1. What is hoisting?

Ans: JavaScript Hoisting refers to the process whereby the interpreter appears to move the declaration of functions, variables or classes to the top of their scope, prior to execution of the code.

Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope.

2. What is scoping?

Ans: Scope in JavaScript refers to the current context of code, which determines the accessibility of variables to JavaScript. The two types of scope are local and global:

1. Global variables are those declared outside of a block
2. Local variables are those declared inside of a block

3. How are var, let const different?

Ans: var declarations are globally scoped or function scoped while let and const are block scoped.

var variables can be updated and re-declared within its scope; let variables can be updated but not re-declared; const variables can neither be updated nor re-declared.

They are all hoisted to the top of their scope. But while var variables are initialized with undefined, let and const variables are not initialized.

While var and let can be declared without being initialized, const must be initialized during declaration.

4. What are the two main differences in arrow functions?

Ans: 1. this value

1.1 Regular function

Inside of a regular JavaScript function, this value (aka the execution context) is dynamic.

The dynamic context means that the value of this depends on how the function is invoked. In JavaScript, there are 4 ways you can invoke a regular function.

During a simple invocation the value of this equals to the global object (or undefined if the function runs in strict mode):

```
function myFunction() {
  console.log(this);
}
```

```
// Simple invocation
myFunction(); // logs global object (window)
```

During a method invocation the value of this is the object owning the method:

```
const myObject = {
  method() {
    console.log(this);
  }
};
// Method invocation
myObject.method(); // logs myObject
```

During an indirect invocation using myFunc.call(thisVal, arg1, ..., argN) or myFunc.apply(thisVal, [arg1, ..., argN]) the value of this equals to the first argument:

```
function myFunction() {
  console.log(this);
}
const myContext = { value: 'A' };
myFunction.call(myContext);  // logs { value: 'A' }
myFunction.apply(myContext); // logs { value: 'A' }
```

During a constructor invocation using new keyword this equals to the newly created instance:

```
function MyFunction() {
  console.log(this);
}
new MyFunction(); // logs an instance of MyFunction
```

1.2 Arrow function

The behavior of this inside of an arrow function differs considerably from the regular function's this behavior. The arrow function doesn't define its own execution context.

No matter how or where being executed, this value inside of an arrow function always equals this value from the outer function. In other words, the arrow function resolves this lexically.

In the following example, myMethod() is an outer function of callback() arrow function:

```
const myObject = {
  myMethod(items) {
    console.log(this); // logs myObject
    const callback = () => {
      console.log(this); // logs myObject
    };
    items.forEach(callback);
  }
};
myObject.myMethod([1, 2, 3]);
```

this value inside the arrow function callback() equals to this of the outer function myMethod().

this resolved lexically is one of the great features of arrow functions. When using callbacks inside methods you are sure the arrow function doesn't define its own this: no more const self = this or callback.bind(this) workarounds.

Contrary to a regular function, the indirect invocation of an arrow function using myArrowFunc.call(thisVal) or myArrowFunc.apply(thisVal) doesn't change the value of this: the context value is always resolved lexically.

2. Constructors

2.1 Regular function

As seen in the previous section, the regular function can easily construct objects.

For example, the new Car() function creates instances of a car:

```
function Car(color) {
  this.color = color;
}
```

```
const redCar = new Car('red');

redCar instanceof Car; // => true
```

Car is a regular function. When invoked with new keyword new Car('red') — new instances of Car type are created.

2.2 Arrow function

A consequence of this resolved lexically is that an arrow function cannot be used as a constructor.

If you try to invoke an arrow function prefixed with new keyword, JavaScrip throws an error:

```
const Car = (color) => {
  this.color = color;
};
```

const redCar = new Car('red'); // TypeError: Car is not a constructor

Invoking new Car('red'), where Car is an arrow function, throws TypeError: Car is not a constructor.

3. arguments object

3.1 Regular function

Inside the body of a regular function, arguments is a special array-like object containing the list of arguments with which the function has been invoked.

Let's invoke myFunction() function with 2 arguments:

```
function myFunction() {
  console.log(arguments);
}

myFunction('a', 'b'); // logs { 0: 'a', 1: 'b', length: 2 }
```

Inside of myFunction() body the arguments is an array-like object containing the invocation arguments: 'a' and 'b'.

3.2 Arrow function

On the other side, no arguments special keyword is defined inside an arrow function.

Again (same as with this value), the arguments object is resolved lexically: the arrow function accesses arguments from the outer function.

Let's try to access arguments inside of an arrow function:

```
function myRegularFunction() {
  const myArrowFunction = () => {
    console.log(arguments);
  }
  myArrowFunction('c', 'd');
}
myRegularFunction('a', 'b'); // logs { 0: 'a', 1: 'b', length: 2 }
```

The arrow function myArrowFunction() is invoked with the arguments 'c', 'd'. Still, inside of its body, arguments object equals to the arguments of myRegularFunction() invocation: 'a', 'b'.

If you'd like to access the direct arguments of the arrow function, then you can use the rest parameters feature:

```
function myRegularFunction() {
  const myArrowFunction = (...args) => {
    console.log(args);
  }
  myArrowFunction('c', 'd');
}
myRegularFunction('a', 'b'); // logs ['c', 'd']
```

...args rest parameter collects the execution arguments of the arrow function: ['c', 'd'].

4. Methods

4.1 Regular function

The regular functions are the usual way to define methods on classes.

In the following class Hero, the method logName() is defined using a regular function:

```
class Hero {
  constructor(heroName) {
```

```
    this.heroName = heroName;
  }
  logName() {
    console.log(this.heroName);
  }
}
const batman = new Hero('Batman');
```

Usually, the regular functions as methods are the way to go.

Sometimes you'd need to supply the method as a callback, for example to setTimeout() or an event listener. In such cases, you might encounter difficulties accessing this value.

For example, let's use use logName() method as a callback to setTimeout():

```
setTimeout(batman.logName, 1000);
// after 1 second logs "undefined"
```

After 1 second, undefined is logged to console. setTimeout() performs a simple invocation of logName (where this is the global object). That's when the method is separated from the object.

Let's bind this value manually to the right context:

```
setTimeout(batman.logName.bind(batman), 1000);
// after 1 second logs "Batman"
```

batman.logName.bind(batman) binds this value to batman instance. Now you're sure that the method doesn't lose the context.

Binding this manually requires boilerplate code, especially if you have lots of methods. There's a better way: the arrow functions as a class field.

4.2 Arrow function

Thanks to Class fields proposal (at this moment at stage 3) you can use the arrow function as methods inside classes.

Now, in contrast with regular functions, the method defined using an arrow binds this lexically to the class instance.

Let's use the arrow function as a field:

```
class Hero {
  constructor(heroName) {
```

```
    this.heroName = heroName;
  }
  logName = () => {
    console.log(this.heroName);
  }
}
const batman = new Hero('Batman');
```

Now you can use batman.logName as a callback without any manual binding of this. The value of this inside logName() method is always the class instance:

```
setTimeout(batman.logName, 1000);
// after 1 second logs "Batman"
```

5. Does Call apply bind work for arrow functions?

Ans: In case of arrow functions our methods: Call/Apply & Bind doesn't work as expected.

"Since arrow functions do not have their own this, the methods call() or apply() can only pass in parameters. thisArg is ignored."

Arrow functions doesn't have their own this. This is lexically bound and it uses the this of the context in which the arrow function was called. Call/Apply & Bind can be used only to pass parameters.

6. What does call apply bind do?

Ans: We use call, bind and apply methods to set the this keyword independent of how the function is called. This is especially useful for the callbacks.

Bind( ): The bind method creates a new function and sets the this keyword to the specified object.

```
Syntax: function.bind(thisArg, optionalArguments)
```

For example: Let's suppose we have two person objects.

```
const john = {
  name: 'John',
  age: 24,
};
```

```
const jane = {
  name: 'Jane',
  age: 22,
};
```

Let's add a greeting function:

```
function greeting() {
  console.log(`Hi, I am ${this.name} and I am ${this.age} years old`);
}
```

We can use the bind method on the greeting function to bind the this keyword to john and jane objects. For example:

```
const greetingJohn = greeting.bind(john);
// Hi, I am John and I am 24 years old
greetingJohn();
const greetingJane = greeting.bind(jane);
// Hi, I am Jane and I am 22 years old
greetingJane();
```

Here greeting.bind(john) creates a new function with this set to john object, which we then assign to greetingJohn variable.

Call ( ): The call method sets the this inside the function and immediately executes that function.

The difference between call() and bind() is that the call() sets the this keyword and executes the function immediately and it does not create a new copy of the function, while the bind() creates a copy of that function and sets the this keyword.

```
Syntax: function.call(thisArg, arg1, agr2, ...)
```

For example:

```
function greeting() {
  console.log(`Hi, I am ${this.name} and I am ${this.age} years old`);
}
const john = {
  name: 'John',
```

```
  age: 24,
};
const jane = {
  name: 'Jane',
  age: 22,
};
// Hi, I am John and I am 24 years old
greeting.call(john);
// Hi, I am Jane and I am 22 years old
greeting.call(jane);
```

Above example is similar to the bind() example except that call() does not create a new function. We are directly setting the this keyword using call().

Apply ( ): The apply() method is similar to call(). The difference is that the apply() method accepts an array of arguments instead of comma separated values.

```
Syntax: function.apply(thisArg, [argumentsArr])
```

For example:

```
function greet(greeting, lang) {
  console.log(lang);
  console.log(`${greeting}, I am ${this.name} and I am ${this.age} years old`);
}
const john = {
  name: 'John',
  age: 24,
};
const jane = {
  name: 'Jane',
  age: 22,
};
```

```
// Hi, I am John and I am 24 years old

greet.apply(john, ['Hi', 'en']);

// Hi, I am Jane and I am 22 years old

greet.apply(jane, ['Hola', 'es']);
```

7. What are closures?

Ans: A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function.

Ex.

```
function makeFunc() {
  var name = 'Mozilla';
  function displayName() {
    alert(name);
  }
  return displayName;
}


var myFunc = makeFunc();

myFunc();
```

displayName() inner function is returned from the outer function before being executed.

Once makeFunc() finishes executing, you might expect that the name variable would no longer be accessible. However, because the code still works as expected, this is obviously not the case in JavaScript.

The reason is that functions in JavaScript form closures. A closure is the combination of a function and the lexical environment within which that function was declared. This environment consists of any local variables that were in-scope at the time the closure was created. In this case, myFunc is a reference to the instance of the function displayName that is created when makeFunc is run. The instance of displayName maintains a reference to its lexical environment, within which the variable name exists. For this reason, when myFunc is invoked, the variable name remains available for use, and "Mozilla" is passed to alert.

8. Write a program to debounce a search bar?

Ans: Debouncing in JavaScript is a practice used to improve browser performance. There might be some functionality in a web page which requires time-consuming computations. If such a method is invoked frequently, it might greatly affect the performance of the browser, as JavaScript is a single threaded language. Debouncing is a programming practice used to ensure that time-consuming tasks do not fire so often, that it stalls the performance of the web page. In other words, it limits the rate at which a function gets invoked.

```
var timerId;
var searchBoxDom = document.getElementById('search-box');

// This represents a very heavy method. Which takes a lot of time to execute
function makeAPICall() {
        var debounceDom = document.getElementById('debounce-count');
        var debounceCount = debounceDom.innerHTML || 0;

        debounceDom.innerHTML = parseInt(debounceCount) + 1
}

// Debounce function: Input as function which needs to be debounced and delay is the debounced time in milliseconds
var debounceFunction = function (func, delay) {
        // Cancels the setTimeout method execution
        clearTimeout(timerId)

        // Executes the func after delay time.
        timerId = setTimeout(func, delay)
}

// Event listener on the input box
searchBoxDom.addEventListener('input', function () {
        var apiCallCountDom = document.getElementById('show-api-call-count');
```

```
    var apiCallCount = apiCallCountDom.innerHTML || 0;
    apiCallCount = parseInt(apiCallCount) + 1;

    // Updates the number of times makeAPICall method is called
    apiCallCountDom.innerHTML = apiCallCount;

    // Debounces makeAPICall method
    debounceFunction(makeAPICall, 200)
})
```

9. Write a program to throttle a search bar?

Ans: Throttling is a technique in which, no matter how many times the user fires the event, the attached function will be executed only once in a given time interval.

```
var timerId;
var divBodyDom = document.getElementById('div-body');

// This represents a very heavy method which takes a lot of time to execute
function makeAPICall() {
    var debounceDom = document.getElementById('debounc-count');
    var debounceCount = debounceDom.innerHTML || 0;

    debounceDom.innerHTML = parseInt(debounceCount) + 1
}

// Throttle function: Input as function which needs to be throttled and delay is the time interval in milliseconds
var throttleFunction = function (func, delay) {
    // If setTimeout is already scheduled, no need to do anything
    if (timerId) {
```

```
                return
        }

        // Schedule a setTimeout after delay seconds
        timerId = setTimeout(function () {
                func()

                // Once setTimeout function execution is finished, timerId = undefined so that in <br>
                // the next scroll event function execution can be scheduled by the setTimeout
                timerId = undefined;
        }, delay)
}

// Event listener on the input box
divBodyDom.addEventListener('scroll', function () {
        var apiCallCountDom = document.getElementById('show-api-call-count');
        var apiCallCount = apiCallCountDom.innerHTML || 0;
        apiCallCount = parseInt(apiCallCount) + 1;

        // Updates the number of times makeAPICall method is called
        apiCallCountDom.innerHTML = apiCallCount;

        // Throttles makeAPICall method such that it is called once in every 200 milliseconds
        throttleFunction(makeAPICall, 200)
})
```

10. create a custom method for an array called myMap, use prototype chain to achieve this

Ans:

```
Array.prototype.myMap = function(callback){
   let result = [];
   for(let index = 0; index < this.length; index++){
      result.push( callback(this[index]) )
   }
   return result;
}
```

11. What is event bubbling?

Ans: When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.

While developing a webpage or a website via JavaScript, the concept of event bubbling is used where the event handlers are invoked when one element is nested on to the other element and are part of the same event. This technique or method is known as Event Bubbling. Thus, while performing event flow for a web page, event bubbling is used. We can understand event bubbling as a sequence of calling the event handlers when one element is nested in another element, and both the elements have registered listeners for the same event. So beginning from the deepest element to its parents covering all its ancestors on the way to top to bottom, calling is performed.

Stopping Bubbling

Beginning from the target and moving towards the top is the bubbling i.e. starting from the child to its parent, it moves straight upwards. But a handler can also take decision to stop the bubbling when the event has been processed completely. In JavaScript, we use the event.stopPropagation () method.

12. What is event loop?

Ans: Event loop: An event loop is something that pulls stuff out of the queue and places it onto the function execution stack whenever the function stack becomes empty.

The event loop is the secret by which JavaScript gives us an illusion of being multithreaded even though it is single-threaded. The below illusion demonstrates the functioning of event loop well:

Here the callback function in the event queue has not yet run and is waiting for its time into the stack when the SetTimeOut() is being executed and the Web API is making the mentioned wait. When the function stack becomes empty, the function gets loaded onto the stack.

That is where the event loop comes into picture, it takes the first event from the Event Queue and places it onto the stack i.e in this case the callback function. From here, this function executes calling

other functions inside it, if any. This cycle is called the event loop and this how JavaScript manages its events.

13. Write a function called sleep that will return a promise, if you do not provide a number to the function, then it will return an error and goto the catch block.

Ans:

```
function sleep(delay){
   return new Promise(function(resolve, reject){
      if(typeof delay !== "number"){
         reject("Please pass number");
      }
     If(!delay){
          reject("Please pass number");
    }

      setTimeout(function(){
         resolve("success");
      }, delay);
   });
}
```

14. Explain promises to a 5 year old, with simple examples

Ans: The Jack and Jill Story

The "Jack and Jill Went Up the Hill..." rhyme has two primary characters, Jack the small boy and his sister Jill. Let's twist the story. Let's introduce their grandparents.

So, Jack & Jill promise their grandparents to fetch some water from the well at the top of the hill. They started on their mission to get it. In the meantime, the grandparents are busy discussing the daily routine, and they want to start cooking once the kids are back with the water.

Now there are two possibilities,

Jack and Jill come down with the water, and the cooking starts.

"Jack fell down and broke his crown. And Jill came tumbling after." - In this case, Jack and Jill return, but unfortunately, they do not get the water.

In this short story, there is a promise of getting the water using the activity of fetching it. The promise can get fulfilled(getting the water) by the kids or reject due to the disaster. Please note, while Jack and Jill were working on executing the promise, the grandparents were not sitting idle. They were planning the day.

The JavaScript promises also work similarly. As developers, we create them to fetch something(data from a data store, configurations, and many more). Usually, the fetching may not happen instantly. We want to fetch things asynchronously. It means we do not want the application to wait for the response, but we can continue to work on the response when it is available.A promise object has the following internal properties,

state: This property can have the following values,

1. pending: When the execution function starts. In our story, when Jack and Jill start to fetch the water.
2. fulfilled: When the promise resolves successfully. Like, Jack and Jill are back with the water.
3. rejected: When the promise rejects. Example. Jack and Jill couldn't complete the mission.

result: This property can have the following values,

1. undefined: Initially, when the state value is pending.
2. value: When the promise is resolved(value).
3. error: When the promise is rejected.

A promise that is either resolved or rejected is called settled.

Q15. what does async await mean?

Ans: An async function is a function declared with the async keyword, and the await keyword is permitted within it. The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

Q16. What does thethis keyword mean?

Ans: This” keyword refers to an object that is executing the current piece of code. It references the object that is executing the current function. If the function being referenced is a regular function, “this” references the global object.

Q17. What are classes? what are getters and setters?

Ans: In a JavaScript class, getters and setters are used to get or set the properties values. “get” is the keyword utilized to define a getter method for retrieving the property value, whereas “set” defines a setter method for changing the value of a specific property.

Q18. How do you declare private and static variables in classes

Ans: Private instance fields are declared with # names, which are identifiers prefixed with #. The # is a part of the name itself. Private fields are accessible on the class constructor from inside the class declaration itself.

Static variable in JavaScript: We used the static keyword to make a variable static just like the constant variable is defined using the const keyword. It is set at the run time and such type of variable works as a global variable.

Q19. What is currying?

Ans: Currying is an advanced technique of working with functions. It’s used not only in JavaScript, but in other languages as well.

Currying is a transformation of functions that translates a function from callable as f(a, b, c) into callable as f(a)(b)(c).

Currying doesn’t call a function. It just transforms it.

Q20. Write a program to flatten an array

Ans:

```
const arr = [ 1, [ 2, 3 ], [ 3 ], [ [ [ 5]],  6]  ]


const flatArr = (arr, depth = 1) => {


  let result = []
  arr.forEach((ar) => {
    if(Array.isArray(ar)) {
      result.push(...flatArr(ar, depth-1))
```

```
    }
    else {
        result.push(ar)
    }
  });
  return result
}


console.log(flatArr(arr));
```