 = {
  count:0,
  cool: function coolFn(){
    if(this.count<3){
      setTimeout(() => {
        this.count++;
        console.log("awesome");
      }, 1000);
    }
  }
}

obj2.cool();
```

39. Modules

```
function User() {
  var username, password;
  function doLogin(user, pw) {
    username = user; password = pw;
    var publicAPI = {
      login: doLogin
    };
    return publicAPI;
  }
  var fred = User();
```

```
    fred.login("fred", "12Battery34!");
}
```

40. Nested Scope

```
function foo() {
    var a = 1;
    function bar() {
        var b = 2;
        function baz() {
            var c = 3;
            console.log(a, b, c);
        }
        baz();
        console.log(a, b);
    }
    bar(); console.log(a);
}

foo();
```

41. Object.freeze

```
/*
Variables declared by const are block scoped and not function scoped like variables declared with var

Object.freeze() takes an object as an argument and returns the same object as an immutable object. This implies that no properties of the object can be added, removed, or changed.

with const varriable declaration,To disable any changes to the object we need Object.freeze().

*/

const user = {
    first_name: 'bolaji',
    last_name: 'ayodeji',
    email: 'hi@bolajiayodeji.com',
    net_worth: 2000
}

Object.freeze(user);
user.last_name = 'Samson'; // this won't work, user is still immutable!
user.net_worth = 983265975975950; // this won't work too, user is still immutable and still broke :
console.log(user);  // user is immutated
```

/*
Objects with nested properties are not actually frozen.
Well, Object.freeze() is a bit shallow, you will need to apply it on nested objects to protect them recursively.

So Object.freeze() doesn't fully freeze an object when it has properties which are nested.

To completely freeze objects and its nested properties, you can write your own library or use already created libraries like Deepfreeze or immutable-js

*/

```
const user2 = {
   first_name: 'bolaji',
   last_name: 'ayodeji',
   contact: {
      email: 'hi@bolajiayodeji.com',
   }
}

Object.freeze(user2);
user2.last_name = 'Samson'; // this won't work, user is still immutable!
user2.contact.email = 'hi7@bolajiayodeji.com';
// this will work because the nested object is not frozen
console.log(user2);
```

43. Promises

/*
Promises save you from callback hell.
We can’t act on them immediately. Only after the promise is kept.

Playing with promises has 2 parts-
1. Creation of Promises
2. Handling of Promises

Promises are used for handling asynchronous operations also called blocking code, examples of which are DB, I/O or API calls

As it can be seen, Promises don’t return values immediately. It waits for the success or failure and then returns accordingly. This lets asynchronous methods return values like synchronous ones.

Instead of returning values right away, async methods supply a promise to return the value.

A promise can be one of these states
1. pending — This is the initial state or state during execution of promise. Neither fulfilled nor rejected.
2. fulfilled — Promise was successful.
3. rejected — Promise failed

Chaining Promises
A promise can be returned to another promise , creating a chain of promises. If one fails all others too. Chaining is very powerful combined with Promise as it gives us the control of the order of events in our code

```
*/

//Creation
new Promise( /* executor */ function (resolve, reject) { });

const Promisee = new Promise((res, rej) => {
  setTimeout(() => {
    res(console.log("Promise resolve"))
  }, 1000)
})

console.log(Promisee)


//Handling and Consuming the Promise

const checkIfDone = () => {
  Promisee.then(ok => {
    console.log(ok)
  })
    .catch(err => {
      console.error(err)
    })
}

console.log(checkIfDone)
/*
```

Running .checkIfDone() will execute the isDone() promise and will wait for it to resolve, using the then callback. If there is an error it will be handled in the catch block.

```
*/
```

44. PromisesChaning

```
new Promise(function (resolve, reject) {
  setTimeout(() => resolve(1), 1000);}
  )
  .then(function (result) {
    alert(result); return result * 3;
  })
  .then(function (result) {
    alert(result); return result * 4;
  }).
  then(function (result) {
    alert(result); return result * 6;
  })
```

45. RestParameter

```
/*
the spread operator takes the array of parameters and spreads them across the arguments in the
function call. But what if we need our function to be able to work with an unknown number of
parameters? That’s where the rest parameter.

The rest parameter syntax allows us to represent an indefinite number of arguments as an
array.

A function's last parameter can be prefixed with ... which will cause all remaining arguments
to be placed within a javascript array. Only the last parameter can be a "rest parameter".
*/

function sum(...theArgs) {
  return theArgs.reduce((previous, current) => {
   return previous + current;
  });
 }

 console.log(sum(1, 2, 3));
 // expected output: 6

console.log(sum(1, 2, 3, 4));


//2
function myFun(a, b, ...manyMoreArgs) {
  console.log("a", a);
  console.log("b", b);
  console.log("manyMoreArgs", manyMoreArgs);
 }

 myFun("one", "two", "three", "four", "five", "six");

 // a, one
 // b, two
 // manyMoreArgs, [three, four, five, six]
```

46. Shallow cloning an array

```
/*
Sometimes, you need to work with an array while ensuring you don't modify the original.
Instead
of a clone method,
arrays have a slice method that lets you perform a shallow copy of any part of an array. Keep in
mind that this only clones the first level. This works well with primitive types, like numbers
and strings, but not objects.
*/

const arrayToClone = [1, 2, 3, 4, 5];
const clone1 = Array.from(arrayToClone);
const clone2 = Array.of(...arrayToClone);
const clone3 = [...arrayToClone]

   console.log(arrayToClone)

// Concatenating Arrays
  var array1 = [1, 2];
  var array2 = [3, 4, 5];
  var array3 = [...array1, ...array2]

  console.log(array3)

//Multiple Arrays
   var array1 = ["a", "b"],
       array2 = ["c", "d"],
       array3 = ["e", "f"],
       array4 = ["g", "h"];

       var arrConc = [...array1, ...array2, ...array3, ...array4]

        console.log(arrConc)

//Without Copying the First Array
```

```
        var longArray = [1, 2, 3, 4, 5, 6, 7, 8],
                shortArray = [9, 10];

                longArray.push(...shortArray)
                console.log(longArray)

/*
  Note that if the second array is too long (>100,000 entries), you may get a stack overflow error
(because of how apply
  works). To be safe, you can iterate instead:
*/

shortArray.forEach(function (elem) {
    longArray.push(elem);
});

/*
  When we have two separate array and we want to make key value pair from that two array, we
can use array's reduce
  function like below
*/

var columns = ["Date", "Number", "Size", "Location", "Age"];
var rows = ["2001", "5", "Big", "Sydney", "25"];
var result =  rows.reduce(function(result, field, index) {
  result[columns[index]] = field;
    return result;
}, {})
console.log(result);

//The filter() method creates an array filled with all array elements that pass a test provided as a
function.
var a=[1, 2, 3, 4, 5].filter(value => value > 2);
console.log(a)

//filter
function startsWithLetterA(str) {
   if(str && str[0].toLowerCase() == 'a') {
          return true
           }
             return false;
              }
var str = 'Since Boolean is a native javascript afunction/constructor that takes';
var strArray = str.split(" ");
var wordsStartsWithA = strArray.filter(startsWithLetterA);

console.log(wordsStartsWithA)
```

47. SpreadOperator

```
/*
 easier way to combine two arrays.
 The spread operator (…) takes the values of arr1 and spreads them across arr2.
*/

const arr = [1, 2, 3, 4];
const arr2 = [...arr, 5, 6, 7, 8, 9, 10];
const result = [...arr, ...arr2];

//console.log(arr2);
console.log(result);


/*
we have a function that takes a number of parameters and we have the parameters we want to pass
it stored in an array. How can we call the function and pass the array of parameters
*/

function spreadPara(num1, num2, num3) {
   console.log(num1 + num2 + num3);
}

let params = [31, 4, 57];
spreadPara.apply(null, params);
```

48. Spred

```
const [b,c, ...xs] = [2, 3, 4, 5];
 console.log(b, c, xs)
```

49. String

```
function func() {

   var str = 'It is a great day.';
   var sub_str = str.substr(5);
   console.log(sub_str);
}

func();
```

50. TemplateLiterals

```
/*
it's declarative.
Take nested quotation marks for instance. Typically, if you wanted to create nested quotes, you
would have to escape the quotation characters so the interpreter wouldn’t accidentally end the
string early or switch between double and single quotes.

The only places that template literals will get you into trouble is with ESLint, JSON and
'use strict'. We still need to use quotes for those.
*/

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

51. Void

```
Var a =  42;
console.log(void a, a);        // undefined  42
```

52. WeakMap

```
/*
 ability to have weak references used in the form of a WeakSet and WeakMap.

 WeakMap
 A WeakMap is similar to an object where the keys in that object are actually a WeakSet.
*/

const requests = new WeakSet();
class Request {
  constructor() {
    requests.add(this);
  }
  makeRequest() {
    if (!requests.has(this)) {
      throw new Error("Invalid access");
    }   // Do work...
  }
}


//weakMap
const requests2 = new WeakSet();
class Request2 {
  constructor() {
    requests.set(this, {
      created: new Date()
    });
  }

  makeRequest() {
    if (requestIsTooOld(this)) {
      throw new Error("Try again?");
    }   // Do work...
  }
}
```

53. While

```
while(numOfCustomers >0) {
              console.log("How may   I help you?");
              numOfCustomers = numOfCustomers  - 1;
              }

        do{
              console.log("How may I help you?");
              numOfCustomers  =  numOfCustomers   -  1;
              }

while    (numOfCustomers > 0);
```

54. With

```
function foo(obj) {
  with (obj) {
    a = 2;
  }
}
Var o1  = { a: 3 };
Var o2  = { b: 3 };
foo(o1); console.log(o1.a);        //       2
foo(o2); console.log(o2.a);        //       undefined
console.log(a);                    //       2        --       Oops,   leaked  global!
```

55. Windows

```
var       a = 2;
(function IIFE(def) {
  def(window);
})(function def(global) {
  var a = 3;
```

```
    console.log(a);            //          3
    console.log(global.a); //          2
});
```

56.