JavaScript Basics

1. Introduction

Q 1.1. List out important features of JavaScript ES6?

1. Template Strings:

Template literals are string literals allowing embedded expressions.

Benefits:

- String interpolation
- Embedded expressions
- Multiline strings without hacks
- String formatting
- String tagging for safe HTML escaping, localization and more

```
// String Substitution
let name = `Abhinav Sharma`;
console.log(`Hi, ${name}`); // Output: "Hi, Abhinav Sharma"

// Multiline String
let msg = `Hello \n
World`;
console.log(`${msg}`); // Output: "Hello World"
```

☆ **Try this example on CodeSandbox**

2. Spread Operator:

Spread operator allows iterables(arrays / objects / strings) to be expanded into single arguments/elements.

```
function sum(x, y, z) {
  return x + y + z;
```

```
}
const numbers = [10, 20, 30];

// Using Apply (ES5)
console.log(sum.apply(null, numbers)); // 60

// Using Spread Operator
console.log(sum(...numbers)); // 60
```

🌟 **Try this example on CodeSandbox**

2.1. Copying an array:

```
let fruits = ["Apple", "Orange", "Banana"];
let newFruitArray = [...fruits];

console.log(newFruitArray);
```

Output:

```
['Apple', 'Orange', 'Banana']
```

🌟 **Try this example on CodeSandbox**

2.2. Concatenating arrays:

```
let arr1 = ["A", "B", "C"];
let arr2 = ["X", "Y", "Z"];

let result = [...arr1, ...arr2];

console.log(result);
```

Output:

```
['A', 'B', 'C', 'X', 'Y', 'Z']
```

🌟 **Try this example on CodeSandbox**

2.3. Spreading elements together with an individual element:

```
let fruits = ["Apple", "Orange", "Banana"];
let newFruits = ["Cherry", ...fruits];

console.log(newFruits);
```

Output:

```
['Cherry', 'Apple', 'Orange', 'Banana']
```

🌟 **Try this example on CodeSandbox**

2.4. Spreading elements on function calls:

```
let fruits = ["Apple", "Orange", "Banana"];

const getFruits = (f1, f2, f3) => {
  console.log(`Fruits: ${f1}, ${f2} and ${f3}`);
};

getFruits(...fruits);
```

Output:

```
Fruits: Apple, Orange and Banana
```

🌟 **Try this example on CodeSandbox**

2.5. Spread syntax for object literals:

```
const obj1 = { id: 101, name: 'Rajiv Sandal' }
const obj2 = { age: 35, country: 'INDIA' }

const employee = { ...obj1, ...obj2 }

console.log(employee);
```

Output:

```
{
  "id": 101,
  "name": "Rajiv Sandal",
  "age": 35,
  "country": "INDIA"
}
```

3. Sets:

Sets are a new object type with ES6 (ES2015) that allow to create collections of unique values. The values in a set can be either simple primitives like strings or

integers, but more complex object types like object literals or arrays can also be part of a set.

```
let numbers = new Set([10, 20, 20, 30, 40, 50]);

console.log(numbers); Set(5) { 10, 20, 30, 40, 50 }
console.log(typeof numbers); // Object
```

🌟 **Try this example on CodeSandbox**

4. Default Parametrs:

```
function add(x = 10, y = 20) {
  console.log(x + y);
}

add(10, 30); // 40
```

🌟 **Try this example on CodeSandbox**

5. repeat():

The `repeat()` method constructs and returns a new string which contains the specified number of copies of the string on which it was called, concatenated together.

```
const msg = "Hello World \n";

console.log(`${msg.repeat(3)}`);
```

Output:

```
Hello World
Hello World
Hello World
```

🌟 **Try this example on CodeSandbox**

6. Arrow Function (=>):

```
let add = (x, y) => x + y;

console.log(add(10, 20)); // 30
```

🌟 **Try this example on CodeSandbox**

7. Arrow function with `this`

```
/**
 * Using Arrow function
 */
const person = {
  name: "Diksha",
  actions: ["bike", "hike", "ski"],
  printActions() {
    this.actions.forEach((action) => {
      console.log(this.name + " likes to " + action);
    });
  },
};

person.printActions();
```

Output:

```
Diksha likes to bike
Diksha likes to hike
Diksha likes to ski
```

🌟 **Try this example on CodeSandbox**

8. Destructing Assignment:

```
// Destructing Assignment
const { title, price, description } = {
  title: "iPhone",
  price: 999,
  description: "The iPhone is a smartphone developed by Apple"
};

console.log(title); // iPhone
console.log(price); // 999
console.log(description); // The iPhone is a smartphone developed by Apple
```

🌟 **Try this example on CodeSandbox**

9. Generators:

A generator is a function that can stop midway and then continue from where it stopped. In short, a generator appears to be a function but it behaves like an `iterator`.

```
function* generator(num) {
  yield num + 10;
  yield num + 20;
```

```
  yield num + 30;
}
let gen = generator(10);

console.log(gen.next().value); // 20
console.log(gen.next().value); // 30
console.log(gen.next().value); // 40
```

⭐ **Try this example on CodeSandbox**

10. Symbols:

They are tokens that serve as unique IDs. We create symbols via the factory function Symbol(). Symbols primary use case is for making private object properties, which can be only of type String or Symbol (Numbers are automatically converted to Strings).

```
const symbol1 = Symbol();
const symbol2 = Symbol(42);
const symbol3 = Symbol("Hi");

console.log(typeof symbol1); // symbol
console.log(symbol3.toString()); // Symbol(Hi)
console.log(Symbol("Hi") === Symbol("Hi")); // false
```

⭐ **Try this example on CodeSandbox**

11. Iterator:

The iterable is a interface that specifies that an object can be accessible if it implements a method who is key is `[symbol.iterator]`.

```
const title = "ES6";
const iterateIt = title[Symbol.iterator]();

console.log(iterateIt.next().value); //output: E
console.log(iterateIt.next().value); //output: S
console.log(iterateIt.next().value); //output: 6
```

⭐ **Try this example on CodeSandbox**

2. VARIABLES

Q 2.1. What are global variables?

Global variables are declared outside of a function or declared with a window object for accessibility throughout the program (unless shadowed by locals). If you declare a variable without using var, even if it's inside a function, it will still be seen as global.

The `var` **statement** declares a function-scoped or globally-scoped variable, optionally initializing it to a value.
Example:

```
var x = 10;

if (x === 10) {
  var x = 20;

  console.log(x); // expected output: 20
}

console.log(x); // expected output: 20
```

Example: Declaring global variable within function

```
window.value = 90;

// Declaring global variable by window object
function setValue() {
  window.value = 100;
}

// Accessing global variable from other function
function getValue() {
  setValue();
  return window.value;
}

console.log(getValue()); // 100
```

Using Undeclared Variables:

- In strict mode, if you attempt to use an undeclared variable, you'll get a reference error when you run your code.
- Outside of strict mode, however, if you assign a value to a name that has not been declared with `let`, `const`, or `var`, you'll end up creating a new global variable. It will be global no matter how deeply nested within functions and blocks your code is, which is almost certainly not what you want, is bug-prone, and is one of the best reasons for using strict mode!
- Global variables created in this accidental way are like global variables declared with `var`: they define properties of the global object. But unlike the

properties defined by proper var declarations, these properties can be deleted with the delete operator.

☆ **Try this example on CodeSandbox**

Q 2.2. What are template literals in es6?

Template literals help make it simple to do string interpolation, or to include variables in a string.

```
const person = { name: 'Tyler', age: 28 };

console.log(`Hi, my name is ${person.name} and I am ${person.age} years old!`);
// 'Hi, my name is Tyler and I am 28 years old!'
```

Template literals, however, preserve whatever spacing you add to them. For example, to create that same multi-line output that we created above, you can simply do:

```
console.log(`This is line one.
This is line two.`);
// This is line one.
// This is line two.
```

Another use case of template literals would be to use as a substitute for templating libraries for simple variable interpolations:

```
const person = { name: 'Tyler', age: 28 };

document.body.innerHTML = `
  <div>
    <p>Name: ${person.name}</p>
    <p>Name: ${person.age}</p>
  </div>
`
```

Q 2.3. What are the differences between variables created using `let`, `var` or `const`?

Variables declared using the `var` keyword are scoped to the function in which they are created, or if created outside of any function, to the global object. `let` and `const` are *block scoped*, meaning they are only accessible within the nearest set of curly braces (function, if-else block, or for-loop).

```
/**
 * All variables are accessible within functions.
 */
function variableScope() {

  var x = 10;
  let y = 20;
  const z = 30;

  console.log(x); // 10
  console.log(y); // 20
  console.log(z); // 30
}

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

variableScope();
/**
 * var declared variables are accessible anywhere in the function scope.
 */
if (true) {
  var a = 10;
  let b = 20;
  const c = 30;
}

console.log(a); // 10
console.log(b); // ReferenceError: b is not defined
console.log(c); // ReferenceError: c is not defined
```

`var` allows variables to be hoisted, meaning they can be referenced in code before they are declared. `let` and `const` will not allow this, instead throwing an error.

```
console.log(a); // undefined
var a = 'foo';

console.log(b); // ReferenceError: can't access lexical declaration 'b' before
initialization
let b = 'baz';

console.log(c); // ReferenceError: can't access lexical declaration 'c' before
initialization
const c = 'bar';
```

Redeclaring a variable with `var` will not throw an error, but 'let' and 'const' will.

```
var a = 'foo';
var a = 'bar';
console.log(a); // "bar"

let b = 'baz';
let b = 'qux'; // Uncaught SyntaxError: Identifier 'b' has already been declared
```

`let` and `const` differ in that `let` allows reassigning the variable's value while `const` does not.

```
// This is ok.
let a = 'foo';
a = 'bar';
console.log(a); // bar

// This causes an exception.
const b = 'baz';
b = 'qux';
console.log(b) // TypeError: Assignment to constant variable.
```

🌟 **Try this example on CodeSandbox**

Q 2.4. What is Hoisting in JavaScript?

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 allows functions to be safely used in code before they are declared.

Example 01: Function Hoisting

One of the advantages of hoisting is that it lets you use a function before you declare it in your code.

```
getName("Sadhika Sandal");

function getName(name) {
  console.log("Hello " + name);
}
```

Output:

```
Hello Sadhika Sandal
```

Example 02: Variable Hoisting

```
console.log(message); // output: undefined
var message = "The variable Has been hoisted";
```

The above code looks like as below to the interpreter,

```
var message;
console.log(message);
message = "The variable Has been hoisted";
```

Example 03: `let` and `const` hoisting
All declarations (function, var, let, const and class) are hoisted in JavaScript, while the `var` declarations are initialized with `undefined`, but `let` and `const` declarations remain uninitialized.

```
console.log(x);
let x = 10;

// Output: ReferenceError: x is not defined
```

They will only get initialized when their lexical binding (assignment) is evaluated during runtime by the JavaScript engine. This means we can't access the variable before the engine evaluates its value at the place it was declared in the source code. This is what we call **Temporal Dead Zone**, A time span between variable creation and its initialization where they can't be accessed.

Note: JavaScript only hoists declarations, not initialisation

🌟 **Try this example on CodeSandbox**

Q 2.5. In which case the function definition is not hoisted in JavaScript?

Let us take the following **function expression**

```
var foo = function foo() {
  return 12;
}
```

In JavaScript `var`-declared variables and functions are `hoisted`. Let us take function `hoisting` first. Basically, the JavaScript interpreter looks ahead to find all the variable declaration and hoists them to the top of the function where It is declared. For example:

```
foo(); // Here foo is still undefined
var foo = function foo() {
  return 12;
};
```

The code above behind the scene look something like this:

```
var foo = undefined;
foo(); // Here foo is undefined
foo = function foo() {
  // Some code stuff
}
var foo = undefined;
foo = function foo() {
  // Some code stuff
}
foo(); // Now foo is defined here
```

Q 2.6. What is the Temporal Dead Zone in ES6?

In ES6, **let** bindings are not subject to "variable hoisting", which means that **let** declarations do not move to the top of the current execution context.

Referencing the variable in the block before the initialization results in a `ReferenceError` (contrary to a variable declared with var, which will just have the **undefined** value). The variable is in a "temporal dead zone" from the start of the block until the initialization is processed.

```
console.log(aVar); // undefined
console.log(aLet); // causes ReferenceError: aLet is not defined

var aVar = 1;
let aLet = 2;
```

Q 2.7. What is the purpose of double exclamation?

The double exclamation or negation(!!) ensures the resulting type is a boolean. If it was falsey (e.g. `0`, `null`, `undefined`, etc.), it will be `false`, otherwise, `true`.
For example, you can test IE version using this expression as below,

```
let isIE11 = false;
isIE11 = !!navigator.userAgent.match(/Trident.*rv[ :]*11\./);
console.log(isIE11); // returns true or false
```

If you do not use this expression then it returns the original value.

```
console.log(navigator.userAgent.match(/Trident.*rv[ :]*11\./));  // returns either
an Array or null
```

Note: The expression !! is not an operator, but it is just twice of ! operator.

Q 2.8. In JavaScript, what is the difference between `var x = 1` and `x = 1`?

`var x = 1`:

- Allowed in 'strict mode'.
- The var statement declares a function-scoped or globally-scoped variable, optionally initializing it to a value.
- Variables declared using var inside a { } block can be accessed from outside the block.
- Variables defined using var inside a function are not accessible (visible) from outside the function.
- Duplicate variable declarations using var will not trigger an error, even in strict mode, and the variable will not lose its value unless another assignment is performed.

```
var x = 1;

if (x === 1) {
  var x = 2;

  console.log(x); // expected output: 2
}

console.log(x); // expected output: 2
var x = 5; // global
function someThing(y) {
  var x = 3; // local
  var z = x + y;
  console.log(z);
}
someThing(4); // 7
console.log(x); // 5
```

`x = 1`:

- Not allowed in 'strict mode'.
- Undeclared Variables like: x = 1 is accessible in: (Block scope - Function scope - Global scope)
- Outside of strict mode, however, if you assign a value to a name that has not been declared with let, const, or var, you'll end up creating a new global variable. It will be global no matter how deeply nested within functions and

blocks your code is, which is almost certainly not what you want, is bug-prone, and is one of the best reasons for using strict mode!

- Global variables created in this accidental way are like global variables declared with var: they define properties of the global object.
- Unlike the properties defined by proper var declarations, these properties can be deleted with the delete operator.
- Not recommended.

```
var x = 5; // global
function someThing(y) {
  x = 1; // still global!
  var z = x + y;
  console.log(z);
}
someThing(4) // 5
console.log(x) // 1
```

Example:

```
{
  console.log(x + y); // NaN
  var x = 1;
  var y = 2;
}
{
  console.log(x + y); // Uncaught ReferenceError: x is not defined
   x = 1;
   y = 2;
}
```

	var x = 1	**x = 1**
Strict mode	✓	❌
Block scope	❌	✓
Function scope	✓	✓
Global scope	✓	✓
Hoisting	✓	❌

	var x = 1	x = 1
Reassigning	✓	✓

Q 2.9. How do you assign default values to variables?

You can use the logical or operator `||` in an assignment expression to provide a default value.
Syntax:

```
var a = b || c;
```

As per the above expression, variable 'a 'will get the value of 'c' only if 'b' is falsy (if is null, false, undefined, 0, empty string, or NaN), otherwise 'a' will get the value of 'b'.

Q 2.10. What is the precedence order between local and global variables?

A local variable takes precedence over a global variable with the same name.

```
var msg = "Good morning";

function greeting() {
  msg = "Good Evening";
  console.log(msg);
}
greeting();
```

Output:

```
Good Evening
```

🌟 **Try this example on CodeSandbox**

Q 2.11. What is variable shadowing in javascript?

Variable shadowing occurs when a variable declared within a certain scope (decision block, method, or inner class) has the same name as a variable declared in an outer scope. This outer variable is said to be shadowed.

If there is a variable in the global scope, and you'd like to create a variable with the same name in a function. The variable in the inner scope will temporarily shadow the variable in the outer scope.

Example:

```
/**
 * Variable Shadowing
 */
var val = 10;

function Hoist(val) {
  console.log(val);
}
Hoist(20); // 20
```

🌟 **Try this example on CodeSandbox**

Q 2.12. Explain `var self = this` in JavaScript?

The **self** is being used to maintain a reference to the original this even as the context is changing. It is a technique often used in event handlers (especially in closures).

`this` is a JavaScript keyword which refers to the current context. Unlike other programming languages, JavaScript does not have block scoping (in C open/close {} curly braces refers to a block). JavaScript has two scopes namely, global and local scope.

Example:

```
/**
 * this Context
 */
const context = {
  prop: 10,
  getCurrentContext: function () {
    return this.prop;
  }
};

console.log(context.getCurrentContext()); // 10
```

Note: 'self' should not be used this way anymore, since modern browsers provide a global variable self pointing to the global object of either a normal window or a WebWorker.

🌟 Try this example on CodeSandbox

Q 2.13. How do you swap variables using destructuring assignment?

```
var x = 10, y = 20;

[x, y] = [y, x];

console.log(x); // 20
console.log(y); // 10
```

🌟 Try this example on CodeSandbox

Q 2.14. What is scope chain in javascript?

The scope chain in JavaScript refers to the chain of nested scopes that a JavaScript program uses to look up variable and function references. When a variable or function is referenced in JavaScript code, the interpreter first looks for it in the current scope. If it's not found there, it moves up the scope chain to the next outer scope and looks for it there. It continues doing this until the variable or function is found or until it reaches the global scope.

Example::

```
let globalVar = "I'm a global variable";

function outer() {
  let outerVar = "I'm an outer variable";

  function inner() {
    let innerVar = "I'm an inner variable";
    console.log(innerVar); // "I'm an inner variable"
    console.log(outerVar); // "I'm an outer variable"
    console.log(globalVar); // "I'm a global variable"
  }

  inner();
}

outer();
```

3. DATA TYPES

Q 3.1. What are data types in javascript?

There are eight basic data types in JavaScript.

Data Types	Description	Example
String	Represents textual data	let str = 'Hi', let str2 = "Hello", let str3 = `Hello World`
Number	An integer or a floating-point number	let num = 3, let num2 = 3.234, let num3 = 3e-2
BigInt	An integer with arbitrary precision	let num = 900719925124740999n, let num = 1n
Boolean	Any of two values: true or false	let flag = true
undefined	A data type whose variable is not initialized	let a;
null	Denotes a null value	let a = null;
Symbol	Data type whose instances are unique and immutable	let value = Symbol('hello');
Object	key-value pairs of collection of data	let student = { };

String:

`String` is used to store text. In JavaScript, strings are surrounded by quotes:

- Single quotes: 'Hello'
- Double quotes: "Hello"
- Backticks: `Hello`

Example::

```
// Strings
const firstName = "Mukul";
const lastName = "Mittal";
const result = `Name: ${firstName} ${lastName}`;

console.log(result); // Name: Mukul Mittal
```

Number:

Number represents integer and floating numbers (decimals and exponentials). A number type can also be `+Infinity`, `-Infinity`, and `NaN` (not a number).

```
const number1 = 3;
const number2 = 3.433;
const number3 = 3e5; // 3 * 10^5

const number4 = 3 / 0;
console.log(number4); // Infinity

const number5 = -3 / 0;
console.log(number5); // -Infinity

// strings can't be divided by numbers
const number6 = "abc" / 3;
console.log(number6); // NaN
```

BigInt:

In JavaScript, Number type can only represent numbers less than (2^{53} - 1) and more than -(2^{53} - 1). However, if you need to use a larger number than that, you can use the BigInt data type.

A BigInt number is created by appending `n` to the end of an integer.

```
// BigInt value
const num1 = 100000000000000000n;
const num2 = 1000000000000000000n;
const num3 = 10;

// Adding two big integers
const result1 = num1 + num2;
console.log(result1); // "1100000000000000000n"

// Error! BitInt and number cannot be added
const result2 = num1 + num2 + num3;
```

```
console.log(result2); // Uncaught TypeError: Cannot mix BigInt and other types
```

Boolean:

This data type represents logical entities. Boolean represents one of two values: `true` or `false`.

```
const dataChecked = true;
const valueCounted = false;
```

undefined:

The undefined data type represents value that is not assigned. If a variable is declared but the value is not assigned, then the value of that variable will be undefined.

```
let name;
console.log(name); // undefined

let name = undefined;
console.log(name); // undefined
```

null:

In JavaScript, `null` is a special value that represents empty or unknown value.

```
const number = null;
```

Symbol:

A value having the data type Symbol can be referred to as a symbol value. Symbol is an immutable primitive value that is unique.

```
// Two symbols with the same description

const value1 = Symbol('hello');
const value2 = Symbol('hello');

let result = (value1 === value2) ? true : false;  // false;

// Note: Though value1 and value2 both contain 'hello', they are different as they
are of the Symbol type.
```

Object:

An object is a complex data type that allows us to store collections of data.

```
const employee = {
    firstName: 'John',
    lastName: 'K',
```

```
    email: 'john.k@gmail.com'
};
```

Q 3.2. What is `undefined` property?

The undefined property indicates that a variable has not been assigned a value, or not declared at all. The type of undefined value is undefined too.

```
var user;    // Value is undefined, type is undefined
console.log(typeof(user)) //undefined
```

Any variable can be emptied by setting the value to undefined.

```
user = undefined
```

Q 3.3. What is difference between `null` vs `undefined`?

Null:

`Null` means an empty or non-existent value. Null is assigned, and explicitly means nothing.

```
var test = null;
console.log(test); // null
```

`null` is also an object. Interestingly, this was actually an error in the original JavaScript implementation:

```
console.log(typeof test); // object
```

Undefined:

Undefined means a variable has been declared, but the value of that variable has not yet been defined. For example:

```
var test2;
console.log(test2); // undefined
```

Unlike null, undefined is of the type undefined:

```
console.log(typeof test2); // undefined
```

Difference:

Null	Undefined
It is an assignment value which indicates that variable points to no object.	It is not an assignment value where a variable has been declared but has not yet been assigned a value.
Type of null is object	Type of undefined is undefined
The null value is a primitive value that represents the null, empty, or non-existent reference.	The undefined value is a primitive value used when a variable has not been assigned a value.
Indicates the absence of a value for a variable	Indicates absence of variable itself
Converted to zero (0) while performing primitive operations	Converted to NaN while performing primitive operations

Q 3.4. What is Coercion in JavaScript?

Type coercion is the automatic or implicit conversion of values from one data type to another (such as strings to numbers). Type conversion is similar to type coercion because they both convert values from one data type to another with one key difference — type coercion is implicit whereas type conversion can be either implicit or explicit.

```
const value1 = '10';
const value2 = 20;

let sum = value1 + value2;

console.log(sum);
```

In the above example, JavaScript has coerced the 10 from a number into a string and then concatenated the two values together, resulting in a string of 1020. JavaScript had a choice between a string or a number and decided to use a string.

```
// Example of explicit coercion
const value1 = '10';
const value2 = 20;
```

```
let sum = Number(value1) + value2;

console.log(sum);
```

4. OPERATORS

Q 4.1. What are various operators supported by javascript?

An operator is capable of manipulating(mathematical and logical computations) a certain value or operand. There are various operators supported by JavaScript as below,

Arithmetic Operators:

Arithmetic operators are used to perform mathematical operations between numeric operands.

Operators	Description	Example (say `let x = 10, y = 20;`)
+	Adds two numeric operands.	x + y
-	Subtract right operand from left operand	y - x
*	Multiply two numeric operands.	x * y
/	Divide left operand by right operand.	y / x
%	Modulus operator. Returns remainder of two operands.	x % 2
++	Increment operator. Increase operand value by one.	x++
--	Decrement operator. Decrease value by one.	x--

Comparison Operators:

JavaScript provides comparison operators that compare two operands and return a boolean value `true` or `false`.

Operators	Description
==	Compares the equality of two operands without considering type.
===	Compares equality of two operands with type.
!=	Compares inequality of two operands.
>	Returns a boolean value true if the left-side value is greater than the right-side value; otherwise, returns false.
<	Returns a boolean value true if the left-side value is less than the right-side value; otherwise, returns false.
>=	Returns a boolean value true if the left-side value is greater than or equal to the right-side value; otherwise, returns false.
<=	Returns a boolean value true if the left-side value is less than or equal to the right-side value; otherwise, returns false.

Logical Operators:

The logical operators are used to combine two or more conditions.

1. && - is known as AND operator. It checks whether two operands are non-zero or not (0, false, undefined, null or "" are considered as zero). It returns 1 if they are non-zero; otherwise, returns 0

2. || - is known as OR operator. It checks whether any one of the two operands is non-zero or not (0, false, undefined, null or "" is considered as zero). It returns 1 if any one of of them is non-zero; otherwise, returns 0.

3. ! - is known as NOT operator. It reverses the boolean result of the operand (or condition). !false returns true, and !true returns false.

Assignment Operators:

The assignment operators to assign values to variables with less key strokes.

Operators	Description
=	Assigns right operand value to the left operand.
+=	Sums up left and right operand values and assigns the result to the left operand.
-=	Subtract right operand value from the left operand value and assigns the result to the left operand.
*=	Multiply left and right operand values and assigns the result to the left operand.
/=	Divide left operand value by right operand value and assign the result to the left operand.
%=	Get the modulus of left operand divide by right operand and assign resulted modulus to the left operand.

Q 4.2. What are the bitwise operators available in javascript?

Below are the list of bit-wise logical operators used in JavaScript

Operator	Usage	Description
Bitwise AND	a & b	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	a	b	Returns a one in each bit position for which the corresponding bits of either or both operands are ones.
Bitwise XOR	a ^ b	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.

Operator	Usage	Description
Bitwise NOT	~ a	Inverts the bits of its operand.
Left shift	a << b	Shifts a in binary representation b (< 32) bits to the left, shifting in zeroes from the right.
Sign-propagating right shift	a >> b	Shifts a in binary representation b (< 32) bits to the right, discarding bits shifted off.
Zero-fill right shift	a >>> b	Shifts a in binary representation b (< 32) bits to the right, discarding bits shifted off, and shifting in zeroes from the left.

Examples:

Operation	Result	Same as	Result
5 & 1	1	0101 & 0001	0001
`5	1`	5	0101
`~ 5`	-6	~0101	1010
`5 << 1`	10	0101 << 1	1010
`5 ^ 1`	4	0101 ^ 0001	0100
`5 >> 1`	2	0101 >> 1	0010
5 >>> 1	2	0101 >>> 1	0010

🌟 **Try this example on CodeSandbox**

Q 4.3. What is the difference between == and === operators?

JavaScript provides both strict(===, !==) and type-converting(==, !=) equality comparison. The strict operators takes type of variable in consideration, while non-strict operators make type correction/conversion based upon values of variables. The strict operators follow the below conditions for different types,

1. Two strings are strictly equal when they have the same sequence of characters, same length, and same characters in corresponding positions.
2. Two numbers are strictly equal when they are numerically equal. i.e, Having the same number value. There are two special cases in this,
    i. NaN is not equal to anything, including NaN.
    ii. Positive and negative zeros are equal to one another.
3. Two Boolean operands are strictly equal if both are true or both are false.
4. Two objects are strictly equal if they refer to the same Object.
5. Null and Undefined types are not equal with ===, but equal with ==. i.e, null===undefined --> false but null==undefined --> true

Example:

```
0 == false // true
0 === false // false
1 == "1" // true
1 === "1" // false
null == undefined // true
null === undefined // false
"0" == false // true
"0" === false // false
[] === [] // false, refer different objects in memory
{} === {} // false, refer different objects in memory
```

🌟 **Try this example on CodeSandbox**

Q 4.4. What is typeof operator?

In JavaScript, the typeof operator returns the data type of its operand in the form of a string. The operand can be any object, function, or variable.

Example:

```
console.log(typeof undeclaredVariable); // "undefined"

let a;
console.log(typeof a); // "undefined"
```

```
const b = "Hello World";
console.log(typeof b); // "string"

const c = 42;
console.log(typeof c); // "number"

const d = 3.1415;
console.log(typeof d); // "number"

const e = true;
console.log(typeof e); // "boolean"

const f = null;
console.log(typeof f); // "object"

const g = undefined;
console.log(typeof g); // "undefined"

const h = { b: "c" };
console.log(typeof h); // "object"

const i = function () {
  return 10;
};

console.log(typeof i); // "function"
```

🌟 **Try this example on CodeSandbox**

Q 4.5. What is an Unary operator?

The unary(+) operator is used to convert a variable to a number. If the variable cannot be converted, it will still become a number but with the value NaN.

Example:

```
var x = "100";
var y = +x;
console.log(typeof x, typeof y); // string, number

var a = "Hello";
var b = +a;
console.log(typeof a, typeof b, b); // string, number, NaN
```

🌟 **Try this example on CodeSandbox**

Q 4.6. What is the purpose of delete operator?

The delete keyword is used to delete the property as well as its value.

```
var user = {name: "Sadhika Chaudhuri", age: 24};
delete user.age;

console.log(user); // {name: "Sadhika Chaudhuri"}
```

Q 4.7. What is a conditional operator in javascript?

The conditional (ternary) operator is the only JavaScript operator that takes three operands which acts as a shortcut for if statement.

Syntax:

```
<condition> ? <value1> : <value2>;
```

Example:

```
const isAuthenticated = false;

console.log(isAuthenticated ? 'Hello, welcome' : 'Sorry, you are not
authenticated');
```

Q 4.8. Can you apply chaining on conditional operator?

Yes, you can apply chaining on conditional operator similar to if ... else if ... else if ... else chain.

Example:

```
function getValue(someParam) {
  return condition1 ? value1
    : condition2 ? value2
    : condition3 ? value3
    : value4;
}

// The above conditional operator is equivalent to:
function getValue(someParam) {
  if (condition1) {
    return value1;
```

```
  } else if (condition2) {
    return value2;
  } else if (condition3) {
    return value3;
  } else {
    return value4;
  }
}
```

Q 4.9. What is the difference between `typeof` and `instanceof` operator?

The `typeof` operator checks if a value has type of primitive type which can be one of boolean, function, object, number, string, undefined and symbol (ES6).
Example:

```
const x = "Hello World";
const y = new String("Hello World");

typeof x; // returns 'string'
typeof y; // returns 'object'
```

The `instanceof` is a binary operator, accepting an object and a constructor. It returns a boolean indicating whether or not the object has the given constructor in its prototype chain.

```
const a = "Hello World";
const b = new String("Hello World");

a instanceof String; // returns false
b instanceof String; // returns true
```

☆ **Try this example on CodeSandbox**

Q 4.10. What is the output of below spread operator array?

```
[...'Hello']
```

Output: ['H', 'e', 'l', 'l', 'o']

Explanation: The string is an iterable type and the spread operator with in an array maps every character of an iterable to one element. Hence, each character of a string becomes an element within an Array.

5. NUMBERS

Q 5.1. How do you generate random integers?

The `Math.random()` function returns a floating-point, pseudo-random number in the range 0 to less than 1 (inclusive of 0, but not 1). For example, if you want generate random integers between 1 to 100, the multiplication factor should be 100,

```
// Example 01:
Math.random(); // returns a random integer between 0 to 1

// Example 02:
Math.floor(Math.random() * 100) + 1; // returns a random integer from 1 to 100

// Example 03:
function getRandomNumber(max) {
  return Math.floor(Math.random() * max) + 1;
}

console.log(getRandomNumber(10)); // returns a random integer from 1 to 10
```

🌟 **Try this example on CodeSandbox**

Q 5.2. What is isNaN?

The `isNaN()` function determines whether a value is NaN (Not a Number) or not. This function returns `true` if the value equates to NaN. The `isNaN()` method converts the value to a number before testing it.

```
isNaN('Hello') // true

isNaN('100') // false

typeof NaN // Number

Number.isNaN('Hello'); // false
```

🌟 **Try this example on CodeSandbox**

Q 5.3. What is the purpose of isFinite function?

The global `isFinite()` function determines whether the passed value is a finite number. It returns `false` if the value is `+infinity`, `-infinity`, or `NaN` (Not-a-Number), otherwise it returns true.

```
isFinite(Infinity);  // false
isFinite(NaN);       // false
isFinite(-Infinity); // false

isFinite(100);  // true
isFinite(1/0); // false

Number.isFinite(0 / 0); // false
Number.isFinite(null); // false
Number.isFinite("123") // false
```

🌟 **Try this example on CodeSandbox**

Q 5.4. Explain NEGATIVE_INFINITY in JavaScript?

The `Number.NEGATIVE_INFINITY` property represents the negative Infinity value.
Syntax:

```
Number.NEGATIVE_INFINITY
```

- Negative infinity is a number in javascript, which is derived by 'dividing negative number by zero'.
- A number object needs not to be created to access this static property.
- The value of negative infinity is the same as the negative value of the infinity property of the global object.

```
/**
 * NEGATIVE_INFINITY
 */

console.log(-10/0); // -Infinity
console.log(Number.NEGATIVE_INFINITY); // -Infinity
console.log(Number.MAX_VALUE + Number.MAX_VALUE); // Infinity
console.log(-2 * Number.MAX_VALUE); // -Infinity

console.log("Math.pow(10, 1000): " + Math.pow(10, 1000)); // Infinity
console.log("Math.log(0): " + Math.log(0)); // -Infinity
```

```
console.log(Number.NEGATIVE_INFINITY === -2 * Number.MAX_VALUE); // true
```

🌟 [Try this example on CodeSandbox]

6. STRING

Q 6.1. What is the difference between slice and splice?

1. slice():

The `slice()` method returns a new array with a copied slice from the original array. The first optional argument is the beginning index and the second optional argument is the ending index (non-inclusive).
Example:

```
let languages = [ "JavaScript", "Python", "Java", "PHP" ];

languages.slice(1,3); // ["Python", "Java"]
languages.slice(2); // (from index 2 until the end of the array).
// ["Java", "PHP"]

console.log(languages); // the original array is not mutated.
// [ "JavaScript", "Python", "Java", "PHP" ]
```

2. splice():

The `splice()` method changes the content of the array in place and can be used to add or remove items from the array. When only one argument is provided, all the items after the provided starting index are removed from the array.
Example:

```
let numbers = [10, 20, 30];

numbers.splice(2, 1, 40, 50); // returns removed array:[30]

console.log(numbers); // Original array is mutated.
// returns: [10, 20, 40, 50]
```

Difference:

Slice	Splice
Doesn't modify the original array(immutable)	Modifies the original array(mutable)
Returns the subset of original array	Returns the deleted elements as array
Used to pick the elements from array	Used to insert or delete elements to/from array

🌟 **Try this example on CodeSandbox**

Q 6.2. How do you check whether a string contains a substring?

There are 3 fastest ways to check whether a string contains a substring or not,

1. Using RegEx:

The regular expression `test()` method checks if a match exists in a string. This method returns `true` if it finds a match, otherwise, it returns `false`.

```
let str = "JavaScript, Node.js, Express.js, React.js, MongoDB";
let exp1 = /MongoDB/g;
let exp2 = /Ajax/;

exp1.test(str); // true
exp2.test(str); // false
```

2. Using indexOf:

The `indexOf()` method is case-sensitive and accepts two parameters. The first parameter is the substring to search for, and the second optional parameter is the index to start the search from (default index is 0).

```
let str = "JavaScript, Node.js, Express.js, React.js, MongoDB";

str.indexOf('MongoDB') !== -1 // true
str.indexOf('PHP') !== -1 // false
str.indexOf('Node', 5) !== -1 // true
```

3. Using includes:

The `includes()` is also case-sensitive and accepts an optional second parameter, an integer which indicates the position where to start searching for.

```
let str = "JavaScript, Node.js, Express.js, React.js, MongoDB";

str.includes('MongoDB') // true
str.includes('PHP') // false
str.includes('Node', 5) //true
```

🌟 **Try this example on CodeSandbox**

Q 6.3. How do you trim a string in javascript?

The `trim()` method removes whitespace from both sides of a string. JavaScript provides 3 simple functions on how to trim strings.

1. string.trim():

The `string.trim()` removes sequences of whitespaces and line terminators from both the start and the end of the string.

```
const name = "  Karan Talwar  ";
console.log(name.trim()); // => 'Karan Talwar'

const phoneNumber = "\t  80-555-123\n ";
console.log(phoneNumber.trim()); // => '80-555-123'
```

2. string.trimStart():

The `string.trimStart()` removes sequences of whitespaces and line terminators only from the start of the string.

```
const name = "   Karan Talwar  ";
console.log(name.trimStart()); // => "Karan Talwar  "

const phoneNumber = "\t  80-555-123\n ";
console.log(phoneNumber.trimStart()); // => "80-555-123 "
```

3. string.trimEnd():

The `string.trimEnd()` removes sequences of whitespaces and line terminators only from the end of the string.

```
const name = "  Karan Talwar ";
console.log(name.trimEnd()); // => " Karan Talwar"

const phoneNumber = "\t  80-555-123\n ";
console.log(phoneNumber.trimEnd()); // => " 80-555-123"
```

🌟 **Try this example on CodeSandbox**

Q 6.4. What is eval function in javascript?

The `eval()` function evaluates JavaScript code represented as a string. The string can be a JavaScript expression, variable, statement, or sequence of statements.

```
console.log(eval('10 + 20')); // 30

let x = 10;
let y = 20;
let z = '50';
eval('x + y + 1'); // returns 31
eval(z);           // returns 50
```

If the argument of `eval()` is not a string, `eval()` returns the argument unchanged. In the following example, the String constructor is specified and eval() returns a String object rather than evaluating the string.

```
eval(new String('10 + 20')); // returns a String object containing "10 + 20"
eval('10 + 20');             // returns 30


// work around
let expression = new String('10 + 20');
eval(expression.toString()); // returns 30
```

Warning: *Executing JavaScript from a string is an enormous security risk. It is far too easy for a bad actor to run arbitrary code when you use eval().*

🌟 **Try this example on CodeSandbox**

Q 6.5. How do you check if a string starts with another string?

You can use ECMAScript 6 `String.prototype.startsWith()` method to check a string starts with another string or not. But it is not yet supported in all browsers. Let us see an example to see this usage,

```
let str = "Hello World";

console.log(str.startsWith("Hello")); // true
console.log(str.startsWith("World")); // false
```

⭐ [Try this example on CodeSandbox]

7. ARRAY

Q 7.1. Explain arrays in JavaScript?

JavaScript array is an object that represents a collection of similar type of elements. It can holds values (of any type) not particularly in named properties/keys, but rather in numerically indexed positions.

Syntax:

```
const array_name = [item-1, item-2, item-3, ...];
```

Example 01: Creating an array

```
// array of numbers
const numbers = [10, 20, 30, 40, 50];

// using new keyword
const numbers = new Array(10, 20, 30, 40, 50);

// array of strings
let fruits = ["Apple", "Orange", "Plum", "Mango"];
```

Example 02: Accessing array elements

```
let fruits = ["Apple", "Orange", "Plum", "Mango"];

fruits[0]; // Apple
fruits[fruits.length - 1] // Mango

// Iterate array elements
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}
```

Example 03: Adding new array elements

```
let fruits = ["Apple", "Orange", "Plum", "Mango"];
```

```
fruits.push("Grapes");  // Adds a new element (Grapes) to fruits
```

⭐ **Try this example on CodeSandbox**

Q 7.2. What are associative arrays in javascript?

Associative arrays are basically objects in JavaScript where indexes are replaced by user-defined keys. They do not have a length property like a normal array and cannot be traversed using a normal for loop.

Syntax:

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

Example:

```
const employee = {
  id: 12345,
  name: "Sakshi Memon",
  email: "sakshi.memon@email.com"
};

// Accesing employee elements
console.log(employee.id); // 12345
console.log(employee.name); // Sakshi Memon

// Array Length
console.log(Object.keys(employee).length); // 3

// Retrieve the elements
for (let key in employee) {
  console.log(key + " = " + employee[key]);
}

// Output
id = 12345
name = Sakshi Memon
email = sakshi.memon@email.com
```

⭐ **Try this example on CodeSandbox**

Q 7.3. Calculate the length of the associative array?

Method 1: Using `Object.keys().length`

```
const employee = {
  id: 12345,
  name: "Sakshi Memon",
  email: "sakshi.memon@email.com"
};

console.log(Object.keys(employee).length); // Output 3
```

Method 2: Using `Object.hasOwnProperty()`

```
function getLength(object) {
  let count = 0;
  for (let key in object) {
    // hasOwnProperty method check own property of object
    if (object.hasOwnProperty(key)) count++;
  }
  return count;
}

console.log(getLength(employee)); // Output 3
```

Method 3: Using `Object.getOwnPropertyNames()`

```
const employee = {
  id: 12345,
  name: "Sakshi Memon",
  email: "sakshi.memon@email.com"
};

Object.getOwnPropertyNames(employee).length; // Output 3
```

🌟 **Try this example on CodeSandbox**

Q 7.4. What is the difference between Array and Array of Objects in JavaScript?

Objects represent a special data type that is mutable and can be used to store a collection of data (rather than just a single value). Arrays are a special type of variable that is also mutable and can also be used to store a list of values.

Example: Arrays

```
const numbers = [10, 20, 30];

// Iterating through loop
for (let i = 0; i < numbers.length; i++) {
  console.log(numbers[i]);
}
```

```
// Pop an element from array
numbers.pop();
console.log("after pop(): " + numbers);
```

Example: Array of Objects

```
const employees = [
  { id: 101, name: "Sakshi Memon", email: "sakshi.memon@email.com" },
  { id: 102, name: "Subhash Shukla", email: "subhash.shukla@email.com" },
  { id: 103, name: "Mohini Karpe", email: "mohini.karpe@email.com" }
];

// Using DOT notation
console.log(employees[0].name);

// Using delete keyword
delete employees[0];

// Iterating using for..in loop
for (let key in employees) {
  console.log(employees[key]);
}
```

Difference:

S.No.	Array	Array of objects
1.	Arrays are best to use when the elements are numbers.	Objects are best to use when the elements strings
3.	The elements can be manipulated using [].	The properties can be manipulated using both . (DOT) notation and [].
4.	The elements can be popped out of an array using the pop() function.	The keys or properties can be deleted by using the delete keyword.
5.	Iterating through an array is possible using For loop, For..in, For..of, and ForEach().	Iterating through an array of objects is possible using For..in, For..of, and ForEach().

☆ **Try this example on CodeSandbox**

Q 7.5. Explain array methods [join(), pop(), push(), shift(), unshift(), concat(), map(), filter(), reduce(), reduceRight(), every(), some(), indexOf(), lastIndexOf(), find(), findIndex(), includes()]

1. array.join():

The `join()` method creates and returns a new string by concatenating all of the elements in an array (or an array-like object), separated by commas or a specified separator string. If the array has only one item, then that item will be returned without using the separator.

```
var elements = ['Fire', 'Air', 'Water'];

console.log(elements.join()); // Output: "Fire,Air,Water"
console.log(elements.join('')); // Output: "FireAirWater"
console.log(elements.join('-')); // Output: "Fire-Air-Water"
```

2. array.pop():

The pop() method removes the last element from an array and returns that element. This method changes the length of the array.

```
var plants = ['broccoli', 'cauliflower', 'kale'];

console.log(plants.pop()); // Output: "kale"
console.log(plants); // Output: Array ["broccoli", "cauliflower"]
console.log(plants.pop()); // Output: "cauliflower"
console.log(plants.pop()); // Output: "broccoli"
console.log(plants.pop()); // Output: "undefined"
```

3. array.push():

The push() method adds one or more elements to the end of an array and returns the new length of the array.

```
const animals = ['pigs', 'goats', 'sheep'];

const count = animals.push('cows');
console.log(count); // Output: 4
console.log(animals); // Output: Array ["pigs", "goats", "sheep", "cows"]
```

4. array.shift():

The shift() method removes the first element from an array and returns that removed element. This method changes the length of the array.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.shift();
console.log(fruits) // Output: Array ["Orange", "Apple", "Mango"]
```

5. array.unshift():

The unshift() method adds one or more elements to the beginning of an array and returns the new length of the array.

```
var fruits = ["Banana", "Orange", "Apple"];
fruits.unshift("Mango","Pineapple");
console.log(fruits); // Output: Array ["Mango", "Pineapple", "Banana", "Orange",
"Apple"]
```

6. array.concat():

The concat() method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.

```
const array1 = ['a', 'b', 'c'];
const array2 = ['d', 'e', 'f'];

console.log(array1.concat(array2)); // Output: Array ["a", "b", "c", "d", "e",
"f"]
```

7. array.map():

The map() method creates a new array with the results of calling a provided function on every element in the calling array.

```
var array1 = [1, 4, 9, 16];

// pass a function to map
const map1 = array1.map(x => x * 2);

console.log(map1); // Output: Array [2, 8, 18, 32]
```

8. array.filter():

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

```
var words = ['spray', 'limit', 'elite', 'exuberant', 'destruction'];

const result = words.filter(word => word.length > 6);

console.log(result); // Output: Array ["exuberant", "destruction"]
```

9. array.reduce():

The reduce() method executes a reducer function (that you provide) on each element of the array, resulting in a single output value.

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

console.log(array1.reduce(reducer)); // Output: 10
console.log(array1.reduce(reducer, 5)); // Output: 15
```

10. array.reduceRight():

The reduceRight() method applies a function against an accumulator and each value of the array (from right-to-left) to reduce it to a single value.

```
const array1 = [[0, 1], [2, 3], [4, 5]].reduceRight(
  (accumulator, currentValue) => accumulator.concat(currentValue)
);

console.log(array1); // Output: Array [4, 5, 2, 3, 0, 1]
```

11. array.every():

The every() method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.

```
function isBelowThreshold(currentValue) {
  return currentValue < 40;
}

var array1 = [1, 30, 39, 29, 10, 13];
console.log(array1.every(isBelowThreshold)); // Output: true
```

12. array.some():

The some() method tests whether at least one element in the array passes the test implemented by the provided function. It returns a Boolean value.

```
var array = [1, 2, 3, 4, 5];

var even = function(element) {
  // checks whether an element is even
  return element % 2 === 0;
};

console.log(array.some(even)); // Output: true
```

13. array.indexOf():

The indexOf() method returns the first index at which a given element can be found in the array, or -1 if it is not present.

```
var beasts = ['ant', 'bison', 'camel'];

console.log(beasts.indexOf('camel')); // Output: 2
console.log(beasts.indexOf('giraffe')); // Output: -1
```

14. array.lastIndexOf():

The lastIndexOf() method returns the index within the calling String object of the last occurrence of the specified value, searching backwards from fromIndex. Returns -1 if the value is not found.

```
var paragraph = 'The quick brown fox jumps over the lazy dog. If the dog barked,
was it really lazy?';

var searchTerm = 'dog';

console.log('The index of the first "' + searchTerm + '" from the end is ' +
paragraph.lastIndexOf(searchTerm));
// Output: "The index of the first "dog" from the end is 52"
```

15. array.find():

The find() method returns the value of the first element in the provided array that satisfies the provided testing function.

```
var array1 = [5, 12, 8, 130, 44];

var found = array1.find(function(element) {
  return element > 100;
});

console.log(found); // Output: 130
```

16. array.findIndex():

The findIndex() method returns the index of the first element in the array that satisfies the provided testing function. Otherwise, it returns -1, indicating that no element passed the test.

```
var array1 = [5, 12, 8, 130, 44];

function isLargeNumber(element) {
  return element > 20;
}

console.log(array1.findIndex(isLargeNumber)); // Output: 3
```

17. array.includes():

The includes() method determines whether an array includes a certain value among its entries, returning true or false as appropriate.

```
var array1 = [1, 2, 3];
console.log(array1.includes(2)); // Output: true

var pets = ['cat', 'dog', 'bat'];
console.log(pets.includes('at')); // Output: false
```

Q 7.6. What are the benefits of using spread syntax and how is it different from rest syntax?

Spread operator or Spread Syntax allow us to expand the arrays and objects into elements in the case of an array and key-value pairs in the case of an object.

Example:

```
function sum(x, y, z) {
  return x + y + z;
}

const numbers = [1, 2, 3];
console.log(sum(...numbers));

// ES-5 way
console.log(sum.apply(null, numbers));
```

Example: Merge arrays

```
const newBrands = ["Tesla", "Mahindra"];
const brands = ["Ford", "Honda", ...newBrands, "BMW"];

console.log(brands);
```

Example: Copy array/object

```
let obj = { a: 10, b: 20, c: 30 };

// spread the object into a list of parameters
let objCopy = { ...obj };

// add new
obj.d = 40;
```

```
console.log(JSON.stringify(obj)); // { "a":10, "b":20, "c":30, "d":40 }
console.log(JSON.stringify(objCopy)); // { "a":10, "b":20, "c":30 }
```

Difference:

The main difference between `rest` and `spread` is that the rest operator puts the rest of some specific user-supplied values into a JavaScript array. But the spread syntax expands iterables into individual elements.

Spread Syntax	Rest Syntax
Spread operator as its name suggests it spreads or expands the content of the given element.	Rest Syntax is just the opposite of spread syntax it collects the data and stores that data in a variable which we can use further in our code.
It expands an Array in form of elements, while in key-value pairs in the case of Objects.	It collects the data in the developer's desired format.
You may or may not use the strict mode inside the function containing the spread operator.	You can not use the strict mode inside function containing the rest operator.
It will overwrite the identical properties inside two objects and replace the former with the latter.	It simply collects all properties and wraps them inside a container.

✯ **Try this example on CodeSandbox**

Q 7.7. What is the difference between for..in and for..of?

Both `for..of` and `for..in` statements iterate over lists; the values iterated on are different though, `for..in` returns a **list of keys** on the object being iterated, whereas `for..of` returns a **list of values** of the numeric properties of the object being iterated.

- **for in**: iterates over all enumerable properties of an object that are **keyed** by strings.
- **for of**: iterates over the **values** of an iterable objects. including: built-in `String`, `Array`, array-like objects (e.g., `arguments` or `NodeList`), `TypedArray`, `Map`, `Set`, and `user-defined` iterables.

Example:

```
// for..in
const list = [10, 20, 30];

for (let i in list) {
  console.log(i); // "0", "1", "2",
}

// for..of
for (let i of list) {
  console.log(i); // "10", "20", "30"
}
```

🌟 [Try this example on CodeSandbox]

Q 7.8. Can you give an example for destructuring an array?

Destructuring is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables. That is, we can extract data from arrays and objects and assign them to variables.

```
// Variable assignment.
const numbers = [10, 20, 30];
const [one, two, three] = numbers;

console.log(one); // 10
console.log(two); // 20
console.log(three); // 30
// Swapping variables
let a = 100;
let b = 200;

[a, b] = [b, a];

console.log(a); // 20
console.log(b); // 10
```

🌟 **Try this example on CodeSandbox**

Q 7.9. What are default values in destructuring assignment?

A variable can be assigned a default value when the value unpacked from the array or object is undefined during destructuring assignment. It helps to avoid setting default values separately for each assignment.

Array Destructuring:

```
const [x = 2, y = 4, z = 6] = [10];

console.log("x: " + x); // 10
console.log("y: " + y); // 4
console.log("z: " + z); // 6
```

Object Destructuring:

```
const { i = 2, j = 4, k = 6 } = { n: 10 };

console.log("i: " + i); // 2
console.log("j: " + j); // 4
console.log("k: " + k); // 6
```

🌟 **Try this example on CodeSandbox**

Q 7.10. When to use reduce(), map(), foreach() and filter() in JavaScript?

1. forEach():

It takes a callback function and run that callback function on each element of array one by one. Basically forEach works as a traditional for loop looping over the array and providing array elements to do operations on them.

```
let numbers = [10, 20, 30];

numbers.forEach(function (number, index) {
  console.log(number + " comes at " + index);
});

// Output
```

```
10 comes at 0
20 comes at 1
30 comes at 2
```

2. filter():

The main difference between forEach() and filter() is that forEach just loop over the array and executes the callback but filter executes the callback and check its return value. If the value is true element remains in the resulting array but if the return value is false the element will be removed for the resulting array.

Note: filter does not update the existing array it will return a new filtered array every time.

```
let numbers = [10, 20, 30];

let result = numbers.filter(function (number) {
  return number !== 20;
});

console.log(result);

// Output
[10, 30]
```

3. map():

map() like filter() & forEach() takes a callback and run it against every element on the array but whats makes it unique is it generate a new array based on your existing array.

Like filter(), map() also returns an array. The provided callback to map modifies the array elements and save them into the new array upon completion that array get returned as the mapped array.

```
let numbers = [10, 20, 30];

let mapped = numbers.map(function (number) {
  return number * 10;
});

console.log(mapped);

// Output
[100, 200, 300]
```

4. reduce():

reduce() method of the array object is used to reduce the array to one single value.

```
let numbers = [10, 20, 30];

let sum = numbers.reduce(function (sum, number) {
  return sum + number;
});

console.log(sum); // Output: 60
```

🌟 **Try this example on CodeSandbox**

Q 7.11. How do you define JSON arrays?

JSON is an acronym for JavaScript Object Notation, and is "an open standard data interchange format".

JSON array represents ordered list of values. JSON array can store multiple values. It can store string, number, boolean or object in JSON array.

```
// Empty JSON array
const empty = [ ];


// JSON Array of Numbers
const numbers = [12, 34, 56, 43, 95];


// JSON Array of Objects
{
 "employees": [
   { "name": "Kabir Dixit", "email": "kabir.dixit@gmail.com", "age": 23 },
   { "name": "Mukta Bhagat", "email": "mukta.bhagat@gmail.com", "age": 28 },
   { "name": "Sakshi Ramakrishnan", "email": "sakshi.ramakrishnan@gmail.com",
"age": 33 }
  ]
}

// access array values
console.log(employees[0].name) // Kabir Dixit
```

Q 7.12. How to validate JSON Object in javascript?

`JSON.parse()` function will use string and converts to JSON object and if it parses invalidate JSON data, it throws an exception (**Uncaught SyntaxError: Unexpected string in JSON**).

```
function isValidJson(json) {
  try {
    JSON.parse(json);
    return true;
  } catch (e) {
    return false;
  }
}

console.log(isValidJson("{}")); // true
console.log(isValidJson("abc")); // false
```

🌟 **Try this example on CodeSandbox**

Q 7.13. What is the purpose JSON stringify?

When sending data to a web server, the data has to be in a string format.
The `JSON.stringify()` method converts a JavaScript object or value to a JSON string format.

```
const user = {'name': 'Shashi Meda', 'email': 'shashi.meda@email.com', 'age': 28}

console.log(JSON.stringify(user)); // {"name":"Shashi
Meda","email":"shashi.meda@email.com","age":28}
```

Q 7.14. How do you parse JSON string?

When receiving the data from a web server, the data is always in a string format. But you can convert this string value to javascript object using `JSON.parse()` method.

```
const user = '{"name": "Shashi Meda", "email": "shashi.meda@email.com", "age":
28}'

console.log(JSON.parse(user));// {'name': 'Shashi Meda', 'email':
'shashi.meda@email.com', 'age': 28}
```

Q 7.15. What is the purpose of compare function while sorting arrays?

The purpose of the compare function is to define an alternative sort order. When the `sort()` function compares two values, it sends the values to the compare function, and sorts the values according to the returned (negative, zero, positive) value.
If omitted, the array elements are converted to strings, then sorted according to each character's Unicode code point value.

```
const numbers = [1, 2, 5, 3, 4];

numbers.sort((a, b) => b - a);
console.log(numbers); // [5, 4, 3, 2, 1]
```

🌟 **Try this example on CodeSandbox**

Q 7.16. Can you describe the main difference between a `.forEach` loop and a `.map()` loop and why you would pick one versus the other?

To understand the differences between the two, Let us look at what each function does.

1. Array.forEach():

- Iterates through the elements in an array.
- Executes a callback for each element.
- Does not return a value.

```
const numbers = [10, 20, 30];
const doubled = numbers.forEach((num, index) => {
  return num * 2;
});

console.log(doubled) // undefined
```

2. Array.map():

- Iterates through the elements in an array.
- "Maps" each element to a new element by calling the function on each element, creating a new array as a result.

```
const numbers = [10, 20, 30];
const doubled = numbers.map(num => {
  return num * 2;
});
```

```
console.log(doubled) // [20, 40, 60]
```

The main difference between `.forEach` and `.map()` is that `.map()` returns a new array. If you need the result, but do not wish to mutate the original array, `.map()` is the clear choice. If you simply need to iterate over an array, `forEach` is a fine choice.
🌟 **Try this example on CodeSandbox**

Q 7.17. What is unshift() method in JavaScript?

The `unshift()` method adds one or more elements to the beginning of an array and returns the new length of the array.
Example:

```
const numbers = [10, 20, 30];

console.log(numbers.unshift(40, 50)); // 5
console.log(numbers); // [40, 50, 10, 20, 30]
```

🌟 **Try this example on CodeSandbox**

Q 7.18. What is a rest parameter?

The rest parameter is used to represent an indefinite number of arguments as an array. The important point here is only the function's last parameter can be a "rest parameter".

This feature has been introduced to reduce the boilerplate code that was induced by the arguments.

Example:

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

console.log(sum(10)); // 10
console.log(sum(10, 20)); // 30
console.log(sum(10, 20, 30)); // 60
```

🌟 **Try this example on CodeSandbox**

Q 7.19. What happens if you do not use rest parameter as a last argument?

The rest parameter should be the last argument, as its job is to collect all the remaining arguments into an array.

Example: If you define a function like below it does not make any sense and will throw an `SyntaxError`.

```
function display(a, ...args, b) {
  console.log(a);
  for (let i = 0; i < args.length; i++) {
    console.log(args[i]);
  }
  console.log(b);
}

display(10, 20, 30, 40, 50);

// Output
SyntaxError: Rest element must be last element
```

🌟 **Try this example on CodeSandbox**

Q 7.20. What is difference between [] and new Array()?

`[]` and `new Array()` are two different ways of creating an array, but they are functionally equivalent.
The primary difference between them is in how they are created and in their behavior when used with certain methods.

`[]` is a shorthand for creating a new array. It is the preferred way to create an array in most cases, because it's more concise and easier to read. For example:

```
const myArray = []; // create a new empty array
```

On the other hand, `new Array()` is a constructor function that creates a new array object. It can be used to create an array of a specific length or with specific elements. For example:

```
const myArray = new Array(); // create a new empty array
const myOtherArray = new Array(3); // create a new array with a length of 3
const myThirdArray = new Array("a", "b", "c"); // create a new array with three
elements
```

One potential pitfall of using `new Array()` is that it can be ambiguous when you pass a single argument to the constructor. For example, `new Array(3)` creates an array with a length of 3, but `new Array("3")` creates an array with a single element, the string "3". This is because the argument is treated as the value of the first element when it's a non-negative integer, but as the length of the array when it's a string or a negative integer.
In summary, `[]` is the preferred way to create a new array in JavaScript, while `new Array()` is an alternative way that can be used when you need more control over the array's length or contents.

8. Regular Expression

Q 8.1. What is a RegExp object?

A regular expression is an object that describes a pattern of characters.

The JavaScript `RegExp` class represents regular expressions, and both String and `RegExp` define methods that use regular expressions to perform powerful pattern-matching and search-and-replace functions on text.
Syntax:

```
// Using literal notation
let pattern = /pattern/attributes;

// Using RegExp Object
let pattern = new RegExp(pattern, attributes);


// * pattern – A string that specifies the pattern of the regular expression or
another regular expression.
// * attributes – An optional string containing any of the "g", "i", and "m"
attributes that specify global, case-insensitive, and multi-line matches,
respectively.
```

Example:

```
let pattern = /ab+c/i; // literal notation

let pattern = new RegExp(/ab+c/, 'i') // constructor with regular expression
literal as first argument
```

Q 8.2. What are the string method available in regular expression?

Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects. These patterns are used with the `exec()` and `test()` methods of `RegExp`, and with the `match()`, `matchAll()`, `replace()`, `replaceAll()`, `search()`, and `split()` methods of String.

Example 01: `test()`

Tests for a match in a string. It returns `true` or `false`

```
let exp = /Hello/;
let res1 = exp.test("Hello World");
let res2 = exp.test("Hi");

console.log(res1); // true
console.log(res2); // false
```

Example 02: `exec()`

Executes a search for a match in a string. It returns an array of information or `null` on a mismatch.

```
let res1 = exp.exec("Hello World");
let res2 = exp.exec("Hi");

console.log(res1); // ['Hello', index: 0, input: 'Hello World', groups: undefined]
console.log(res2); // null
```

Method	Description
exec()	Executes a search for a match in a string. It returns an array of information or `null` on a mismatch.
test()	Tests for a match in a string. It returns `true` or `false`.
match()	Returns an array containing all of the matches, including capturing groups, or `null` if no match is found.

Method	Description
matchAll()	Returns an iterator containing all of the matches, including capturing groups.
search()	Tests for a match in a string. It returns the index of the match, or `-1` if the search fails.
replace()	Executes a search for a match in a string, and replaces the matched substring with a replacement substring.
replaceAll()	Executes a search for all matches in a string, and replaces the matched substrings with a replacement substring.
split()	Uses a regular expression or a fixed string to break a string into an array of substrings.

✯ **Try this example on CodeSandbox**

Q 8.3. What are modifiers in regular expression?

Modifiers can be used to perform case-insensitive and global searches.

Modifier	Description
i	Perform case-insensitive matching
g	Perform a global match rather than stops at first match
m	Perform multiline matching

Example 01: Global Search

```
let text = "Hello World! Hello World!";
let pattern = /Hello/g;

console.log(text.match(pattern)); // ['Hello', 'Hello']
```

Example 02: Case-insensitive match

```
let string = "Hello World!";
let pattern = /WORLD/i;
```

```
console.log(string.match(pattern2)); // ['World', index: 6, input: 'Hello World!',
groups: undefined]
```

Example 03: Multiline match

The "m" modifier specifies a multiline match. It only affects the behavior of start `^` and end `$`. `^` specifies a match at the start of a string. `$` specifies a match at the end of a string.

```
let paragraph = `Lorem Ipsum is simply dummy text of the printing and typesetting
industry.`;

let pattern = /Lorem/m;

console.log(paragraph.match(pattern3)); // ["Lorem"]
```

🌟 **Try this example on CodeSandbox**

Q 8.4. What are regular expression patterns?

Regular Expressions provided group of patterns in order to match characters. Basically they are categorized into 3 types,

Brackets:

These are used to find a range of characters.

- **[...]**: Any one character between the brackets.
- **[^...]**: Any one character not between the brackets.
- **[0-9]**: It matches any decimal digit from **0** through **9**.
- **[a-z]**: It matches any character from lowercase **a** through lowercase **z**.
- **[A-Z]**: It matches any character from uppercase **A** through uppercase **Z**.
- **[a-Z]**: It matches any character from lowercase **a** through uppercase **Z**.

Metacharacters:

These are characters with a special meaning

- **.**: a single character
- **\s**: a whitespace character (space, tab, newline)
- **\S**: non-whitespace character
- **\d**: a digit (0-9)

- **\D**: a non-digit
- **\w**: a word character (a-z, A-Z, 0-9, _)
- **\W**: a non-word character
- **[\b]**: a literal backspace (special case).
- **[aeiou]**: matches a single character in the given set
- **[^aeiou]**: matches a single character outside the given set
- **(foo|bar|baz)**: matches any of the alternatives specified

Quantifiers:

These are useful to define quantities

- **p+**: It matches any string containing one or more p's.
- **p: It matches any string containing zero or more p's.
- **p?**: It matches any string containing at most one p.
- **p{N}**: It matches any string containing a sequence of **N** p's
- **p{2,3}**: It matches any string containing a sequence of two or three p's.
- **p{2, }**: It matches any string containing a sequence of at least two p's.
- **p$**: It matches any string with p at the end of it.
- **^p**: It matches any string with p at the beginning of it.

Example:

```
// Brackets
"Hello World".match(/[a-d]/); // -> matches 'a'
"Hello World".match(/[A-D]/); // -> no match
"Hello World".match(/[A-D]/i); // -> matches 'a'

// Metacharacters
"Hello World".match(/[A-Za-z]\s[A-Za-z]/); // -> matches
"Hello World".match(/[0-9]\s[A-Za-z]/); // -> no match

// Quantifiers
"Hello".match(/l+/); // -> matches
"Hello".match(/A*/); // -> no match
```

☆ [Try this example on CodeSandbox]

Q 8.5. How do you search a string for a pattern?

1. Using test() It searches a string for a pattern, and returns `true` or `false`, depending on the result.

```
let re1 = /Hi/;
let re2 = /you/;

re1.test("How are you?"); // false
re2.test("How are you?"); // true
```

2. Using exec() It searches a string for a specified pattern, and returns the found text as an object. If no match is found, it returns an empty (null) object.

```
let re1 = /Hi/;
let re2 = /you/;

re1.exec("How are you?"); // null
re2.exec("How are you?"); // ["you"]
```

☆ **Try this example on CodeSandbox**

Q 8.6. What is the purpose of exec method?

The purpose of exec method is similar to test method but it returns a founded text as an object instead of returning true/false.

```
// Using test() method
var pattern = /you/;
console.log(pattern.test("How are you?")); // true


// Using exec() method
var pattern = /you/;
console.log(pattern.exec("How are you?")); // ["you", index: 8, input: "How are
you?", groups: undefined]
```

Q 8.7. How do you validate an email in javascript?

The `test()` method returns `true` if there is a match in the string with the regex pattern. The regular expression (regex) describes a sequence of characters used for defining a search pattern

```
// Program to validate the email address

function validateEmail(email) {
```

```
  // regex pattern for email
  const re = /\S+@\S+\.\S+/g;

  // check if the email is valid
  let result = re.test(email);
  if (result) {
    console.log("Valid");
  } else {
    console.log("Not valid.");
  }
}

let email = "pradeep.kumar@gmail.com";
let email2 = "pradeep.kumar.com";

validateEmail(email); // Valid
validateEmail(email2); // Not Valid
```

🌟 **Try this example on CodeSandbox**

Q 8.8. How do you detect a mobile browser using regexp?

You can detect mobile browser by simply running through a list of devices and checking if the useragent matches anything. This is an alternative solution for RegExp usage,

```
function detectMobile() {
  if (
    navigator.userAgent.match(/Android/i) ||
    navigator.userAgent.match(/webOS/i) ||
    navigator.userAgent.match(/iPhone/i) ||
    navigator.userAgent.match(/iPad/i) ||
    navigator.userAgent.match(/iPod/i) ||
    navigator.userAgent.match(/BlackBerry/i) ||
    navigator.userAgent.match(/Windows Phone/i)
  ) {
    return true;
  } else {
    return false;
  }
}
```

9. FUNCTIONS

Q 9.1. What are the benefits of using arrow function over es5 function?

Arrows is a new syntax for functions, which brings several benefits:

- Arrow syntax automatically binds `this` to the surrounding code's context
- The syntax allows an implicit return when there is no body block, resulting in shorter and simpler code in some cases
- Last but not least, `=>` is shorter and simpler than `function`, although stylistic issues are often subjective

Example 01: Arrow Function with No Argument

If a function doesn't take any argument, then you should use empty parentheses.

```
let greet = () => console.log('Hello');
greet(); // Hello
```

Example 02: Arrow Function with One Argument

If a function has only one argument, you can omit the parentheses.

```
let greet = x => console.log(x);
greet('Hello'); // Hello
```

Example 03: Arrow Function as an Expression

You can also dynamically create a function and use it as an expression.

```
let age = 25;

let welcome = (age < 18) ?
  () => console.log('Baby') :
  () => console.log('Adult');

welcome(); // Adult
```

Example 04: Multiline Arrow Functions

If a function body has multiple statements, you need to put them inside curly brackets `{}`.

```
let area = (r) => {
  const pi = 3.14;
  return pi * r * r;
}

let result = area(10);
console.log(result); // 314
```

Note: Unlike regular functions, arrow functions do not have their own `this`. The value of `this` inside an arrow function remains the same throughout the lifecycle of the function and is always bound to the value of `this` in the closest non-arrow parent function.

☆ **Try this example on CodeSandbox**

Q 9.2. What is the benefit of using the arrow syntax for a method in a constructor?

The main advantage of using an arrow function as a method inside a constructor is that the value of `this` gets set at the time of the function creation and can't change after that. So, when the constructor is used to create a new object, `this` will always refer to that object.

```
const Person = function(firstName) {
  this.firstName = firstName;
  this.sayName1 = function() { console.log(this.firstName); };
  this.sayName2 = () => { console.log(this.firstName); };
};

const john = new Person('John');
const dave = new Person('Dave');

john.sayName1(); // John
john.sayName2(); // John

// The regular function can have its 'this' value changed, but the arrow function
cannot
john.sayName1.call(dave); // Dave (because "this" is now the dave object)
john.sayName2.call(dave); // John

john.sayName1.apply(dave); // Dave (because 'this' is now the dave object)
john.sayName2.apply(dave); // John

john.sayName1.bind(dave)(); // Dave (because 'this' is now the dave object)
john.sayName2.bind(dave)(); // John

var sayNameFromWindow1 = john.sayName1;
sayNameFromWindow1(); // undefined (because 'this' is now the window object)
```

```
var sayNameFromWindow2 = john.sayName2;
sayNameFromWindow2(); // John
```

Q 9.3. Difference between `Function`, `Method` and `Constructor` calls in JavaScript?

1. Functions: The simplest usages of function call:

```
function display(name) {
  return "Hello " + name;
}

display("World"); // "Hello World"
```

2. Methods: in JavaScript are nothing more than object properties that are functions.

```
var obj = {
  display : function() {
    return "Hello " + this.name;
  },
  name: 'Minali Peri'
}
obj.display();  // "Hello Minali Peri"
```

3. Constructors: Like function and method, `constructors` are defined with function.

```
function Employee(name, age) {
  this.name = name;
  this.age = age;
}

var emp1 = new Employee('Drishya Sama', 28);
emp1.name; // "Drishya Sama"
emp1.age; // 28
```

Unlike function calls and method calls, a constructor call `new Employee('Drishya Sama', 28)` creates a new object and passes it as the value of `this`, and implicitly returns the new object as its result. The primary role of the constructor function is to initialize the object.

Q 9.4. When you should not use arrow functions in ES6?

An arrow function is a shorter syntax for a function expression and does not have its own **this, arguments, super, or new.target**. These function are best suited for non-method functions, and they cannot be used as constructors.

Arrow functions in ES6 has two limitations:

- Do not work with new
- Fixed this bound to scope at initialisation

When should not use Arrow Functions:

1. Object methods:

The counter object has two methods: current() and next(). The current() method returns the current counter value and the next() method returns the next counter value.

```
// Usin arrow function
const counter = {
  count: 0,
  next: () => ++this.count,
  current: () => this.count
};

console.log(counter.next()); // NaN
```

2. Event handlers:

If we click the button, we would get a TypeError. It is because this is not bound to the button, but instead bound to its parent scope.

```
let button = document.getElementById('press');

button.addEventListener('click', () => {
  this.classList.toggle('on');
});
```

3. Prototype methods:

The `this` value in these `next()` and `current()` methods reference the global object. Since the `this` value inside the methods to reference the Counter object, it needs to use the regular functions instead

```
function Counter() {
    this.count = 0;
}

Counter.prototype.next = () => {
    return this.count;
```

```
};

Counter.prototype.current = () => {
    return ++this.next;
}
```

4. Functions that use the arguments object:

Arrow functions don't have the arguments object. Therefore, if a function that uses arguments object, you cannot use the arrow function.

```
const concat = (separator) => {
    let args = Array.prototype.slice.call(arguments, 1);
    return args.join(separator);
}
```

🌟 **Try this example on CodeSandbox**

Q 9.5. What are the properties of function objects in javascript?

JavaScript function objects are used to define a piece of JavaScript code. This code can be called within a JavaScript code as and when required.

Javascript Function Objects Property:

Name	Description
arguments	An array corresponding to the arguments passed to a function.
arguments.callee	Refers the currently executing function.
arguments.length	Refers the number of arguments defined for a function.
constructor	Specifies the function that creates an object.
length	The number of arguments defined by the function.
prototype	Allows adding properties to a Function object.

Q 9.6. What is a first class function?

In javaScript, functions can be stored as a variable inside an object or an array as well as it can be passed as an argument or be returned by another function. That makes function **first-class function** in JavaScript.

Example 01: Assign a function to a variable

```
const message = function() {
   console.log("Hello World!");
}

message(); // Invoke it using the variable
```

Example 02: Pass a function as an Argument

```
function sayHello() {
   return "Hello, ";
}
function greeting(helloMessage, name) {
  console.log(helloMessage() + name);
}
// Pass `sayHello` as an argument to `greeting` function
greeting(sayHello, "JavaScript!");
```

Example 03: Return a function

```
function sayHello() {
   return function() {
      console.log("Hello!");
   }
}
```

Example 04: Using a variable

```
const sayHello = function() {
   return function() {
      console.log("Hello!");
   }
}
const myFunc = sayHello();
myFunc();
```

Example 05: Using double parentheses

```
function sayHello() {
```

```
   return function() {
      console.log("Hello!");
   }
}
sayHello()();
```

We are using double parentheses `()()` to invoke the returned function as well.
🌟 **Try this example on CodeSandbox**

Q 9.7. What is a higher order function?

A Higher-Order function is a function that receives a function as an argument or returns the function as output.

For example, `Array.prototype.map()`, `Array.prototype.filter()` , `Array.prototype.forEach()` and `Array.prototype.reduce()` are some of the Higher-Order functions in javascript.
Example 01: .map()

```
const array = [10, 20, 30];

const result = array.map(function (item) {
  return item * 2;
});
console.log(result); // [20, 40, 60]
```

Example 02: .filter()

```
const randomNumbers = [4, 11, 42, 14, 39];

const filteredArray = randomNumbers.filter((number) => {
  return number > 15;
});
console.log(filteredArray); // [42, 39]
```

Example 03: .forEach()

```
const numbers = [28, 77, 45];

numbers.forEach((number) => {
  console.log(number);
});
```

Example 04: .reduce()

```
const arrayOfNumbers = [10, 20, 30];

const sum = arrayOfNumbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue;
});
console.log("Sum: " + sum); // 60
```

🌟 **Try this example on CodeSandbox**

Q 9.8. What is a unary function?

Unary function (i.e. monadic) is a function that accepts exactly one argument. It stands for single argument accepted by a function.

```
// Unary function
const unaryFunction = (number) => number + 10;

console.log(unaryFunction(10)); // 20
```

🌟 **Try this example on CodeSandbox**

Q 9.9. What is currying function?

Currying is the process of taking a function with multiple arguments and turning it into a sequence of functions each with only a single argument.

In other words, when a function, instead of taking all arguments at one time, takes the first one and return a new function that takes the second one and returns a new function which takes the third one, and so forth, until all arguments have been fulfilled.

```
// Normal function
const add = (a, b, c) => {
  return a + b + c;
};
console.log(add(10, 10, 10)); // 30

// Currying function
const addCurry = (a) => {
  return (b) => {
    return (c) => {
      return a + b + c;
    };
```

```
  };
};
console.log(addCurry(20)(20)(20)); // 60
```

Note: Curried functions are great to improve code re-usability and functional composition.

🌟 **Try this example on CodeSandbox**

Q 9.10. What is a pure function?

Pure functions are functions that accept an input and returns a value without modifying any data outside its scope(Side Effects). Its output or return value must depend on the input/arguments and pure functions must return a value.

Example: Pure Function

It is a pure function because you always get a Hello `<name>` as output for the `<name>` pass as an input.

```
// Pure Function

function sayGreeting(name) {
  return `Hello ${name}`;
}

console.log(sayGreeting("World"));
```

Example: Not Pure Function

The function's output now depends on an outer state called greeting. What if someone changes the value of the greeting variable to `Hola`? It will change the output of the `sayGreeting()` function even when you pass the same input.

```
let greeting = "Hello";

function sayGreeting(name) {
  return `${greeting} ${name}`;
}
```

A function must pass two tests to be considered **pure**:

- Same inputs always return same outputs
- No side-effects

Benefits:

- **Predictable**: It produces a predictable output for the same inputs.
- **Readable**: Anyone reading the function as a standalone unit can understand its purpose completely.
- **Reusable**: Can reuse the function at multiple places of the source code without altering its and the caller's behavior.
- **Testable**: We can test it as an independent unit.

Q 9.11. What is memoization in JavaScript?

Memoization is a programming technique which attempts to increase a function's performance by **caching** its previously computed results.

Each time a memoized function is called, its parameters are used to index the cache. If the data is present, then it can be returned, without executing the entire function. Otherwise the function is executed and then the result is added to the cache.

```
// Memoized function to Add Number

const memoizedAdd = () => {
  let cache = {};
  return (number) => {
    if (number in cache) {
      console.log('Fetching from cache: ');
      return cache[number];
    }
    else {
      console.log('Calculating result: ');
      let result = number + 10;
      cache[number] = result;
      return result;
    }
  }
}
// returned function from memoizedAdd
const sum = memoizedAdd();

console.log(sum(10)); // Calculating result: 20
console.log(sum(10)); // Fetching from cache: 20
```

🌟 **Try this example on CodeSandbox**

Q 9.12. What is an arguments object?

The arguments object is an Array-like object (`arguments`) accessible inside functions that contains the values of the arguments passed to that function.
Example:

```
/**
 * Arguments Object
 */

function sum() {
  let total = 0;
  for (let i = 0, len = arguments.length; i < len; ++i) {
    total += arguments[i];
  }
  return total;
}

sum(10, 20, 30); // returns 60
```

⭐ **Try this example on CodeSandbox**

Q 9.13. What is the way to find the number of parameters expected by a function?

The **length** property indicates the number of parameters expected by the function.

```
// function.length

function fun1() {}
console.log(fun1.length); // 0

function fun2(arg1, arg2) {}
console.log(fun2.length); // 2
```

⭐ **Try this example on CodeSandbox**

Q 9.14. What is the difference between Call, Apply and Bind?

1. Call: invokes the function and allows you to pass in arguments one by one.

Example:

```
const employee1 = { firstName: "Sahima", lastName: "Mutti" };
const employee2 = { firstName: "Aarush", lastName: "Krishna" };

function say(greeting) {
  console.log(greeting + " " + this.firstName + " " + this.lastName);
}

say.call(employee1, "Hi");    // Hi Sahima Mutti
say.call(employee2, "Hello"); // Hello Aarush Krishna
```

2. Apply: invokes the function and allows you to pass in arguments as an array.

Example:

```
const employee1 = { firstName: "Sahima", lastName: "Mutti" };
const employee2 = { firstName: "Aarush", lastName: "Krishna" };

function say(greeting) {
  console.log(greeting + " " + this.firstName + " " + this.lastName);
}

say.apply(employee1, ["Hi"]);    // Hi Sahima Mutti
say.apply(employee2, ["Hello"]); // Hello Aarush Krishna
```

3. Bind: returns a new function, allowing you to pass in a this array and any number of arguments.

Example:

```
const employee1 = { firstName: "Sahima", lastName: "Mutti" };
const employee2 = { firstName: "Aarush", lastName: "Krishna" };

function say(greeting) {
  console.log(greeting + " " + this.firstName + " " + this.lastName);
}

var sayEmployee1 = say.bind(employee1);
var sayEmployee2 = say.bind(employee2);

sayEmployee1("Hi");    // Hi Sahima Mutti
sayEmployee2("Hello"); // Hello Aarush Krishna
```

🌟 **Try this example on CodeSandbox**

Q 9.15. What is bind method in javascript?

The `bind()` method creates a new function, when invoked, has the `this` sets to a provided value. The `bind()` method allows an object to borrow a method from

another object without making a copy of that method. This is known as function **borrowing** in JavaScript.
Example:

```
/**
 * bind() function
 */
const person = {
  firstName: "Chhavi",
  lastName: "Goswami",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  }
};

const member = {
  firstName: "Vasuda",
  lastName: "Sahota"
};

let fullName = person.fullName.bind(member);
console.log(fullName()); // Vasuda Sahota
```

🌟 **Try this example on CodeSandbox**

Q 9.16. What is an anonymous function?

An anonymous function is a function without a name. Anonymous functions are commonly assigned to a variable name or used as a callback function.

Example 01: Anonymous function

```
let show = function () {
  console.log("Anonymous function");
};
show();
```

Example 02: anonymous functions as arguments

```
setTimeout(function () {
  console.log("Execute later after 1 second");
}, 1000);
```

Example 03: Immediately invoked function execution

```
const person = {
  firstName: "Ayaan",
```

```
  lastName: "Memon"
};

(function () {
  console.log(person.firstName + " " + person.lastName); // Ayaan Memon
})(person);
```

Example 04: Arrow functions

```
let add = (a, b) => a + b;

add(10, 20); // 30
```

🌟 **Try this example on CodeSandbox**

Q 9.17. Explain the difference between `function foo() {}` and `var foo = function() {}`?

1. Function Declaration:

Function declarations are evaluated upon entry into the enclosing scope, before any step-by-step code is executed. The function's name (`foo`) is added to the enclosing scope.

```
foo(); // Function Declaration Example!

function foo() {
  console.log("Function Declaration Example!");
}
```

2. Function Expression:

Function expressions are evaluated as part of the step-by-step code, at the point where they appear. That one creates a function with no name, which it assigns to the foo variable.

```
foo(); // TypeError: foo is not a function

var foo = function() {
  console.log(typeof foo); // function
};
console.log(typeof foo);     // undefined
```

🌟 **Try this example on CodeSandbox**

Q 9.18. When to use function declarations and expressions in JavaScript?

Function Declarations:

A declared function is "saved for later use", and will be executed later, when it is invoked (called).

```
// Function declaration
function add(num1, num2) {
    return num1 + num2;
}
```

function is only declared here. For using it, it must be invoked using function name. e.g add(10, 20);

Function Expression:

A function expression can be stored in a variable:

```
// Function expression
var add = function (num1, num2) {
    return num1 + num2;
};
```

After a function expression has been stored in a variable, the variable can be used as a function. Functions stored in variables do not need function names. They are always invoked (called) using the variable name.

Difference:

- `Function declarations` load before any code is executed while `Function expressions` load only when the interpreter reaches that line of code.
- Similar to the `var` statement, function declarations are hoisted to the top of other code. Function expressions aren't hoisted, which allows them to retain a copy of the local variables from the scope where they were defined.

Benefits of Function Expressions:

There are several different ways that function expressions become more useful than function declarations.

- As closures
- As arguments to other functions

- As Immediately Invoked Function Expressions (IIFE)

Q 9.19. What is the difference between a method and a function in javascript?

1. Function:

A function is a piece of code that is called by name and function itself not associated with any object and not defined inside any object. It can be passed data to operate on (i.e. parameter) and can optionally return value.

Example:

```
// Function
function message(msg) {
  return msg;
}

// Call the function
message("Welcome to JavaScript");
```

Here, `message()` function call is not associated with `object` hence not invoked through any object.

2. Method:

A JavaScript method is a property of an object that contains a function definition. Methods are functions stored as object properties.

Example:

```
// Method
let employee = {
  firstName: "Ajay",
  lastName: "Nagi",
  getName: function () {
    return "Employe Name: " + this.firstName + " " + this.lastName;
  }
};

// Call the method
console.log(employee.getName());
```

Here `employee` is an object and `getName` is a method which is associated with `employee`.

☆ **Try this example on CodeSandbox**

Q 9.20. What is Function binding?

Function binding (`.bind()`) is a method on the prototype of all functions in JavaScript. It allows to create a new function from an existing function, change the new function's `this` context, and provide any arguments you want the new function to be called with. The arguments provided to `bind` will precede any arguments that are passed to the new function when it is called.

Example:

```
// Function Binding
const person = {
  firstName: "Nirupama",
  lastName: "Randhawa",
  getName: function () {
    return this.firstName + " " + this.lastName;
  }
};

const member = {
  firstName: "Alisha",
  lastName: "Chhabra"
};

let getName = person.getName.bind(member);
console.log(getName()); // Alisha Chhabra
```

✯ **Try this example on CodeSandbox**

Q 9.21. Explain how `this` works in JavaScript?

The `this` keyword refers to an `object`. Which object depends on how this is being invoked (used or called). The `this` keyword refers to different objects depending on how it is used.

- In an object method, `this` refers to the object.
- Alone, `this` refers to the global object.
- In a function, `this` refers to the global object.
- In a function, in strict mode, `this` is `undefined`.
- In an event, `this` refers to the element that received the event.
- Methods like `call()`, `apply()`, and `bind()` can refer `this` to any object.

Example:

```
// this keyword in object method

const person = {
  firstName: "Nirupama",
  lastName: "Randhawa",
  getName: function () {
    return this.firstName + " " + this.lastName;
  }
};
```

Q 9.22. What is generator in JS?

Generator-Function:

A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the `yield` keyword rather than `return`.
The `yield` statement suspends function's execution and sends a value back to caller, but retains enough state to enable function to resume where it is left off. When resumed, the function continues execution immediately after the last `yield` run.

Syntax:

```
function* gen() {
     yeild 1;
     yeild 2;
     ...
     ...
}
```

Generator-Object:

Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a "for in" loop.

Example:

```
// Generate Function

function* fun() {
  yield 10;
  yield 20;
  yield 30;
}

// Calling the Generate Function
var gen = fun();
```

```
gen.next().value; // 10
gen.next().value; // 20
gen.next().value; // 30
```

☆ [Try this example on CodeSandbox]

Q 9.23. Compare Async-Await and Generators usage to achive same functionality?

1. Generators/Yield:

Generators are objects created by generator functions — functions with an * (asterisk) next to their name. The yield keyword pauses generator function execution and the value of the expression following the yield keyword is returned to the generator's caller. It can be thought of as a generator-based version of the return keyword.

```
// Generator function

const generator = (function* () {
  // waiting for .next()
  const number = yield 5;
  // waiting for .next()
  console.log(number); // => 15
})();

console.log(generator.next()); // => { done: false, value: 5 }
console.log(generator.next(15)); // => { done: true, value: undefined }
```

2. Async/Await:

Async keyword is used to define an asynchronous function, which returns a `AsyncFunction` object.
Await keyword is used to pause async function execution until a `Promise` is fulfilled, that is resolved or rejected, and to resume execution of the `async` function after fulfillments. When resumed, the value of the `await` expression is that of the fulfilled `Promise`.
Key points:

1. Await can only be used inside an async function.
2. Functions with the async keyword will always return a promise.
3. Multiple awaits will always run in sequential order under a same function.

4. If a promise resolves normally, then await promisereturns the result. But in case of a rejection it throws the error, just if there were a throw statement at that line.
5. Async function cannot wait for multiple promises at the same time.
6. Performance issues can occur if using await after await as many times one statement doesn't depend on the previous one.

```
// Async/Await

async function asyncFunction() {
  const promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("resolved!"), 1000);
  });

  const result = await promise;
  // wait till the promise resolves (*)

  console.log(result); // "resolved!"
}

asyncFunction();
```

Generator and Async-await — Comparison:

1. Generator functions/yield and Async functions/await can both be used to write asynchronous code that "waits", which means code that looks as if it was synchronous, even though it really is asynchronous.
2. Generator function are executed yield by yield i.e one yield-expression at a time by its iterator (the next method) where as Async-await, they are executed sequential await by await.
3. Async/await makes it easier to implement a particular use case of Generators.
4. The return value of Generator is always {value: X, done: Boolean} where as for Async function it will always be a promise that will either resolve to the value X or throw an error.
5. Async function can be decomposed into Generator and promise implementation which are good to know stuff.

🌟 **Try this example on CodeSandbox**

Q 9.24. How do you compare two date objects?

Two dates can be compared by converting them into numeric values using `date.getTime()` method to correspond to their time. Also, the relational operators `<`, `<=`, `>`, `>=` can be used to compare JavaScript dates.
However, the equality operators `==`, `!=`, `===`, `!==` cannot be used to compare (the value of) dates because:

- Two distinct objects are never equal for either strict or abstract comparisons.
- An expression comparing Objects is only true if the operands reference the same Object.

Example:

```
// Using getTime()
let d1 = new Date();
let d2 = new Date(d1);

console.log(d1.getTime() === d2.getTime()); // true

// Using '<' and '>'
let d3 = new Date(2022, 10, 31);
let d4 = new Date(2022, 10, 30);

console.log(d3 < d4); // true
```

🌟 **Try this example on CodeSandbox**

Q 9.25. What are closures?

A closure is the combination of a function and the lexical environment within which that function was declared. i.e, it is an inner function that has access to the outer or enclosing function's variables.

Closure is useful in hiding implementation detail in JavaScript. In other words, it can be useful to create private variables or functions.

1. Lexical Scope:

In lexical scoping free variables must belong to a parent scope.

Example:

```
/**
 * Lexical Scope
 **/
```

```
function init() {
  let name = "JavaScript closures"; // name is a local variable created by init
  function displayName() {
    // displayName() is the inner function, a closure
    console.log(name); // use variable declared in the parent function
  }
  return displayName;
}
init();

var closure = init();
closure();
```

As per the above code, the inner `function displayName()` has access to the variables in the outer `function init()`.
🌟 **Try this example on CodeSandbox**

2. Dynamic Scope:

In dynamic scoping free variables must belong to the calling scope

Example:

```
/**
 * Dynamic Scope
 **/

function fun1() {
  console.log(a); // 10
}

function fun2() {
  var a = 20;
  fun1();
}

var a = 10;
fun2();

// Output
10
```

🌟 **Try this example on CodeSandbox**

Q 9.26. What is callback() function in javascript?

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

```
// callback() function

function greeting(name) {
  alert('Hello ' + name);
}

function processUserInput(callback) {
  var name = prompt('Please enter your name:');
  callback(name);
}

processUserInput(greeting);
```

The above example is a synchronous callback, as it is executed immediately.

Note: callbacks are often used to continue code execution after an asynchronous operation has completed — these are called asynchronous callbacks..

🌟 **Try this example on CodeSandbox**

Q 9.27. How to avoid callback hell in javascript?

Callback hell is a phenomenon that afflicts a JavaScript developer when he tries to execute multiple asynchronous operations one after the other. Some people call it to be the **pyramid of doom**.

Example:

```
doSomething(param1, param2, function(err, paramx){
    doMore(paramx, function(err, result){
        insertRow(result, function(err){
            yetAnotherOperation(someparameter, function(s){
                somethingElse(function(x){
                });
            });
        });
    });
});
```

Techniques for avoiding callback hell:

- Write comments
- Split functions into smaller functions
- Using Async.js
- Using Promises
- Using Async-Await

Q 9.28. How do you encode an URL?

The encodeURI() function is used to encode complete URI which has special characters except (`,`, `/`, `?`, `:`, `@`, `&`, `=`, `+`, `$`, `#`) characters.

```
var uri = 'https://mozilla.org/?x=шеллы';
var encoded = encodeURI(uri);

console.log(encoded); // https://mozilla.org/?x=%D1%88%D0%B5%D0%BB%D0%BB%D1%8B
```

Q 9.29. How do you decode an URL?

The decodeURI() function is used to decode a Uniform Resource Identifier (URI) previously created by encodeURI().

```
var uri = 'https://mozilla.org/?x=шеллы';
var encoded = encodeURI(uri);

console.log(encoded); // https://mozilla.org/?x=%D1%88%D0%B5%D0%BB%D0%BB%D1%8B
try {
  console.log(decodeURI(encoded)); // "https://mozilla.org/?x=шеллы"
} catch(e) { // catches a malformed URI
  console.error(e);
}
```

Q 9.30. How function overloading works in JavaScript?

Function overloading refers to the ability to define multiple functions with the same name but with different parameters. In many programming languages, the function to be executed is determined at compile time based on the parameters provided. However, in JavaScript, function overloading does not work in the same way because JavaScript functions can be called with any number and type of arguments.

One way to achieve function overloading in JavaScript is by using conditional statements to determine the appropriate behavior based on the arguments passed to the function.

Example

```
function myFunction() {
  if (arguments.length === 1) {
    console.log("Hello " + arguments[0]);
  } else if (arguments.length === 2) {
    console.log("Hello " + arguments[0] + " and " + arguments[1]);
  } else {
    console.log("Hello world");
  }
}

myFunction(); // output: "Hello world"
myFunction("Alice"); // output: "Hello Alice"
myFunction("Bob", "Charlie"); // output: "Hello Bob and Charlie"
```

10. EVENTS

Q 10.1. What is event handling in javascript?

The change in the state of an object is known as an **Event**. In html, there are various events which represents that some activity is performed by the user or by the browser.

When javascript code is included in HTML, js react over these events and allow the execution. This process of reacting over the events is called **Event Handling**. Thus, js handles the HTML events via **Event Handlers**.

Some of the HTML event handlers are:

Mouse events:

Event Handler	Description
onclick	When mouse click on an element
onmouseover	When the cursor of the mouse comes over the element
onmouseout	When the cursor of the mouse leaves an element

Event Handler	Description
onmousedown	When the mouse button is pressed over the element
onmouseup	When the mouse button is released over the element
onmousemove	When the mouse movement takes place.

Form events:

Event Handler	Description
onfocus	When the user focuses on an element
onsubmit	When the user submits the form
onblur	When the focus is away from a form element
onchange	When the user modifies or changes the value of a form element

Window/Document events:

Event Handler	Description
onload	When the browser finishes the loading of the page
onunload	When the visitor leaves the current webpage, the browser unloads it
onresize	When the visitor resizes the window of the browser

Example: Click Event

```
<!DOCTYPE html>
<html>
  <head>
```

```
    <script>
      function greeting() {
        alert("Hello! Good morning");
      }
    </script>
  </head>
  <body>
    <h2>Click Event Example</h2>
    <button type="button" onclick="greeting()">Click me</button>
  </body>
</html>
```

☆ Try this example on CodeSandbox

Q 10.2. How to create and trigger events in javascript?

Events can be handled either through `addEventListener()` method or we can trigger events on individual components by defining specific JavaScript functions.
Syntax:

```
document.addEventListener(event, function, phase)
```

Example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Click Event</title>
    <meta charset="UTF-8" />
  </head>
  <body>
    <h3>Click on the page to trigger click event</h3>
    <script type="text/javascript">
      document.addEventListener("click", function () {
        console.log("You clicked inside the document");
      });
    </script>
  </body>
</html>
```

☆ Try this example on CodeSandbox

Q 10.3. What is an event delegation?

Event Delegation is basically a pattern to handle events efficiently. Instead of adding an event listener to each and every similar element, we can add an event listener to a parent element and call an event on a particular target using the `event.target` property of the event object.

Example:

```
<div id="buttons">
  <button class="buttonClass">Click me</button>
  <button class="buttonClass">Click me</button>
  <button class="buttonClass">Click me</button>
</div>

<script>
  document.getElementById('buttons')
    .addEventListener('click', event => {
      if (event.target.className === 'buttonClass') {
        console.log('Click!');
      }
    });
</script>
```

🌟 **Try this example on CodeSandbox**

Q 10.4. What is an event flow?

Event flow is the order in which event is received on the web page. When you click an element that is nested in various other elements, before your click actually reaches its destination, or target element, it must trigger the click event each of its parent elements first, starting at the top with the global window object.

There are two ways of event flow

- Top to Bottom (Event Capturing)
- Bottom to Top (Event Bubbling)

Q 10.5. What is event bubbling?

Event bubbling is a type of event propagation where the event first triggers on the innermost target element, and then successively triggers on the ancestors (parents)

of the target element in the same nesting hierarchy till it reaches the outermost DOM element.

Example: If you click on EM, the handler on DIV runs.

```
<div onclick="alert('The handler!')">
  <em>If you click on <code>EM</code>, the handler on <code>DIV</code> runs.</em>
</div>
```

Stopping bubbling:

```
<body onclick="alert(`the bubbling doesn\'t reach here`)">
  <button onclick="event.stopPropagation()">Click me</button>
</body>
```

🌟 **Try this example on CodeSandbox**

Q 10.6. What is event capturing?

Event capturing is a type of event propagation where the event is first captured by the outermost element and then successively triggers on the descendants (children) of the target element in the same nesting hierarchy till it reaches the inner DOM element.

Example:

```
<article id="ancestor">
    Article Element
    <div id="parent">
      DIV Element
      <p id="child">
        P Element
      </p>
    </div>
</article>

<script>
  // Script to click event handler to capture on each element
  for (let elem of document.querySelectorAll("*")) {
    elem.addEventListener(
      "click",
      (e) => console.log("Capturing:", elem.tagName),
      true
    );
  }
</script>
```

🌟 **Try this example on CodeSandbox**

Q 10.7. How do you submit a form using JavaScript?

Generally, a form is submitted when the user presses a submit button. JavaScript provides the form object that contains the `submit()` method. Use the "id" of the form to get the form object.
Example:

```
<form id="myForm" action="/action_page.php">
  Search: <input type='text' name='query' />
  <input type="button" onclick="handleSubmit()" value="Submit form">
</form>

<script>
function handleSubmit() {
  document.getElementById("myForm").submit();
}
</script>
```

Q 10.8. What is the purpose of void(0)?

The `void(0)` is used to prevent the page from refreshing. This will be helpful to eliminate the unwanted side-effect, because it will return the `undefined` primitive value.
It is commonly used for HTML document that
uses `href="JavaScript:void(0);"` within an `<a>` element. i.e, when you click a link, the browser loads a new page or refreshes the same page. But this behavior will be prevented using this expression.
Example: the below link notify the message without reloading the page

```
<a href="JavaScript:void(0);" onclick="alert('Prevent the page from
refreshing!')">Click Me!</a>
```

🌟 **Try this example on CodeSandbox**

Q 10.9. What is the use of preventDefault method?

The `preventDefault()` method is used to prevent the browser from executing the default action of the selected element. It can prevent the user from processing the request by clicking the link.
For example, prevent form submission when clicking on submit button and prevent opening the page URL when clicking on hyper link are some common usecases.

```
document.getElementById("link").addEventListener("click", function(event) {
   event.preventDefault();
});
```

Note: Remember that not all events are cancelable.

Q 10.10. What is the use of stopPropagation method?

The `stopPropagation` method is used to stop the event from bubbling up the event chain.
For example, the below nested divs with stopPropagation method prevents default event propagation when clicking on nested div(Div1)

```
<p>Click DIV1 Element</p>
<div onclick="secondFunc()">DIV 2
  <div onclick="firstFunc(event)">DIV 1</div>
</div>

<script>
function firstFunc(event) {
  alert("DIV 1");
  event.stopPropagation();
}

function secondFunc() {
  alert("DIV 2");
}
</script>
```

Q 10.11. What is difference between stoppropagation, stopimmediatepropagation and preventdefault in javascript?

1. event.preventDefault():

This method is used to stop the browser's default behavior when performing an action.

Example:

```
<p>Please click on the checkbox control.</p>

<form>
  <label for="id-checkbox">Checkbox:</label>
  <input type="checkbox" id="id-checkbox" />
</form>

<div id="output-box"></div>

<script>
  document.querySelector("#id-checkbox").addEventListener("click", function(event)
{
  document.getElementById("output-box").innerHTML += "Sorry!
<code>preventDefault()</code> won't let you check this!<br>";
  event.preventDefault();
}, false);
</script>
```

2. event.stopPropagation():

This method is used to prevent the propagation of an event as defined in the capturing/bubbling phase of the flow of an event in the browser.

Example:

```
<div class="parent" (onClick)="console.log('parent')">
    <button class="child" (onClick)="buttonClick(event)"></button>
</div>
<script>
    function buttonClick(event) {
        event.stopPropagation();
        console.log('child');
    }
</script>
```

3. event.stopImmediatePropagation():

With `stopImmediatePropagation()`, along with the event propagation, other event handlers will also be prevented from execution.
As a result, clicking on the div element will:

- Prevent event bubbling to the parent elements
- Prevent the execution of any other event listener attached to the element

Q 10.12. What is the use of setTimeout?

The `setTimeout()` method is used to call a function or evaluates an expression after a specified number of milliseconds.
Syntax:

```
setTimeout(callback function, delay in milliseconds)
```

Example:

```
setTimeout(() => {
  console.log("Delayed for 1 second.");
}, "1000")
```

Q 10.13. What is the use of setInterval?

The `setInterval()` method is used to call a function or evaluates an expression at specified intervals (in milliseconds). The `setInterval()` method continues calling the function until `clearInterval()` is called, or the window is closed.
Example:

```
<p id="timer"></p>

<script>
setInterval(myTimer, 1000);

function myTimer() {
  const date = new Date();
  document.getElementById("timer").innerHTML = date.toLocaleTimeString();
}
</script>
```

☆ [Try this example on CodeSandbox]

Q 10.14. What is the purpose of clearTimeout method?

The `clearTimeout()` function is used in javascript to clear the timeout which has been set by `setTimeout()` function before that. i.e, The return value of setTimeout() function is stored in a variable and it's passed into the `clearTimeout()` function to clear the timer.
For example, the below setTimeout method is used to display the message after 3 seconds. This timeout can be cleared by clearTimeout() method.

```
// clearTimeout()

var msg;
function greeting() {
  console.log("Hello World!");
  stop();
}
function start() {
  console.log("start");
  msg = setTimeout(greeting, 3000);
}
function stop() {
  console.log("stop");
  clearTimeout(msg);
}

start();

// Output
Start
Hello World!
Stop
```

☆ **Try this example on CodeSandbox**

Q 10.15. What is the purpose of clearInterval method?

The `clearInterval()` function is used in javascript to clear the interval which has been set by `setInterval()` function. i.e, The return value returned by setInterval() function is stored in a variable and it's passed into the clearInterval() function to clear the interval.
For example, the below setInterval method is used to display the message for every 3 seconds. This interval can be cleared by clearInterval() method.

```
// clearInterval()

var msg;
function greeting() {
```

```
  console.log("Hello World!");
  stop();
}
function start() {
  console.log("start");
  msg = setInterval(greeting, 3000);
}
function stop() {
  console.log("stop");
  clearInterval(msg);
}

start();

// Output
Start
Hello World!
Stop
```

🌟 **Try this example on CodeSandbox**

Q 10.16. What is the difference between document load and DOMContentLoaded events?

The `DOMContentLoaded` event is fired when the initial HTML document has been completely loaded and parsed, without waiting for assets(stylesheets, images, and subframes) to finish loading. Whereas The load event is fired when the whole page has loaded, including all dependent resources(stylesheets, images).

11. OBJECTS

Q 11.1. What are the possible ways to create objects in JavaScript?

1. Object Constructor:

The simpliest way to create an empty object is using Object constructor. Currently this approach is not recommended.

```
let object = new Object();
```

2. Object create method:

The create method of Object creates a new object by passing the prototype object as a parameter

```
let object = Object.create(null);
```

3. Object Literal:

The object literal syntax is equivalent to create method when it passes `null` as parameter

```
let person = {};
```

4. Function Constructor:

Create any function and apply the new operator to create object instances,

```
function Person(name) {
 let object = {};
 object.name = name;
 object.age = 26;

 return object;
}

let person = new Person("Alex");
```

5. Function Constructor with prototype:

This is similar to function constructor but it uses prototype for their properties and methods,

```
function Person(){}

Person.prototype.name = "Alex";
let person = new Person();
```

6. ES6 Class:

ES6 introduces class feature to create the objects

```
class Person {
 constructor(name) {
    this.name = name;
 }
```

```
}

let person = new Person("Alex");
```

7. Singleton Pattern:

A Singleton is an object which can only be instantiated one time. Repeated calls to its constructor return the same instance and this way one can ensure that they don't accidentally create multiple instances.

```
let object = new function() {
  this.name = "Alex";
}
```

Q 11.2. What are the recommendations to create new object?

It is recommended to avoid creating new objects using `new Object()`. Instead you can initialize values based on it is type to create the objects.

- Assign {} instead of new Object()
- Assign "" instead of new String()
- Assign 0 instead of new Number()
- Assign false instead of new Boolean()
- Assign [] instead of new Array()
- Assign /()/ instead of new RegExp()
- Assign function (){} instead of new Function()

Example:

```
let obj1 = {};
let obj2 = "";
let obj3 = 0;
let obj4 = false;
let obj5 = [];
let obj6 = /()/;
let obj7 = function(){};
```

Q 11.3. What are the different ways to access object properties?

There are 3 possible ways for accessing the property of an object.

1. Dot notation: It uses dot for accessing the properties

```
objectName.property
```

2. Square brackets notation: It uses square brackets for property access

```
objectName["property"]
```

3. Expression notation: It uses expression in the square brackets

```
objectName[expression]
```

Q 11.4. How to check if an object is an array or not?

The `Array.isArray()` method determines whether an object is an array. This function returns `true` if the object is an array, and `false` if not.

```
 // Creating some variables
  var v1 = {name: "John", age: 22};
  var v2 = ["red", "green", "blue", "yellow"];
  var v3 = [10, 20, 30, 40, 50];
  var v4 = null;

  // Testing the variables data type
  typeof(v1); // Returns: "object"
  typeof(v2); // Returns: "object"
  typeof(v3); // Returns: "object"
  typeof(v3); // Returns: "object"

  // Testing if the variable is an array
  Array.isArray(v1);  // Returns: false
  Array.isArray(v2);  // Returns: true
  Array.isArray(v3);  // Returns: true
  Array.isArray(v4);  // Returns: false
```

Note: The `Array.isArray()` method is supported in all major browsers, such as Chrome, Firefox, IE (9 and above)

Q 11.5. Can you give an example for destructuring an object?

Destructuring is an expression available in ES6 which enables a succinct and convenient way to extract values of Objects or Arrays and place them into distinct variables.

Example:

```
// Object Destructuring

let person = { name: "Sarah", country: "India", job: "Developer" };

let name = person.name;
let country = person.country;
let job = person.job;

console.log(name); // Sarah
console.log(country); // India
console.log(job); // Developer
```

🌟 **Try this example on CodeSandbox**

Q 11.6. How do you clone an object in JavaScript?

Using the object spread operator `...`, the object own enumerable properties can be copied into the new object. This creates a shallow clone of the object.

```
const obj = { a: 10, b: 20 }
const shallowClone = { ...obj }
```

With this technique, prototypes are ignored. In addition, nested objects are not cloned, but rather their references get copied, so nested objects still refer to the same objects as the original.

Example 01: Clone the Object Using Object.assign()

```
const person = {
    name: 'John',
    age: 21,
}

// cloning the object
const clonePerson = Object.assign({}, person);
```

```
console.log(clonePerson);

// changing the value of clonePerson
clonePerson.name = 'Peter';

console.log(clonePerson.name);
console.log(person.name);


// Output
{name: "John", age: 21}
Peter
John
```

Example 02: Clone the Object Using Spread Syntax

```
const person = {
    name: 'John',
    age: 21,
}

// cloning the object
const clonePerson = { ... person}

console.log(clonePerson);

// changing the value of clonePerson
clonePerson.name = 'Peter';

console.log(clonePerson.name);
console.log(person.name);


// Output
{name: "John", age: 21}
Peter
John
```

Example 03: Clone the Object Using JSON.parse()

```
const person = {
    name: 'John',
    age: 21,
}

// cloning the object
const clonePerson = JSON.parse(JSON.stringify(person));

console.log(clonePerson);

// changing the value of clonePerson
clonePerson.name = 'Peter';

console.log(clonePerson.name);
console.log(person.name);
```

```
// Output
{name: "John", age: 21}
Peter
John
```

🌟 [Try this example on CodeSandbox]

Q 11.7. How do you copy properties from one object to other?

You can use `Object.assign()` method which is used to copy the values and properties from one or more source objects to a target object. It returns the target object which has properties and values copied from the target object. The syntax would be as below,

```
Object.assign(target, ...sources)
```

Let us take example with one source and one target object,

```
const target = { a: 1, b: 2 };
const source = { b: 3, c: 4 };

const returnedTarget = Object.assign(target, source);

console.log(target); // { a: 1, b: 3, c: 5 }
console.log(returnedTarget); // { a: 1, b: 3, c: 5 }
```

As observed in the above code, there is a common property(`b`) from source to target so it is value is been overwritten.

Q 11.8. What is the difference between native, host and user objects?

1. Native Objects:

Are objects that are part of the JavaScript language defined by the ECMAScript specification. For example, `String`, `Math`, `RegExp`, `Object`, `Function` etc core objects defined in the ECMAScript spec.

2. Host Objects:

Are objects provided by the browser or runtime environment (Node). For example, `window`, `XmlHttpRequest`, `DOM nodes` etc considered as host objects.

3. User Objects:

Are objects defined in the javascript code. For example, User object created for profile information.

Q 11.9. What are the properties of Intl object?

The `Intl` object is the namespace for the ECMAScript Internationalization API that provides language number formatting, string comparison, and date/time formatting. Below are the list of properties available on Intl object,

1. Collator:

Intl.Collator provides a language-aware comparison of Strings for sorting and searching.

Example:

```
// Intl.Collator()

let collatorEs = new Intl.Collator("es").compare;
console.log(["a", "z", "ñ", "b"].sort(collatorEs)); // ["a", "b", "ñ", "z"]


let collatorEsCaseFirts = new Intl.Collator("es", { caseFirst: "upper" }).compare;
console.log(["a", "A", "z", "ñ", "b"].sort(collatorEsCaseFirts)); // ["A", "a",
"b", "ñ", "z"]
```

🌟 **Try this example on CodeSandbox**

2. DateTimeFormat:

These are the objects that enable language-sensitive date and time formatting.

Example:

```
// Intl.DateTimeFormat()

let now = new Date();
let nowEnUs = new Intl.DateTimeFormat("en-US");
let noeEs = new Intl.DateTimeFormat("es-ES");

console.log(nowEnUs.format(now)); // 5/17/2022
console.log(noeEs.format(now)); // 17/5/2022
```

🌟 **Try this example on CodeSandbox**

3. ListFormat:

These are the objects that enable language-sensitive list formatting.

Example:

```
// Intl.ListFormat()

let lfEn = new Intl.ListFormat("en", {
    localeMatcher: "lookup",
    type: "disjunction",
    style: "narrow"
});
console.log(lfEn.format(['Hannibal smith', 'Murdock' , 'Faceman', 'B.A."
Baracus']));
// Hannibal smith, Murdock, Faceman, or B.A." Baracus

let lfEs = new Intl.ListFormat("es", {
    localeMatcher: "lookup",
    type: "disjunction",
    style: "narrow"
});
console.log(lfEs.format(['Hannibal smith', 'Murdock' , 'Faceman', 'B.A."
Baracus']));
// Hannibal smith, Murdock, Faceman o B.A." Baracus
```

🌟 **Try this example on CodeSandbox**

4. NumberFormat:

Objects that enable language-sensitive number formatting.

```
// Intl.NumberFormat()

let myNumber = 1000000.999;
let nfEs = new Intl.NumberFormat('es-ES');
let nfEn = new Intl.NumberFormat('en-EU');

console.log(nfEs.format(myNumber)); //1.000.000,999

console.log(nfEn.format(myNumber)); // 1,000,000.999
```

🌟 **Try this example on CodeSandbox**

5. RelativeTimeFormat:

Objects that enable language-sensitive relative time formatting.

Example:

```
// Intl.RelativeTimeFormat()
```

```
let rtfEn = new Intl.RelativeTimeFormat('en', { numeric: 'auto'  });

console.log(rtfEn.format(2, 'day')); // in 2 days
console.log(rtfEn.format(-1, 'day')); // yesterday
console.log(rtfEn.format(-5, 'month')); //5 months ago


let rtfEs = new Intl.RelativeTimeFormat('es', { numeric: 'auto' });

console.log(rtfEs.format(2, 'day')); // pasado mañana
console.log(rtfEs.format(-1, 'day')); // ayer
console.log(rtfEs.format(-5, 'month')); // Hace 5 meses
```

✯ **Try this example on CodeSandbox**

6. Locale:

Intl.Locale has a toString method that represents the complete contents of the locale. This method allows Locale instances to be provided as an argument to existing Intl constructors.

Example:

```
// Example: Intl.Locale()

let newLocale = new Intl.Locale("en-US", { language: "es" });
console.log(newLocale.toString()); // es-US

let now = new Date();
let dtfMyNewLocale = new Intl.DateTimeFormat(newLocale);

console.log(dtfMyNewLocale.format(now)); // 17/5/2022


let newLocale2 = new Intl.Locale("en-US", { language: "en" });
console.log(newLocale2.toString()); //en-US

let now2 = new Date();
let dtfMyNewLocale2 = new Intl.DateTimeFormat(newLocale2);

console.log(dtfMyNewLocale2.format(now2)); // 5/17/2022
```

✯ **Try this example on CodeSandbox**

Q 11.10. How do you convert date to another timezone in javascript?

The `.toLocaleString()` method to convert date in one timezone to another.
For example, let us convert current date to British English timezone as below,

```
console.log(event.toLocaleString('en-GB', { timeZone: 'UTC' })); //29/06/2019,
09:56:00
```

Q 11.11. Explain the difference between mutable and immutable objects?

A mutable object is an object whose state can be modified after it is created. An immutable object is an object whose state cannot be modified after it is created.

In JavaScript numbers, strings, null, undefined and Booleans are primitive types which are immutable. Objects, arrays, functions, classes, maps, and sets are mutable.

Q 11.12. How to create immutable object in javascript

In JavaScript, some built-in types (numbers, strings) are immutable, but custom objects are generally mutable. Some built-in immutable JavaScript objects are `Math`, `Date`.
Here are a few ways to add/simulate immutability on plain JavaScript objects.

1. Object Constant Properties:

By combining `writable: false` and `configurable: false`, you can essentially create a constant (cannot be changed, redefined or deleted) as an object property, like:
Example:

```
let myObject = {};

Object.defineProperty(myObject, 'number', {
  value: 10,
  writable: false,
  configurable: false,
});

console.log(myObject.number); // 10
myObject.number = 20;
console.log(myObject.number); // 10
```

2. Prevent Extensions:

This method prevents the addition of new properties to our existing object. `preventExtensions()` is a irreversible operation. We can never add extra properties to the object again.

Example:

```
const myCar = {
  maxSpeed: 250,
  batteryLife: 300,
  weight: 123
};

Object.isExtensible(myCar); // true
Object.preventExtensions(myCar);

Object.isExtensible(myCar); // false
myCar.color = 'blue';
console.log(myCar.color) // undefined
```

3. Seal:

It prevents additions or deletion of properties. `seal()` also prevents the modification of property descriptors.

Example:

```
const myCar = {
  maxSpeed: 250,
  batteryLife: 300,
  weight: 123
};

Object.isSealed(myCar); // false
Object.seal(myCar);
Object.isSealed(myCar); // true

myCar.color = 'blue';
console.log(myCar.color); // undefined

delete myCar.batteryLife; // false
console.log(myCar.batteryLife); // 300

Object.defineProperty(myCar, 'batteryLife'); // TypeError: Cannot redefine
property: batteryLife
```

4. Freeze:

It does the same that `Object.seal()` plus it makes the properties non-writable.

Example:

```
const myCar = {
  maxSpeed: 250,
  batteryLife: 300,
```

```
  weight: 123
};

Object.isFrozen(myCar); // false
Object.freeze(myCar);
Object.isFrozen(myCar); // true

myCar.color = 'blue';
console.log(myCar.color); // undefined

delete myCar.batteryLife;
console.log(myCar.batteryLife); // 300

Object.defineProperty(myCar, 'batteryLife'); // TypeError: Cannot redefine
property: batteryLife

myCar.batteryLife = 400;
console.log(myCar.batteryLife); // 300
```

Q 11.13. How do you determine whether object is frozen or not?

`Object.isFrozen()` method is used to determine if an object is frozen or not. An object is frozen if all of the below conditions hold true,

1. If it is not extensible.
2. If all of its properties are non-configurable.
3. If all its data properties are non-writable. The usage is going to be as follows,

```
const object = {
  property: 'Welcome JS world'
};
Object.freeze(object);
console.log(Object.isFrozen(object));
```

Q 11.14. How can you achieve immutability in your own code?

For "mutating" objects, use the spread operator, `Object.assign`, `Array.concat()`, etc., to create new objects instead of mutate the original object.
Example:

```
// Array Example
const arr = [10, 20, 30];
const newArr = [...arr, 40, 50]; // [10, 20, 30, 40, 50]

// Object Example
const human = Object.freeze({ race: "human" });
const aditya = { ...human, name: "Aditya" }; // {race: "human", name: "Aditya"}
const alienAditya = { ...aditya, race: "alien" }; // {race: "alien", name:
"Aditya"}
```

⭐ **Try this example on CodeSandbox**

Q 11.15. What is the drawback of declaring methods directly in JavaScript objects?

One of the drawback of declaring methods directly in JavaScript objects is that they are very memory inefficient. When you do that, a new copy of the method is created for each instance of an object.

Example:

```
const Employee = function (name, company, salary)
{
  this.name = name || "";
  this.company = company || "";
  this.salary = salary || 5000;

  // We can create a method like this:
  this.formatSalary = function () {
    return "$ " + this.salary;
  };
};

// we can also create method in Employee's prototype:
Employee.prototype.formatSalary2 = function () {
  return "$ " + this.salary;
};

// Creating Objects
let emp1 = new Employee("Yuri Garagin", "Company 1", 1000);
let emp2 = new Employee("Dinesh Gupta", "Company 2", 2000);
```

Here, each instance variable `emp1`, `emp2` has own copy of `formatSalary` method. However the `formatSalary2` will only be added once to an object `Employee.prototype`.

⭐ **Try this example on CodeSandbox**

Q 11.16. How do you compare Object and Map?

1. Object:

A data structure in which data is stored as key value pairs. In an object the key has to be a number, string, or symbol. The value can be anything so also other objects, functions, etc. An object is a **nonordered** data structure, i.e. the sequence of insertion of key value pairs is not remembered.

Example:

```
// Object()

let obj = {};

// adding properties to a object
obj.prop = 10;
obj[2] = 20;

// getting nr of properties of the object
Object.keys(obj).length; // 2

// deleting a property
delete obj[2];

obj; // {prop: 10}
```

2. ES6 Map:

A data structure in which data is stored as key value pairs. In which **a unique key maps to a value**. Both the key and the value can be in **any data type**. A map is an iterable data structure. This means that the sequence of insertion is remembered and that we can access the elements in e.g. a `for..of` loop.

Example:

```
// Map()

const myMap = new Map();

const keyString = "a string",
  keyObj = {},
  keyFunc = function () {};

// setting the values
myMap.set(keyString, "value associated with 'a string'");
myMap.set(keyObj, "value associated with keyObj");
myMap.set(keyFunc, "value associated with keyFunc");
```

```
myMap.size; // 3

// getting the values
myMap.get(keyString); // "value associated with 'a string'"
myMap.get(keyObj); // "value associated with keyObj"
myMap.get(keyFunc); // "value associated with keyFunc"

myMap.get("a string"); // "value associated with 'a string'"
// because keyString === 'a string'
myMap.get({}); // undefined, because keyObj !== {}
myMap.get(function () {}); // undefined, because keyFunc !== function () {}
```

Key differences:

- A `Map` is ordered and iterable, whereas a objects is not ordered and not iterable
- We can put any type of data as a `Map` key, whereas objects can only have a number, string, or symbol as a key.
- A `Map` inherits from `Map.prototype`. This offers all sorts of utility functions and properties which makes working with `Map` objects a lot easier.

🌟 **Try this example on CodeSandbox**

Q 11.17. What is shallow copy and deep copy in javascript?

1. Shallow Copy:

Shallow copy is a bit-wise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the reference addresses are copied i.e., only the memory address is copied.

A Shallow copy of the object can be done using `object.assign()`

Example:

```
// Shallow Copy

let obj = {
  a: 10,
  b: 20,
};

let objCopy = Object.assign({}, obj);
```

```
console.log(objCopy); // Result - { a: 1, b: 2 }
```

2. Deep Copy:

A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

A Deep copy of the object can be done using `JSON.parse(JSON.stringify(object))`

Example:

```
// Deep Copy

let obj2 = {
  a: 10,
  b: {
    c: 20
  }
};

let newObj = JSON.parse(JSON.stringify(obj2));
obj2.b.c = 30;

console.log(obj2); // { a: 10, b: { c: 20 } }
console.log(newObj); // { a: 10, b: { c: 20 } }
```

☆ [Try this example on CodeSandbox]

Q 11.18. Write a function called deepClone which takes an object and creates a object copy of it?

```
var newObject = deepClone(obj);
```

Solution:

```
function deepClone(object) {
  var newObject = {};
  for (var key in object) {
    if (typeof object[key] === "object" && object[key] !== null) {
      newObject[key] = deepClone(object[key]);
    } else {
      newObject[key] = object[key];
    }
  }
```

```
  return newObject;
}
```

Explanation: We have been asked to do deep copy of object so What is basically It is mean ?. Let us understand in this way you have been given an object `personalDetail` this object contains some property which again a type of object here as you can see `address` is an object and `phoneNumber` in side an `address` is also an object. In simple term `personalDetail` is nested object(object inside object). So Here deep copy means we have to copy all the property of `personalDetail` object including nested object.

```
var personalDetail = {
    name : 'Alex',
    address : {
      location: 'xyz',
      zip : '123456',
      phoneNumber : {
        homePhone: 8797912345,
        workPhone : 1234509876
      }
    }
}
```

So when we do deep clone then we should copy every property (including the nested object).

Q 11.19. Write a function called `Clone` which takes an object and creates a object copy of it but not copy deep property of object?

```
  var objectLit = {foo : 'Bar'};
    var cloneObj = Clone(obj); // Clone is the function which you have to write
    console.log(cloneObj === Clone(objectLit)); // this should return false
    console.log(cloneObj == Clone(objectLit)); // this should return true
```

solution:

```
function Clone(object){
  var newObject = {};
  for(var key in object){
      newObject[key] = object[key];
  }
  return newObject;
}
```

Q 11.20. How do you check if a key exists in an object?

1. Using `in` operator:
You can use the in operator whether a key exists in an object or not

```
const obj = { key: undefined };

console.log("key" in obj); // true, regardless of the actual value
```

and If you want to check if a key doesn't exist, remember to use parenthesis,

```
const obj = { not_key: undefined };

console.log(!("key" in obj)); // true if "key" doesn't exist in object
```

2. Using `hasOwnProperty()` method:
You can use `hasOwnProperty` to particularly test for properties of the object instance (and not inherited properties)

```
const obj = { key: undefined };

console.log(obj.hasOwnProperty("key")); // true
```

🌟 **Try this example on CodeSandbox**

Q 11.21. How do you loop through or enumerate javascript object?

You can use the `for-in` loop to loop through javascript object. You can also make sure that the key you get is an actual property of an object, and doesn't come from the prototype using `hasOwnProperty` method.

```
var object = {
    "k1": "value1",
    "k2": "value2",
    "k3": "value3"
};

for (var key in object) {
    if (object.hasOwnProperty(key)) {
        console.log(key + " -> " + object[key]); // k1 -> value1 ...
    }
```

```
}
```

Q 11.22. How do you test for an empty object?

a.) Using Object keys(ECMA 5+): You can use object keys length along with constructor type.

```
Object.keys(obj).length === 0 && obj.constructor === Object
```

b.) Using Object entries(ECMA 7+): You can use object entries length along with constructor type.

```
Object.entries(obj).length === 0 && obj.constructor === Object
```

Q 11.23. What is a proxy object?

The `Proxy` object allows to create an object that can be used in place of the original object, but which may redefine fundamental `Object` operations like getting, setting, and defining properties.
Proxy objects are commonly used to log property accesses, validate, format, or sanitize inputs, and so on.

Syntax:

```
const proxy = new Proxy(target, handler)
```

In this syntax:

- **target**: is an object to wrap.
- **handler**: is an object that contains methods to control the behaviors of the `target`.

Example:

```
// define an object called user
const user = {
  firstName: "Aniket",
  lastName: "Narula",
  email: "aniket.narula@email.com"
```

```
};

// define a handler object:
const handler = {
  get(target, property) {
    console.log(`Property ${property} has been read.`);
    return target[property];
  }
};

// create a proxy object:
const proxyUser = new Proxy(user, handler);

console.log(proxyUser.firstName);
console.log(proxyUser.lastName);

// Output
Property firstName has been read.
Aniket
Property lastName has been read.
Narula

user.firstName = 'Sonam';
console.log(proxyUser.firstName);

// Output
Property firstName has been read.
Sonam
```

There are many real-world applications for Proxies

- Validation
- Value correction
- Property lookup extensions
- Tracing property accesses
- Revocable references
- Implementing the DOM in javascript

🌟 **Try this example on CodeSandbox**

Q 11.24. What is Reflection in JavaScript?

Reflection is defined as the ability of a program to inspect and modify its structure and behavior at runtime. `Reflect` is not a function object. `Reflect` helps with forwarding default operations from the handler to the target.
Example:

```
// Math.max()
let number = Reflect.apply(Math.max, Math, [10, 20, 30]);
console.log(number); // 30

// FromCharCode()
let string = Reflect.apply(String.fromCharCode, undefined, [ 104, 101, 108, 108,
111]); // "hello"
console.log(string); // "hello"

// RegExp()
let index = Reflect.apply(RegExp.prototype.exec, /o/, ["Hello"]).index;
console.log(index); // 4
```

🌟 **Try this example on CodeSandbox**

Q 11.25. How do you display the current date in javascript?

You can use `new Date()` to generate a new Date object containing the current date and time.
Example:

```
// Current Date

let today = new Date();
let dd = String(today.getDate()).padStart(2, '0');
let mm = String(today.getMonth() + 1).padStart(2, '0'); //January is 0!
let yyyy = today.getFullYear();

today = mm + '/' + dd + '/' + yyyy;
document.write(today);
```

🌟 **Try this example on CodeSandbox**

Q 11.26. How do you add a key value pair in javascript?

There are two possible solutions to add new properties to an object. Let us take a simple object to explain these solutions.

```
const object = {
    key1: value1,
```

```
    key2: value2
};
```

a.) Using dot notation: This solution is useful when you know the name of the property

```
object.key3 = "value3";
```

b.) Using square bracket notation: This solution is useful when the name of the property is dynamically determined.

```
obj["key3"] = "value3";
```

Q 11.27. How do you check whether an object can be extendable or not?

The `Object.isExtensible()` method is used to determine if an object is extensible or not. i.e, Whether it can have new properties added to it or not.

```
// Validate object is extendable or not

const person = {
  firstName: "Sima",
  lastName: "Chander",
  email: "sima.chander@email.com"
};
console.log(Object.isExtensible(person)); //true

Object.preventExtensions(person);
console.log(Object.isExtensible(person)); // false
```

Note: By default, all the objects are extendable. i.e, The new properties can added or modified.

🌟 **Try this example on CodeSandbox**

Q 11.28. How to compare two objects in javascript?

Objects are reference types so you can't just use `===` or `==` to compare 2 objects. One quick way to compare if 2 objects have the same key value, is using `JSON.stringify()`. Another way is using Lodash `.isEqual()` function.

Example:

```
const obj1 = { id: 100 };
const obj2 = { id: 100 };

// Using JavaScript
JSON.stringify(obj1) === JSON.stringify(obj2); // true

// Using Lodash
_.isEqual(obj1, obj2); // true
```

⭐ **Try this example on CodeSandbox**

Q 11.29. How do you get enumerable key and value pairs?

The `Object.entries()` method is used to return an array of a given object own enumerable string-keyed property [key, value] pairs, in the same order as that provided by a `for...in` loop. Let us see the functionality of object.entries() method in an example,

```
const object = {
  a: 'Good morning',
  b: 100
};

for (let [key, value] of Object.entries(object)) {
  console.log(`${key}: ${value}`); // a: 'Good morning'
                                   // b: 100
}
```

Note: The order is not guaranteed as object defined.

Q 11.30. What is the main difference between Object.values and Object.entries method?

The `Object.values()` method's behavior is similar to `Object.entries()` method but it returns an array of values instead [key,value] pairs.

```
const object = {
  a: 'Good morning',
  b: 100
};
```

```
for (let value of Object.values(object)) {
  console.log(`${value}`); // 'Good morning'
                                100
}
```

Q 11.31. How can you get the list of keys of any object?

You can use `Object.keys()` method which is used return an array of a given object's own property names, in the same order as we get with a normal loop. For example, you can get the keys of a user object,

```
const user = {
  name: 'John',
  gender: 'male',
  age: 40
};

console.log(Object.keys(user)); //['name', 'gender', 'age']
```

Q 11.32. What is difference between array[] vs Object()?

- `[]` is declaring an array.
- `{}` is declaring an object.

An array has all the features of an object with additional features (you can think of an array like a sub-class of an object) where additional methods and capabilities are added in the Array sub-class. In fact, typeof [] === "object" to further show you that an array is an object.

The additional features consist of a magic `.length` property that keeps track of the number of items in the array and a whole slew of methods for operating on the array such as `.push()`, `.pop()`, `.slice()`, `.splice()`, etc... You can see a list of array methods here.
An object gives you the ability to associate a property name with a value as in:

```
var x = {};
x.foo = 3;
x["whatever"] = 10;
```

```
console.log(x.foo);      // shows 3
console.log(x.whatever); // shows 10
```

Object properties can be accessed either via the `x.foo` syntax or via the array-like syntax `x["foo"]`. The advantage of the latter syntax is that you can use a variable as the property name like `x[myvar]` and using the latter syntax, you can use property names that contain characters that Javascript won't allow in the `x.foo` syntax.
An array is an object so it has all the same capabilities of an object plus a bunch of additional features for managing an **ordered**, **sequential** list of numbered indexes starting from `0` and going up to some length. Arrays are typically used for an ordered list of items that are accessed by numerical index. And, because the array is ordered, there are lots of useful features to manage the order of the list `.sort()` or to add or remove things from the list.

Q 11.33. What is difference between `{}` vs `new Object()`?

1. Object Literal Syntax (`{}`):
Object literal syntax is a shorthand way of creating an object. We can create an object by placing a comma-separated list of key-value pairs inside curly braces `{ }`. The key represents a property name of the object and the value represents the value of that property.
Syntax:

```
let myObj = {
  prop1: value1,
  prop2: value2,
  prop3: value3
};
```

Example:

```
let person = {
  name: "John",
  age: 30,
  city: "New York"
};
```

2. Object Constructor Syntax (`new Object()`):
The object constructor syntax is a way of creating an object using the `new` operator and the `Object` constructor function. We can create an empty object by calling the `Object` constructor without any arguments. We can also create an object by passing an object literal as an argument to the `Object` constructor.
Syntax:

```
let myObj = new Object();
```

Example:

```
let person = new Object();
person.name = "John";
person.age = 30;
person.city = "New York";
```

Difference:

The primary difference between using `{}` and `new Object()` to create an object is that the former uses object literal syntax while the latter uses object constructor syntax.
Object literals are more concise and easier to read and write, especially when creating objects with a small number of properties. On the other hand, object constructors are more flexible and can be used to create objects with properties that are not known at the time of object creation.

In general, it is recommended to use object literal syntax (`{}`) for creating objects unless there is a specific reason to use the object constructor syntax (`new Object()`).

12. WINDOW AND DOCUMENT OBJECT

Q 12.1. What is the difference between window and document object?

1. Window Object:

The window object is the topmost object of the DOM hierarchy. It represents a browser window or frame that displays the contents of the webpage. Whenever a window appears on the screen to display the contents of the document, the window object is created.

Syntax:

```
window.property_name;
```

Window Object Properties:

Property	Description
closed	Returns a boolean true if a window is closed.
console	Returns the Console Object for the window.
document	Returns the Document object for the window.
frames	Returns all window objects running in the window.
history	Returns the History object for the window.
innerHeight	Returns the height of the window's content area (viewport) including scrollbars
innerWidth	Returns the width of a window's content area (viewport) including scrollbars
localStorage	Allows to save key/value pairs in a web browser. Stores the data with no expiration date
location	Returns the Location object for the window.
navigator	Returns the Navigator object for the window.
opener	Returns a reference to the window that created the window
outerHeight	Returns the height of the browser window, including toolbars/scrollbars
outerWidth	Returns the width of the browser window, including toolbars/scrollbars

Property	Description
pageXOffset	Returns the pixels the current document has been scrolled (horizontally) from the upper left corner of the window
pageYOffset	Returns the pixels the current document has been scrolled (vertically) from the upper left corner of the window
parent	Returns the parent window of the current window
screen	Returns the Screen object for the window
screenLeft	Returns the horizontal coordinate of the window relative to the screen
screenTop	Returns the vertical coordinate of the window relative to the screen
screenX	Returns the horizontal coordinate of the window relative to the screen
screenY	Returns the vertical coordinate of the window relative to the screen
sessionStorage	Allows to save key/value pairs in a web browser. Stores the data for one session
scrollX	An alias of pageXOffset
scrollY	An alias of pageYOffset
self	Returns the current window
top	Returns the topmost browser window

2. Document Object:

The document object represent a web page that is loaded in the browser. By accessing the document object, we can access the element in the HTML page. The document object can be accessed with a `window.document` or just `document`.

Syntax:

```
document.property_name;
```

Document Object Properties:

Property	Description
addEventListener()	Attaches an event handler to the document
baseURI	Returns the absolute base URI of a document
body	Sets or returns the document's body (the `<body>` element)
characterSet	Returns the character encoding for the document
close()	Closes the output stream previously opened with document.open()
cookie	Returns all name/value pairs of cookies in the document
createAttribute()	Creates an attribute node
createElement()	Creates an Element node
createEvent()	Creates a new event
createTextNode()	Creates a Text node
defaultView	Returns the window object associated with a document, or null if none is available.
designMode	Controls whether the entire document should be editable or not.
doctype	Returns the Document Type Declaration associated with the document
documentElement	Returns the Document Element of the document (the element)

Property	Description
documentURI	Sets or returns the location of the document
forms	Returns a collection of all `<form>` elements in the document
getElementById()	Returns the element that has the ID attribute with the specified value
getElementsByClassName()	Returns an HTMLCollection containing all elements with the specified class name
getElementsByName()	Returns an live NodeList containing all elements with the specified name
getElementsByTagName()	Returns an HTMLCollection containing all elements with the specified tag name
images	Returns a collection of all `<img>` elements in the document
normalize()	Removes empty Text nodes, and joins adjacent nodes
open()	Opens an HTML output stream to collect output from document.write()
querySelector()	Returns the first element that matches a specified CSS selector(s) in the document
querySelectorAll()	Returns a static NodeList containing all elements that matches a specified CSS selector(s) in the document
readyState	Returns the (loading) status of the document
referrer	Returns the URL of the document that loaded the current document
removeEventListener()	Removes an event handler from the document (that has been attached with the addEventListener() method)
title	Sets or returns the title of the document
URL	Returns the full URL of the HTML document

Property	Description
write()	Writes HTML expressions or JavaScript code to a document
writeln()	Same as write(), but adds a newline character after each statement

Difference:

Window	Document
It is the root level element in any web page	It is the direct child of the window object. This is also known as Document Object Model(DOM)
By default window object is available implicitly in the page	You can access it via window.document or document.
It has methods like alert(), confirm() and properties like document, location	It provides methods like getElementById(), getElementByTagName(), createElement() etc

Q 12.2. How do you access history in javascript?

The `window.history` object allows you to access the history stack of the browser. To navigate to a URL in the history, you use the `back()`, `forward()`, and `go()` methods. The `history.length` returns the number of URLs in the history stack.

1. Move backward:

```
window.history.back();

// Or

history.back();
```

2. Move forward:

```
history.forward();
```

3. Move to a specific URL in the history:

To move to a specific URL in the history stack, you use the `go()` method.
The `go()` method accepts an integer that is the relative position to the current page.
The current page's position is `0`.

```
// to move backward a page
history.go(-1);

// To move forward a page
history.go(1)

// To refresh the current page
history.go(0);
history.go();
```

4. Check history stack:

```
history.length
```

Q 12.3. How do you find operating system details?

The `window.navigator` object contains information about the visitor's browser os details. Some of the OS properties are avaialble under platform property,
Example:

```
let OS = "Unknown";

if (navigator.userAgent.indexOf("Win") !== -1) OS = "Windows";
if (navigator.userAgent.indexOf("Mac") !== -1) OS = "MacOS";
if (navigator.userAgent.indexOf("X11") !== -1) OS = "UNIX";
if (navigator.userAgent.indexOf("Linux") !== -1) OS = "Linux";

console.log(OS);
console.log(navigator.userAgent);

// Output
Windows
VM87:8 Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/101.0.0.0 Safari/537.36
```

🌟 **Try this example on CodeSandbox**

Q 12.4. How do you detect a browser language preference?

You can use navigator object to detect a browser language preference as below,

```
var language = navigator.languages && navigator.languages[0] || // Chrome / Firefox
               navigator.language ||   // All browsers
               navigator.userLanguage; // IE <= 10

console.log(language);
```

Q 12.5. What is BOM?

The Browser Object Model (BOM) allows JavaScript to "talk to" the browser. It consists of the objects navigator, history, screen, location and document which are children of window. The Browser Object Model is not standardized and can change based on different browsers.

Q 12.6. How do you redirect new page in javascript?

To redirect to a new URL or page, you assign the new URL to the `location.href` property or use the `location.assign()` method.
The `location.replace()` method does redirect to a new URL but does not create an entry in the history stack of the browser.
Example:

```
function redirect() {
  window.location.href = 'newPage.html';
}
```

☆ **Try this example on CodeSandbox**

Q 12.7. How to get the current url with javascript?

The `window.location` object can be used to get the current page address (URL) and to redirect the browser to a new page. You can also use `document.URL` for **read-only** purpose.
Location Properties:

Properties	Description
href	returns the href (URL) of the current page
hostname	returns the domain name of the web host
pathname	returns the path and filename of the current page
protocol	returns the web protocol used (http: or https:)
assign()	loads a new document
host	The hostname and port of the URL
port	The port number in the URL
search	The query portion of the URL
hash	The anchor portion of the URL

Example:

```
console.log('location.href', window.location.href); // Returns full URL
```

☆ [Try this example on CodeSandbox]

Q 12.8. How to get query string values in javascript?

The `URLSearchParams()` provides an interface to work with query string parameters. The `has()` method of the `URLSearchParams()` determines if a parameter with a specified name exists.

Example: URLSearchParams()

```
const urlParams = new URLSearchParams(window.location.search);
const clientCode = urlParams.get('clientCode');
```

The URLSearchParams() methods:

- **keys()** returns an iterator that iterates over the parameter keys.
- **values()** returns an iterator that iterates over the parameter values.
- **entries()** returns an iterator that iterates over the (key, value) pairs of the parameters.

Example: keys()

```
const urlParams = new URLSearchParams('?type=list&page=20');

for (const key of urlParams.keys()) {
    console.log(key);
}

// Output
type
page
```

Example: values()

```
const urlParams = new URLSearchParams('?type=list&page=20');

for (const value of urlParams.values()) {
    console.log(value);
}

// Output
list
20
```

Example: entries()

```
const urlParams = new URLSearchParams('?type=list&page=20');

for (const entry of urlParams.entries()) {
    console.log(entry);
}

// Output
["type", "list"]
```

```
["page", "20"]
```

⭐ [Try this example on CodeSandbox]

Q 12.9. What is difference between window.frames, window.parent and window.top in JavaScript?

- **window.frames** – the collection of "children" windows (for nested frames).
- **window.parent** – property returns the immediate parent of the current window
- **window.top** – returns the topmost window in the hierarchy of window objects

Q 12.10. What are the properties used to get size of window?

1. The screen size:

The screen size is the width and height of the screen: a monitor or a mobile screen.

Example:

```
const screenWidth  = window.screen.width;
const screenHeight = window.screen.height;
```

2. The available screen size:

The available screen size consists of the width and height of the active screen without the Operating System toolbars.

Example:

```
const availScreenWidth  = window.screen.availWidth;
```

```
const availScreenHeight = window.screen.availHeight;
```

3. The window outer size:

The window outer size consists of the width and height of the entire browser window, including the address bar, tabs bar, and other browser panels.

Example:

```
const windowOuterWidth  = window.outerWidth;
const windowOuterHeight = window.outerHeight;
```

4. The window inner size:

The window inner size (aka viewport size) consists of the width and height of the viewport that displays the web page.

Example:

```
const windowInnerWidth  = window.innerWidth;
const windowInnerHeight = window.innerHeight;
```

5. The web page size:

The web page size consists of the width and height of the page content rendered.

Example:

```
const pageWidth  = document.documentElement.scrollWidth;
const pageHeight = document.documentElement.scrollHeight;
```

Q 12.11 What are the ways to execute javascript after page load?

You can execute javascript after page load in many different ways,

1. window.onload:

```
window.onload = function ...
```

2. document.onload:

```
document.onload = function ...
```

3. body onload:

```
<body onload="script();">
```

4. Defer the script:

```
<script src="deferMe.js" defer></script>
```

Q 12.12. What is the difference between document load event and document domcontentloaded event?

1. DOMContentLoaded:

The `DOMContentLoaded` event is fired when the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading. DOMContentLoaded is a great event to use to hookup UI functionality to complex web pages.

Example:

```
document.addEventListener("DOMContentLoaded", function(e) {
  console.log("DOM loaded!");
});
```

2. loadEvent:

The load event fires when all files have finished loading from all resources, including ads and images.

Example:

```
document.addEventListener("load", function(e) {
  console.log("Page loaded!");
```

```
});
```

Q 12.13. What do you understand by Screen objects?

- **window**: is the execution context and global object for that context's JavaScript
- **document**: contains the DOM, initialized by parsing HTML
- **screen**: The screen object contains information about the visitor's screen.

Property	Description
availHeight	Returns the height of the screen (excluding the Windows Taskbar)
availWidth	Returns the width of the screen (excluding the Windows Taskbar)
colorDepth	Returns the bit depth of the color palette for displaying images
height	Returns the total height of the screen
pixelDepth	Returns the color resolution (in bits per pixel) of the screen
width	Returns the total width of the screen

🌟 **Try this example on CodeSandbox**

Q 12.14. How to change style of html element using javascript?

Below is the syntax for manipulating the style property on an HTML element using JavaScript:

Syntax:

```
HTMLElement.style.property = "new style"
```

1. Using Style Property:

```
// Example 01:
document.getElementById("title").style.fontSize = "30px";

// Example 02:
document.getElementById("message").style = "color:#f00;padding:5px;"
```

2. Using ClassName Property:

```
// Example 01:
document.getElementById("title").style.className = "custom-title";

// Example 02:
const x = document.getElementsByClassName("message");

for (i = 0; i < x.length; i++) {
  x[i].style = "padding:20px;border:1px solid #bbb;";
}
```

Q 12.15. How do you print the contents of web page?

The window object provided print() method which is used to prints the contents of the current window. It opens Print dialog box which lets you choose between various printing options.

Example:

```
  <input type="button" value="Print" onclick="window.print()" />
```

Note: In most browsers, it will block while the print dialog is open.

Q 12.16. How do I modify the url without reloading the page?

The `window.localtion.url` property will be helpful to modify the url but it reloads the page. HTML5 introduced
the `history.pushState()` and `history.replaceState()` methods, which allow you to add and modify history entries, respectively.

Example:

```
window.history.pushState('newPage', 'Title', '/newPage.html');
```

Q 12.17. When would you use `document.write()`?

The **document.write()** method is used to delete all the content from the HTML document and inserts the new content. It is also used to give the additional text to an output which is open by the `document.open()` method.
The `document.write()` only works while the page is loading; If you call it after the page is done loading, it will overwrite the whole page. This method is mostly used for testing purpose.
Example:

```
document.write("Hello World!");
```

Q 12.18. What is the difference between an attribute and a property?

Attributes are defined on the HTML markup whereas properties are defined on the DOM. For example, the below HTML element has 2 attributes type and value,

```
<input type="text" value="Name:">
```

You can retrieve the attribute value as below,

```
const input = document.querySelector('input');
console.log(input.getAttribute('value')); // Good morning
console.log(input.value); // Good morning
```

And after you change the value of the text field to "Good evening", it becomes like

```
console.log(input.getAttribute('value')); // Good morning
console.log(input.value); // Good evening
```

Q 12.19. What is the difference between firstChild and firstElementChild?

1. firstChild:

The firstChild property returns the first child node of the specified node, as a Node object.

```
<ul id="myList">
    <li>Coffee</li>
    <li>Tea</li>
</ul>
let list = document.getElementById("myList").firstChild.innerHTML; // Coffee
```

2. firstElementChild:

The firstElementChild property returns the first child element of the specified element.

```
<ul id="myList">
  <li>Coffee</li>
  <li>Tea</li>
</ul>
let list = document.getElementById("myList").firstElementChild.innerHTML; //
Coffee
```

Difference:

firstChild returns the first child **node** (an element node, a text node or a comment node). Whitespace between elements are also text nodes.

firstElementChild returns the first child **element** (not text and comment nodes).

Q 12.20. What is difference between document.getElementById() and document.querySelector()?

1. document.getElementById():

Returns an element object representing the element whose id property matches the specified string. Since element IDs are required to be unique if specified, they're a useful way to get access to a specific element quickly.

```
element = document.getElementById(id);
```

2. document.querySelector():

Returns the first matching Element node within the node's subtree. If no matching node is found, null is returned.

```
element = document.querySelector(selectors);
```

3. document.querySelectorAll():

Returns a NodeList containing all matching Element nodes within the node's subtree, or an empty NodeList if no matches are found.

```
element = document.querySelectorAll(selectors);
```

Note: `querySelector()` *is more useful when we want to use more complex selectors.*

Q 12.21. Name the two functions that are used to create an HTML element dynamically?

In an HTML document, the `document.createElement()` method creates the HTML element specified by tagName.
Syntax:

```
const element = document.createElement(tagName[, options]);
```

HTML

```
<!DOCTYPE html>
<html>
  <head>
      <title>||Working with elements||</title>
  </head>
<body>
  <div id="app">The text above has been created dynamically.</div>
</body>
</html>
```

JavaScript

```
document.body.onload = addElement;
```

```
function addElement () {
  // create a new div element
  var newDiv = document.createElement("div");

  var newContent = document.createTextNode("Hi there and greetings!");
  // add the text node to the newly created div
  newDiv.appendChild(newContent);

  // add the newly created element and its content into the DOM
  var currentDiv = document.getElementById("app");
  document.body.insertBefore(newDiv, currentDiv);
}
```

Create Dynamic Button:

```
// Create a button

let btn = document.createElement("BUTTON");
btn.innerHTML = "CLICK ME";
document.body.appendChild(btn);
```

Removing Elements Dynamically:

```
// Removes an element from the document

function removeElement(elementId) {
  let element = document.getElementById(elementId);
  element.parentNode.removeChild(element);
}
```

⭐ **Try this example on CodeSandbox**

Q 12.22. What is difference between append() vs appendChild()?

- ParentNode.append() allows you to also append DOMString object, whereas Node.appendChild() only accepts Node objects.
- ParentNode.append() has no return value, whereas Node.appendChild() returns the appended Node object.
- ParentNode.append() can append several nodes and strings, whereas Node.appendChild() can only append one node.

The main difference is that `appendChild()` is a DOM function meanwhile `append()` is a JavaScript function.

```
document.getElementById("yourId").append("Hello");
```

```
var p = document.createElement("p");
document.getElementById("yourId").appendChild(p);
```

Q 12.23. How to check if page is fully loaded using javascript?

```
if (document.readyState === 'complete') {
  // The page is fully loaded
}
```

Q 12.24. What is a web-storage event and its event handler?

The StorageEvent is an event that fires when a storage area has been changed in the context of another document. Whereas onstorage property is an EventHandler for processing storage events. The syntax would be as below

```
  window.onstorage = functionRef;
```

Let us take the example usage of onstorage event handler which logs the storage key and its values

```
if (typeof(Storage) !== "undefined") {
  window.onstorage = function(e) {
    console.log('The ' + e.key +
      ' key has been changed from ' + e.oldValue +
      ' to ' + e.newValue + '.');
  };
} else {
  // Browser doesnot support web-storage
}
```

Q 12.25. How to detect browser type in javascript?

To detect user browser information use the `navigator.userAgent()` property.

```
let browser;
```

```
const agt = navigator.userAgent.toLowerCase();

if (agt.indexOf("chrome") > -1) {
  browser = "Google Chrome";
} else if (agt.indexOf("safari") > -1) {
  browser = "Apple Safari";
} else if (agt.indexOf("opera") > -1) {
  browser = "Opera";
} else if (agt.indexOf("firefox") > -1) {
  browser = "Mozilla Firefox";
} else if (agt.indexOf("mise") > -1 || agt.indexOf("trident") > -1) {
  browser = "Microsoft Internet Explorer";
}

alert("You are using: " + browser + " \n\nNavigator: " + agt);
```

☆ **Try this example on CodeSandbox**

13. CLASSES

Q 13.1. Explain how prototypal inheritance works?

The Prototypal Inheritance is a feature in javascript used to add methods and properties in objects. It is a method by which an object can inherit the properties and methods of another object.

In order to get and set the [[Prototype]] of an object, we use `Object.getPrototypeOf()` and `Object.setPrototypeOf()`. Nowadays, in modern language, it is being set using `__proto__`.

Syntax:

```
ChildObject.__proto__ = ParentObject
```

Example:

In the given example, there are two objects **ParentUser** and **ChildUser**. The object ChildUser inherits the methods and properties of the object ParentUser and further uses them.

```
// Parent Object
let ParentUser = {
```

```
  talk: true,
  Canfly() {
    return "Sorry, Can't fly";
  },
};

// Child Object
let ChildUser = {
  CanCode: true,
  CanCook() {
    return "Can't say";
  },

  //  Inheriting the properties and methods of Parent Object
  __proto__: ParentUser,
};

// Property of Parent Object
console.log("Can a User talk?: " + ChildUser.talk);

// Method of ParentUser
console.log("Can a User fly?: " + ChildUser.Canfly());

// Property of ChildUser
console.log("Can a User code?: " + ChildUser.CanCode);

// Method of ChildUser
console.log("Can a User cook?: " + ChildUser.CanCook());
```

☆ [Try this example on CodeSandbox]

Q 13.2. What is the difference between prototype and `proto` in JavaScript?

1. Proto:

It is an actual object that provides a way inherit to inherit properties from JavaScript with the help of an object which is created with new. Every object with behavior associated has internal property [[prototype]].

Syntax:

```
Object.__proto__ = value
```

Example:

```
function Employee(id, name) {
  this.id = id;
```

```
  this.name = name;
}
const employee = new Employee(1090, "Sarvesh Ghose");

// Object have proto property
employee

// Also if apply strict equal to check
// if both point at the same
// location then it will return true.
Employee.prototype === employee._proto_ // false
```

🌟 **Try this example on CodeSandbox**

2. Prototype:

It is a special object which means it holds shared attributes and behaviors of instances. It is a way to inherit properties from javascript as it is available in every function declaration.

Syntax:

```
objectTypeName.prototype.SharedPropertyName = value;
```

Example:

```
// Constructor function
function Employee(id, name) {
  this.id = id;
  this.name = name;
}

// Objects
const employee = new Employee(3325, "Karishma Som");

// Prototype
Employee.prototype.getName = function () {
  return this.name;
};

// Function call using object
console.log(employee.getName());
```

🌟 **Try this example on CodeSandbox**

Q 13.3. What are the differences between ES6 class and ES5 function constructors?

Classes are a template for creating objects. They encapsulate data with code to work on that data. Classes in JS are built on prototypes but also have some syntax and semantics that are not shared with ES5 class-like semantics.

ES6 Classes formalize the common JavaScript pattern of simulating class-like inheritance hierarchies using functions and prototypes. They are effectively simple sugaring over prototype-based OO, offering a convenient declarative form for class patterns which encourage interoperability.

ES6 Class Properties

- Class keyword
- getter/setter method
- constructor function
- extends keyword
- super keyword
- static keyword

Example: ES5 Function Constructor

```
// ES5 Function Constructor
function Student(name, studentId) {
  // Call constructor of superclass to initialize superclass-derived members.
  Person.call(this, name);

  // Initialize subclass's own members.
  this.studentId = studentId;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;
```

Example: ES6 Class

```
// ES6 Class
class Student extends Person {
  constructor(name, studentId) {
    super(name);
    this.studentId = studentId;
  }
}
```

It's much more verbose to use inheritance in ES5 and the ES6 version is easier to understand and remember.

Q 13.4. What is class expression in es6 class?

A class expression is another way to define a class. Class expressions can be named or unnamed. The name given to a named class expression is local to the class's body. However, it can be accessed via the name property.

Example:

```
// Unnamed Class
let Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
console.log(Rectangle.name); // Rectangle

// Named Class
let Triangle = class TriangleClass {
  constructor(base, height) {
    this.base = base;
    this.height = height;
  }
};
console.log(Triangle.name); // TriangleClass
```

🌟 [Try this example on CodeSandbox]

Q 13.5. What is difference between private, public and static variables?

Private variables can be accessed by all the members (functions and variables) of the owner object but not by any other object. Public variables can be accessed by all the members of the owner as well as other objects that can access the owner. Static variables are related to a class. They come into existence as soon as a class come into existence.

Example:

```
// Constructor Function
function MyClass () {

  var privateVariable = "I am private!";  // Private variable
  this.publicVariable = "I am public!";  // Public variable

  this.publicMethod = function () {  // Public Method
```

```
    return privateVariable;
  };
}

// Instance method will be available to all instances but only load once in memory
MyClass.prototype.publicMethod = function () {
  return this.publicVariable;
};

// Static variable shared by all instances
MyClass.staticProperty = "I am static!";

var myInstance = new MyClass();

console.log(myClass.publicMethod()); // I am private!
console.log(MyClass.staticProperty); // I am static!
```

🌟 **Try this example on CodeSandbox**

Q 13.6. What is difference between Classic Inheritance and Prototypical Inheritance?

1. Class Inheritance:

Instances inherit from classes (like a blueprint — a description of the class), and create sub-class relationships: hierarchical class taxonomies. Instances are typically instantiated via constructor functions with the new keyword. Class inheritance may or may not use the class keyword from ES6.

2. Prototypal Inheritance:

Instances inherit directly from other objects. Instances are typically instantiated via factory functions or Object.create(). Instances may be composed from many different objects, allowing for easy selective inheritance.

Features

- Classes: create tight coupling or hierarchies/taxonomies.
- Prototypes: mentions of concatenative inheritance, prototype delegation, functional inheritance, object composition.
- No preference for prototypal inheritance & composition over class inheritance.

The difference between classical inheritance and prototypal inheritance is that classical inheritance is limited to classes inheriting from other classes while

prototypal inheritance supports the cloning of any object using an object linking mechanism. A prototype basically acts as a template for other objects, whether they are extending the base object or not.

Example:

```
function Circle(radius) {
  this.radius = radius;
}

Circle.prototype.area = function () {
  let radius = this.radius;
  return Math.PI * radius * radius;
};

Circle.prototype.circumference = function () {
  return 2 * Math.PI * this.radius;
};

const circle = new Circle(5);

console.log(circle.area()); // 78.53981633974483
console.log(circle.circumference()); // 31.41592653589793
```

🌟 **Try this example on CodeSandbox**

Q 13.7. How do you create an object with prototype?

The `Object.create()` method is used to create a new object with the specified prototype object and properties. i.e, It uses existing object as the prototype of the newly created object. It returns a new object with the specified prototype object and properties.

Example:

```
const user = {
  name: "Jayesh Sahni",
  printInfo: function () {
    console.log(`My name is ${this.name}.`);
  }
};

const admin = Object.create(user);
admin.name = "Disha Choudhry"; // Here, "name" is a property set on "admin" but
not on "user" object
admin.printInfo(); // My name is Disha Choudhry
```

🌟 **Try this example on CodeSandbox**

Q 13.8. How to use constructor functions for inheritance in JavaScript?

Let say we have `Person` class which has name, age, salary properties and **incrementSalary()** method.

```
// Functions Constructor
function Person(name, age, salary) {
  this.name = name;
  this.age = age;
  this.salary = salary;
  this.incrementSalary = function (byValue) {
    this.salary = this.salary + byValue;
  };
}
```

Now we wish to create Employee class which contains all the properties of Person class and wanted to add some additional properties into Employee class.

```
function Employee(company){
    this.company = company;
}

// Prototypal Inheritance
Employee.prototype = new Person("Sundar Pichai", 24, 5000);
```

In the example above, **Employee** type inherits from **Person**. It does so by assigning a new instance of `Person` to `Employee` prototype. After that, every instance of `Employee` inherits its properties and methods from `Person`.

```
// Prototypal Inheritance
Employee.prototype = new Person("Sundar Pichai", 24, 5000);

var employee = new Employee("Google");

console.log(employee instanceof Person); // true
console.log(employee instanceof Employee); // true
```

☆ [Try this example on CodeSandbox]

Q 13.9. What is prototype chain?

Prototype chaining is used to build new types of objects based on existing ones. It is similar to inheritance in a class based language. The prototype on object instance is available through `Object.getPrototypeOf(object)` or `__proto__` property whereas prototype on constructors function is available through **Object.prototype**.
Example:

```
function Person(firstName, lastName, age) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
}
// Prototype chaining
Person.prototype.getFullName = function () {
  return this.firstName + " " + this.lastName;
};

// create an instance of the Person class
const person = new Person("Vanya", "Dayal", 25);

person.hasOwnProperty("firstName"); // true
person.hasOwnProperty("getFullName"); // false
person.getFullName(); // Vanya Dayal
```

🌟 **Try this example on CodeSandbox**

Q 13.10. What are javascript accessors?

ECMAScript 5 introduced javascript object accessors or computed properties through getters and setters. Getters uses `get` keyword whereas Setters uses `set` keyword.

```
var user = {
  firstName: "John",
  lastName : "Abraham",
  language : "en",
  get lang() {
    return this.language;
  }
  set lang(lang) {
  this.language = lang;
  }
};

console.log(user.lang); // getter access lang as en
user.lang = 'fr';
console.log(user.lang); // setter used to set lang as fr
```

Q 13.11. How do you define property on Object constructor?

The Object.defineProperty() static method is used to define a new property directly on an object, or modifies an existing property on an object, and returns the object.

```
const newObject = {};

Object.defineProperty(newObject, 'newProperty', {
  value: 100,
  writable: false
});

console.log(newObject.newProperty); // 100

newObject.newProperty = 200; // It throws an error in strict mode due to writable
setting
```

Q 13.12. What is the difference between get and defineProperty?

Both has similar results until unless you use classes. If you use `get` the property will be defined on the prototype of the object whereas
using `Object.defineProperty()` the property will be defined on the instance it is applied to.

Q 13.13. What are the advantages of Getters and Setters?

Below are the list of benefits of Getters and Setters,

- They provide simpler syntax
- They are used for defining computed properties, or accessors in JS.
- Useful to provide equivalence relation between properties and methods
- They can provide better data quality
- Useful for doing things behind the scenes with the encapsulated logic.

Q 13.14. Can I add getters and setters using defineProperty method?

Yes, You can use `Object.defineProperty()` method to add Getters and Setters. For example, the below counter object uses increment, decrement, add and substract properties,

```
var counterObj = {counter : 0};

// Define getters
Object.defineProperty(obj, "increment", {
  get : function () {this.counter++;}
});
Object.defineProperty(obj, "decrement", {
  get : function () {this.counter--;}
});

// Define setters
Object.defineProperty(obj, "add", {
  set : function (value) {this.counter += value;}
});
Object.defineProperty(obj, "subtract", {
  set : function (value) {this.counter -= value;}
});

obj.add = 10;
obj.subtract = 5;
console.log(obj.increment); //6
console.log(obj.decrement); //5
```

Q 13.15. What is a decorator?

A decorator is an expression that evaluates to a function and that takes the target, name, and decorator descriptor as arguments. Also, it optionally returns a decorator descriptor to install on the target object.

Let us define admin decorator for user class at design time,

```
function admin(isAdmin) {
  return function(target) {
      target.isAdmin = isAdmin;
  }
}

@admin(true)
class User() {
}
console.log(User.isAdmin); // true
```

```
@admin(false)
class User() {
}
console.log(User.isAdmin); // false
```

14. ERROR HANDLING

Q 14.1. What is an error object?

An error object is a built in error object that provides error information when an error occurs. It has two properties: **name** and **message**.

Example:

```
try {
  greeting("Welcome");
}
catch(err) {
  console.log(err.name + ": " + err.message);
}

// Output
ReferenceError: greeting is not defined
```

☆ [Try this example on CodeSandbox]

Q 14.2. Define the various types of errors which occur in JavaScript?

There are three main types of errors that can occur while compiling a JavaScript program: **syntax errors**, **runtime errors** (also called **exceptions**), and **logical errors**. When an exception occurs, an object representing the error is created and thrown. The JavaScript language defines seven types of built-in error objects.

1. Error:

The "Error" type is used to represent generic exceptions. This type of exception is most often used for implementing user defined exceptions.

```
const error = new Error("error message");
```

2. RangeError:

"RangeError" exceptions are generated by numbers that fall outside of a specified range.

```
const pi = 3.14159;

pi.toFixed(100000);  // RangeError: toFixed() digits argument must be between 0
and 100
```

3. ReferenceError:

A "ReferenceError" exception is thrown when a non-existent variable is accessed. These exceptions commonly occur when an existing variable name is misspelled.

```
function sum() {
  val++;  // ReferenceError: val is not defined
}
```

4. SyntaxError:

A "SyntaxError" is thrown when the rules of the JavaScript language are broken.

```
if (foo) {  // SyntaxError
  // the closing curly brace is missing
```

5. TypeError:

A "TypeError" exception occurs when a value is not of the expected type. Attempting to call a non-existent object method is a common cause of this type of exception.

```
const foo = {};

foo.bar(); // TypeError
```

6. URIError:

A "URIError" exception is thrown by methods such as encodeURI() and decodeURI() when they encounter a malformed URI. The following example generates a "URIError" while attempting to decode the string "%".

```
decodeURIComponent("%"); // URIError
```

7. EvalError:

"EvalError" exceptions are thrown when the eval() function is used improperly. These exceptions are not used in the most recent version of the EcmaScript standard. However, they are still supported in order to maintain backwards compatibility with older versions of the standard.

Q 14.3. What are the various statements in error handling?

Below are the list of statements used in an error handling,

1. **try:** This statement is used to test a block of code for errors
2. **catch:** This statement is used to handle the error
3. **throw:** This statement is used to create custom errors.
4. **finally:** This statement is used to execute code after try and catch regardless of the result.

Example:

```
function errorHandling() {

  const message = document.getElementById("app");
  message.innerHTML = "";
  let x = document.getElementById("app").value;

  try {
    if (x === "") throw "is empty";
    if (isNaN(x)) throw "is not a number";
    x = Number(x);
    if (x > 10) throw "is too high";
    if (x < 5) throw "is too low";
  } catch (err) {
    message.innerHTML = "Error: " + err + ".";
  } finally {
    document.getElementById("app").value = "";
  }
}

errorHandling(); // Error: is not a number.
```

☆ **Try this example on CodeSandbox**

15. PROMISES

Q 15.1. What is a promise?

A promise is an object that may produce a single value some time in the future with either a resolved value or a reason that it's not resolved (for example, network error). It will be in one of the 3 possible states: **fulfilled**, **rejected**, or **pending**.

Syntax:

```
const promise = new Promise(function (resolve, reject) {
  // promise description
})
```

Example:

```
let promise = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1 second signal that the job is done with the result "done"
  setTimeout(() => resolve("done"), 1000);
});
```

Promises are used to handle asynchronous operations. They provide an alternative approach for callbacks by reducing the callback hell and writing the cleaner code.

Promises have three states:

1. **pending:** initial state, neither fulfilled nor rejected.
2. **fulfilled:** meaning that the operation was completed successfully.
3. **rejected:** meaning that the operation failed.

Q 15.2. What is promise chaining?

The process of executing a sequence of asynchronous tasks one after another using promises is known as Promise chaining. It allows you to chain on another then call

which will run when the second promise is fulfilled. The `.catch()` can still be called to handle any errors that might occur along the way.
Example:

```
// Promise Chain
new Promise(function (resolve, reject) {
  setTimeout(() => resolve(10), 1000);
})
  .then(function (result) {
    console.log(result); // 10
    return result + 20;
  })
  .then(function (result) {
    console.log(result); // 30
    return result + 30;
  });
```

In the above handlers, the result is passed to the chain of .then() handlers with the below work flow,

- The initial promise resolves in one second,
- After that `.then()` handler is called by logging the result(10) and then return a promise with the value of `result + 20`.
- After that the value passed to the next `.then()` handler by logging the result(20) and return a promise with `result + 30`.

🌟 **Try this example on CodeSandbox**

Q 15.3. What is promise.all()?

`Promise.all` is a promise that takes an array of promises as an input (an iterable), and it gets resolved when all the promises get resolved or any one of them gets rejected.
Example:

```
// promise.all()
const promise1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 10, "First");
});

const promise2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 20, "Second");
});

Promise.all([promise1, promise2])
```

```
  .then((values) => {
    console.log(values);
  })
  .catch((error) => console.log(`Error in promises ${error}`));
// expected output: Array ["First", "Second"]
```

Note: Remember that the order of the promises (output the result) is maintained as per input order.

⭐ **Try this example on CodeSandbox**

Q 15.4. What is the purpose of race method in promise?

`Promise.race()` method will return the promise instance which is firstly resolved or rejected.
Example: Let us take an example of `race()` method where promise2 is resolved first

```
const promise1 = new Promise(function (resolve, reject) {
  setTimeout(resolve, 500, "First");
});

const promise2 = new Promise(function (resolve, reject) {
  setTimeout(resolve, 100, "Second");
});

Promise.race([promise1, promise2]).then(function (value) {
  console.log(value); // "Second" // Both promises will resolve, but promise2 is
faster
});
```

⭐ **Try this example on CodeSandbox**

Q 15.5. What are the pros and cons of promises over callbacks?

Below are the list of pros and cons of promises over callbacks,

Pros:

- It avoids callback hell which is unreadable
- Easy to write sequential asynchronous code with `.then()`

- Easy to write parallel asynchronous code with `Promise.all()`
- Solves some of the common problems of callbacks(call the callback too late, too early, many times and swallow errors/exceptions)
- Integrated error handling.

Cons:

- It makes little complex code
- It cannot return multiple arguments.
- We need to load a polyfill if ES6 is not supported

Q 15.6. How does await and async works in es6?

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.

Async keyword is used along with the function declaration which specifies that this function is now able to accept all types of asynchronous events on itself. **Await** basically waits for the results which are particularly to be fetched from the source from which that async function is about to fetch the data.

Example:

```
// async() and await()
async function fetchMethod() {
  try {
    let response = await fetch("https://api.github.com/users/1");
    let data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}

fetchMethod();
```

🌟 **Try this example on CodeSandbox**

Q 15.7. What is difference between fetch() and XMLHttpRequest() in JavaScript?

1. XMLHttpRequest:

`XMLHttpRequest()` is a built-in browser object that allows to make HTTP requests in JavaScript. XMLHttpRequest has two modes of operation: **synchronous** and **asynchronous**.
Example:

```
const xhr = new XMLHttpRequest();

xhr.onreadystatechange = function () {
  if (this.readyState === 4 && this.status === 200) {
    console.log(this.responseText);
  }
};

xhr.open("GET", "https://jsonplaceholder.typicode.com/todos/1", true); // this
makes asynchronous true or false
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xhr.send();
```

✯ **Try this example on CodeSandbox**

2. Fetch():

Fetch allows to make network requests similar to `XMLHttpRequest`. Fetch makes it easier to make asynchronous requests and handle responses better than with the older XMLHttpRequest. It is an improvement over the `XMLHttpRequest` API. The main difference between `Fetch()` and `XMLHttpRequest()` is that the Fetch API uses Promises, hence avoiding **callback hell**.
Example:

```
fetch("https://jsonplaceholder.typicode.com/todos/1")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.log("Something went wrong!", err);
  });
```

✯ **Try this example on CodeSandbox**

Q 15.8. Explain fetch() properties in JavaScript?

A `fetch()` function is available in the global window object. The fetch() function takes one mandatory argument, the path to the resource you want to fetch. It returns a Promise, whether it is successful or not. If request is successful `.then()` function will receive Response object, if request fails then `.catch()` function will receive an error object

Example:

```
fetch("https://api.github.com/users/learning-zone")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.log("Something went wrong!", err);
  });
```

🌟 **Try this example on CodeSandbox**

Headers Object

The Headers interface allows to create own headers object via the `Headers()` constructor. A headers object is a collection of name-value pairs.

Example:

```
let reqHeader = new Headers();
reqHeader.append("Content-Type", "text/json");
let initObject = {
  method: "GET",
  headers: reqHeader
};

fetch("https://api.github.com/users/learning-zone", initObject)
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.log("Something went wrong!", err);
  });
```

🌟 **Try this example on CodeSandbox**

Request Object

The Request Object represents a resource request. Instead of passing an URL of the resource into the `fetch()` call, you can create a request object using

the `Request()` constructor, and pass that as an argument to `fetch()`. By passing Request object to the `fetch()`, you can make customised requests.

Example:

```
let reqHeader = new Headers();
reqHeader.append("Content-Type", "text/json");

let initObject = {
  method: "GET",
  headers: reqHeader
};

const userRequest = new Request(
  "https://api.github.com/users/learning-zone",
  initObject
);

fetch(userRequest)
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.log("Something went wrong!", err);
  });
```

🌟 **Try this example on CodeSandbox**

Q 15.9. What is the difference between Promise and AJAX?

A Promise is **an interface** for asynchronous operations. They keep track of when asynchronous operations complete and what their results are and let you coordinate that completion and those results (including error conditions) with other code or other asynchronous operations. They aren't actually asynchronous operations in themselves.

An Ajax call is a specific asynchronous operation that can be used with with a traditional callback interface or wrapped in a promise interface. We can make an Ajax call either with a traditional callback using the `XMLHttpRequest()` interface or we can make an Ajax call (in modern browsers), using a promise with the `fetch()` interface.

Q 15.10. What is difference between async or defer keyword in JavaScript?

1. async Attribute

The async attribute is used to indicate to the browser that the script file can be executed asynchronously. The HTML parser does not need to pause at the point it reaches the script tag to fetch and execute, the execution can happen whenever the script becomes ready after being fetched in parallel with the document parsing.

```
<script src="script.js" async></script>
```

This attribute is only available for externally located script files. When an external script has this attribute, the file can be downloaded while the HTML document is still parsing. Once it has been downloaded, the parsing is paused for the script to be executed.

2. defer Attribute

The defer attribute tells the browser not to wait for the script. Instead, the browser will continue to process the HTML, build DOM. The script loads "in the background", and then runs when the DOM is fully built.

```
<script src="script.js" defer></script>
```

Like an asynchronously loaded script, the file can be downloaded while the HTML document is still parsing. However, even if the file is fully downloaded long before the document is finished parsing, the script is not executed until the parsing is complete.

Q 15.11. What is request header in javascript?

The `headers` read-only property of the `Request` interface contains the `Headers` object associated with the request.
Syntax

```
const myHeaders = request.headers;
```

Example:

```
const myHeaders = new Headers();
```

```
myHeaders.append('Content-Type', 'image/jpeg');

var myInit = {
  method: 'GET',
  headers: myHeaders,
  mode: 'cors',
  cache: 'default'
};

const myRequest = new Request('flowers.jpg', myInit);
myContentType = myRequest.headers.get('Content-Type'); // returns 'image/jpeg'
```

Q 15.12. Explain ajax request in javascript?

Ajax stands for Asynchronous Javascript And Xml. It load data from the server and selectively updating parts of a web page without reloading the whole page.

Basically, Ajax uses browser's built-in `XMLHttpRequest()` object to send and receive information to and from a web server asynchronously, in the background, without blocking the page or interfering with the user's experience.

Example:

```
(function() {
      var xhr;
      document.getElementById('ajaxButton').addEventListener('click',
makeRequest);

      function makeRequest() {
        if (window.XMLHttpRequest) {
          // code for IE7+, Firefox, Chrome, Opera, Safari
          xhr = new XMLHttpRequest();
        } else {
          // code for IE6, IE5
          xhr = new ActiveXObject('Microsoft.XMLHTTP');
        }
        xhr.onreadystatechange = function() {
          if (this.readyState == 4 && this.status == 200) {
            document.getElementById("result").innerHTML = '<pre>' +
this.responseText + '</pre>';
          }
        };
        xhr.open('GET', 'https://jsonplaceholder.typicode.com/posts', true);
//this makes asynchronous true or false
        xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
        xhr.send();
      }
    })();
```

⭐ [Try this example on CodeSandbox]

Q 15.13. What is XMLHTTPRequest Object?

The `XMLHTTPRequest()` object is an API which is used for fetching data from the server. An object of XMLHTTPRequest is used for asynchronous communication between client and server. It retrieve any type of data such as json, xml, text etc.
XMLHttpRequest Object Methods:

Method	Description
new XMLHttpRequest()	Creates a new XMLHttpRequest object
abort()	Cancels the current request
getAllResponseHeaders()	Returns header information
getResponseHeader()	Returns specific header information
open(method,url,async,user,psw)	Specifies the request method: the request type GET or POST url: the file location async: true (asynchronous) or false (synchronous) user: optional user name psw: optional password
send()	Sends the request to the server Used for GET requests
send(string)	Sends the request to the server. Used for POST requests
setRequestHeader()	Adds a label/value pair to the header to be sent

XMLHttpRequest Object Properties:

Property	Description
onreadystatechange	Defines a function to be called when the readyState property changes
readyState	Holds the status of the XMLHttpRequest. 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
responseText	Returns the response data as a string
responseXML	Returns the response data as XML data
status	Returns the status-number of a request 200: "OK" 403: "Forbidden" 404: "Not Found"
statusText	Returns the status-text (e.g. "OK" or "Not Found")

Q. 15.14. How to get responses of multiple api calls, when some API fails?

Promise.allSettled():

To handle failures in multiple API calls, `Promise.allSettled()` method can be used. The method returns a Promise that resolves after all of the given promises have either resolved or rejected. It returns an array of objects with two properties: status and value.
The status property is either "fulfilled" or "rejected", depending on whether the promise was resolved or rejected. The value property is the value of the promise if it was resolved or the reason for rejection if it was rejected.

Example:

```
const promises = [fetch('api1'), fetch('api2'), fetch('api3')];

Promise.allSettled(promises)
  .then(results => {
    const successfulResults = [];
    const failedResults = [];

    results.forEach(result => {
      if (result.status === 'fulfilled') {
        successfulResults.push(result.value);
      } else {
        failedResults.push(result.reason);
      }
    });

    console.log('Successful API calls:', successfulResults);
    console.log('Failed API calls:', failedResults);
  })
  .catch(error => console.log('Error:', error));
```

Q. 15.15. Explain the use of Promise.any()?

The `Promise.any()` method is used to handle multiple promises simultaneously, and it resolves with the value of the first fulfilled promise.regardless of whether any of the other promises reject. If all of the promises reject, then `Promise.any()` rejects with an `AggregateError` object that contains an array of rejection reasons.
The `Promise.any()` method takes an iterable of Promises as an input, and returns a new Promise. Here's an example:

```
const promise1 = new Promise((resolve, reject) => setTimeout(reject, 1000,
'Promise 1 rejected'));
const promise2 = new Promise((resolve, reject) => setTimeout(resolve, 500,
'Promise 2 resolved'));
const promise3 = new Promise((resolve, reject) => setTimeout(resolve, 1500,
'Promise 3 resolved'));

Promise.any([promise1, promise2, promise3])
  .then((value) => console.log(value)) // 'Promise 2 resolved'
  .catch((error) => console.error(error)); // AggregateError: All promises were
rejected
```

Note: `Promise.any()` is not yet widely supported by all browsers, so you may need to use a polyfill or a different approach to achieve similar functionality in older browsers.

16. Collections

Q 16.1. What is the difference between ES6 Map and WeakMap?

Map:

It is used to associate a key to a value irrespective of the datatype such as strings, numbers, objects etc. To assign values to a map you need to use the set method:

```
const map = new Map();

map.set("a", 10);
map.set("b", 20);
map.set(3, 30);

console.log(map.get("a")); // 10

map.set("a", 50);

console.log(map.get("a")); // 50
console.log(map.size); // 3

map.delete("b");

console.log(map.size); // 2
console.log(map); // {'a' => 50, 3 => 30}
```

🌟 **Try this example on CodeSandbox**

WeakMap:

The WeakMap object is a collection of key/value pairs in which the keys are weakly referenced. In this case, keys must be objects and the values can be arbitrary values. WeakMap accepts only objects but not any primitive values (strings, numbers)

```
// WeakMap()

function Obj() {
  this.val = new Array(10).join("---");
}

window.obj = new Obj();
var map = new WeakMap();
console.log(window.obj); // {val: "-----------------", constructor: Object}
map.set(window.obj, 20); // WeakMap {Obj => 20}
console.log(map);
```

🌟 **Try this example on CodeSandbox**

Difference between Map and WeakMap:

1. A WeakMap accepts only **objects** as keys whereas a Map, in addition to **objects**, accepts primitive datatype such as **strings**, **numbers** etc.
2. WeakMap objects doesn't avert garbage collection if there are no references to the object which is acting like a key. Therefore there is no method to retrieve keys in WeakMap, whereas in Map there are methods such as `Map.prototype.keys()` to get the keys.
3. There is no size property exists in WeakMap.

Browser support for Map and WeakMap:

The latest Chrome, Firefox, Edge and Safari support Map and WeakMap on desktop. It's supported only in IE11 but not IE10 and below. On mobile, newer browsers also have support, but IE Mobile doesn't.

☆ **Try this example on CodeSandbox**

Q 16.2 What is the difference between Set and WeakSet in ES6?

Set:

Using the `Set()` class we can create an array like heterogeneous iterable object, which will contain only unique values in it. Unique is not just unique by values but also by types. i.e. it will consider `"2"` and `2` separate or different.
Syntax:

```
const mySet = new Set([iterable]);
```

Example::

```
const set = new Set([10, 20]);

set.add(30); // 10, 20, 30
set.add(30); // 10, 20, 30
set.add("Hello"); // 10, 20, 30, 'Hello'
set.add({ a: 10, b: 20 }); // 10, 20, 30, 'Hello', {a:10, b:20}

set.add(function () {}); // 10, 20, 30, 'Hello', {a:10, b:20}, [Function]

// Iterating Sets
for (let item of set) console.log(item);
```

```
console.log(set.has("Hello")); // ture
set.delete("Hello"); // 'Hello' deleted
console.log(set.has("Hello")); // false

console.log(set.size); // 5
set.clear(); // Set Cleared
```

⭐ **Try this example on CodeSandbox**

WeakSet:

A `WeakSet()` is a collection similar to Set, which holds unique values; but it only holds Objects and nothing else. If an object which is there in your WeakSet object has no other reference variable left, it will automatically be deleted.

Syntax:

```
const myWeakSet = new WeakSet([iterable with only objects]);
```

Example::

```
const weakSet = new WeakSet([{ a: 10 }]);
const obj1 = { o: 10 };
const obj2 = { o: 20 };

weakSet.add(obj1);
weakSet.add(obj2);

weakSet.has(obj2); // true
delete obj2; // Don't take it literally - you can't delete objects like that. Use
scope to execute this.

weakSet.has(obj2); // false, because you deleted obj2, so WeakSet releases it
automatically
weakSet.delete(obj1); // obj1 deleted from the set
weakSet.add(2); // ERROR, no primitive value
```

⭐ **Try this example on CodeSandbox**

Set	WeakSet
Can contain any type of values	Can only contain objects
To find number of elements use `.size()`	To find elements count use `.length()`

Set	WeakSet
.forEach() is available to iterate	No .forEach() to iterate
Nothing is auto destroyed	If an element object has no other reference left, it will be auto released to garbage collector

Q 16.3. List down the collection of methods available on WeakSet?

Below are the list of methods available on WeakSet,

- **add(value)**: A new object is appended with the given value to the weakset
- **delete(value)**: Deletes the value from the WeakSet collection.
- **has(value)**: It returns true if the value is present in the WeakSet Collection, otherwise it returns false.
- **length()**: It returns the length of weakSetObject

Let us see the functionality of all the above methods in an example,

```
const weakSetObject = new WeakSet();
const firstObject = {};
const secondObject = {};

// add(value)
weakSetObject.add(firstObject);
weakSetObject.add(secondObject);

console.log(weakSetObject.has(firstObject)); // true
weakSetObject.delete(secondObject);
```

☆ **Try this example on CodeSandbox**

Q 16.4. List down the collection of methods available on WeakMap?

Below are the list of methods available on WeakMap,

- set(key, value): Sets the value for the key in the WeakMap object. Returns the WeakMap object.
- delete(key): Removes any value associated to the key.
- has(key): Returns a Boolean asserting whether a value has been associated to the key in the WeakMap object or not.
- get(key): Returns the value associated to the key, or undefined if there is none.

Let us see the functionality of all the above methods in an example,

```
const weakMapObject = new WeakMap();
const firstObject = {};
const secondObject = {};

// set(key, value)
weakMapObject.set(firstObject, 'John');
weakMapObject.set(secondObject, 100);

console.log(weakMapObject.has(firstObject)); //true
console.log(weakMapObject.get(firstObject)); // John

weakMapObject.delete(secondObject);
```

Q 16.5. What is an Iterator?

An iterator is an object which defines a sequence and a return value upon its termination. It implements the Iterator protocol with a `.next()` method which returns an object with two properties:

- **value**: The next value in the iteration sequence.
- **done**: This is true if the last value in the sequence has already been consumed.

Example:

```
// custom Iterator
function numbers() {
  let n = 0;
  return {
    next: function () {
      n += 10;
      return { value: n, done: false };
    }
  };
}

// Create an Iterator
const number = numbers();
```

```
console.log(number.next()); // {value: 10, done: false}
console.log(number.next()); // {value: 20, done: false}
console.log(number.next()); // {value: 30, done: false}
```

🌟 **Try this example on CodeSandbox**

17. MODULES

Q 17.1. What is modules in ES6?

Making objects, functions, classes or variables available to the outside world is as simple as exporting them, and then importing them where needed in other files.

Benefits

- Code can be split into smaller files of self-contained functionality.
- The same modules can be shared across any number of applications.
- Ideally, modules need never be examined by another developer, because they've has been proven to work.
- Code referencing a module understands it's a dependency. If the module file is changed or moved, the problem is immediately obvious.
- Module code (usually) helps eradicate naming conflicts. Function `x()` in module1 cannot clash with function x() in module2. Options such as namespacing are employed so calls become `module1.x()` and `module2.x()`.

Exporting:

```
export const myNumbers = [1, 2, 3, 4];
const animals = ['Panda', 'Bear', 'Eagle']; // Not available directly outside the
module

export function myLogger() {
  console.log(myNumbers, animals);
}

export class Alligator {
   constructor() {
     // ...
   }
```

```
}
```

Exporting with alias:

```
export { myNumbers, myLogger as Logger, Alligator }
```

Default export:

```
export const myNumbers = [1, 2, 3, 4];
const animals = ['Panda', 'Bear', 'Eagle'];

export default function myLogger() {
  console.log(myNumbers, pets);
}

export class Alligator {
  constructor() {
    // ...
  }
}
```

18. MISCELLANEOUS

Q 18.1. Describe the Revealing Module Pattern in javascript?

Revealing module pattern is a design pattern, which let you organise your javascript code in modules, and gives better code structure. It gives you power to create public/private variables/methods (using closure), and avoids polluting global scope

It uses IIFE (Immediately invoked function expression: (function(){})();) to wrap your module function, thus creating a local scope for all your variables and methods.

Syntax:

```
const returnedValue = (function() { ... })();
```

Example:

```
const myModule = (function () {
```

```
  "use strict";

  var _privateProperty = "I am a private property";
  var publicProperty = "I am a public property";

  function _privateMethod() {
    console.log(_privateProperty);
  }

  function publicMethod() {
    _privateMethod();
  }

  return {
    publicMethod: publicMethod,
    publicProperty: publicProperty
  };
})();

myModule.publicMethod(); // outputs 'I am a private property'
console.log(myModule.publicProperty); // outputs 'I am a public property'
console.log(myModule._privateProperty); // is undefined protected by the module
closure
myModule._privateMethod(); // TypeError: protected by the module closure
```

☆ **Try this example on CodeSandbox**

Q 18.2. How do you detect javascript disabled in the page?

You can use `<noscript>` tag to detect javascript disabled or not. The code block inside `<noscript>` get executed when JavaScript is disabled, and are typically used to display alternative content when the page generated in JavaScript.

```
<script type="javascript">
    // JS related code goes here
</script>
<noscript>
    <a href="next_page.html?noJS=true">JavaScript is disabled in the apge. Please
click Next Page</a>
</noscript>
```

Q 18.3. What is the difference between feature detection, feature inference, and using the UA string?

1. Feature Detection

Feature detection involves working out whether a browser supports a certain block of code, and running different code depending on whether it does (or doesn't), so that the browser can always provide a working experience rather crashing/erroring in some browsers. For example:

```
if ('geolocation' in navigator) {
  // Can use navigator.geolocation
} else {
  // Handle lack of feature
}
```

Modernizr is a great library to handle feature detection.

2. Feature Inference

Feature inference checks for a feature just like feature detection, but uses another function because it assumes it will also exist, e.g.:

```
if (document.getElementsByTagName) {
  element = document.getElementById(id);
}
```

This is not really recommended. Feature detection is more foolproof.

3. UA String

This is a browser-reported string that allows the network protocol peers to identify the application type, operating system, software vendor or software version of the requesting software user agent. It can be accessed via `navigator.userAgent`. However, the string is tricky to parse and can be spoofed. For example, Chrome reports both as Chrome and Safari. So to detect Safari you have to check for the Safari string and the absence of the Chrome string. Avoid this method.

Q 18.4. What is strict mode?

The Strict Mode is allows you to place a program, or a function, in a `strict` operating context. This strict context prevents certain actions from being taken and throws more exceptions.
Example:

```
(function(){
"use strict";
```

```
// Define your library strictly...
})();
```

Advantages:

- Makes it impossible to accidentally create global variables.
- Makes assignments which would otherwise silently fail to throw an exception.
- Makes attempts to delete undeletable properties throw (where before the attempt would simply have no effect).
- Requires that function parameter names be unique.
- `this` is undefined in the global context.
- It catches some common coding bloopers, throwing exceptions.
- It disables features that are confusing or poorly thought out.

Q 18.5. Describe Singleton Pattern In JavaScript?

The **singleton pattern** is a type of creational pattern that restricts the instantiation of a class to a **single** object. This allows the class to create an instance of the class the first time it is instantiated; however, on the next try, the existing instance of the class is returned. No new instance is created.

Example:

```
/**
 * Singleton Pattern
 **/

let instance = null;

class Printer {

  constructor(pages) {
    this.display = function () {
      console.log(
        `You are connected to the printer. You want to print ${pages} pages.`
      );
    };
  }

  static getInstance(numOfpages) {
    if (!instance) {
      instance = new Printer(numOfpages);
```

```
    }
    return instance;
  }
}

var obj1 = Printer.getInstance(2);
console.log(obj1);
obj1.display();

var obj2 = Printer.getInstance(3);
console.log(obj2);
obj2.display();

console.log(obj2 === obj1); // true
```

🌟 **Try this example on CodeSandbox**

Q 18.6. What do you understand by ViewState and SessionState?

1. Session State:

Contains information that is pertaining to a specific session (by a particular client/browser/machine) with the server. It is a way to track what the user is doing on the site.. across multiple pages...amid the statelessness of the Web. e.g. the contents of a particular user's shopping cart is session data. Cookies can be used for session state.

- Maintained at session level.
- Session state value availability is in all pages available in a user session.
- Information in session state stored in the server.
- In session state, user data remains in the server. The availability of the data is guaranteed until either the user closes the session or the browser is closed.
- Session state is used for the persistence of user-specific data on the server's end.

2. View State:

On the other hand is information specific to particular web page. It is stored in a hidden field so that it is not visible to the user.

- Maintained at page level only.
- View state can only be visible from a single page and not multiple pages.
- Information stored on the client's end only.

- View state will retain values in the event of a postback operation occurring.
- View state is used to allow the persistence of page-instance-specific data.

Q 18.7. Explain browser console logs features?

The `Console` method **log()** outputs a message to the web console. The message may be a single string or it may be any one or more JavaScript objects.
1. console.table()

```
const artists = [
  {
    first: 'René',
    last: 'Magritte'
  },
  {
    first: 'Chaim',
    last: 'Soutine'
  },
  {
    first: 'Henri',
    last: 'Matisse'
  }
];
console.table(artists);
```

Output

2. console.log()

3. console.warn()

4. console.error()

Q 18.8. What are the difference between console.log() and console.dir()?

- `console.log()` prints the element in an HTML-like tree Output

- `console.dir()` prints the element in a JSON-like tree Output

Q 18.9. How to Copy Text to Clipboard?

```
<!-- The text field -->
<input type="text" id="inputText" value="Hello World">

<!-- The button used to copy the text -->
<button onclick="copy()">Copy text</button>
function copy() {
  /* Get the text field */
  let copyText = document.getElementById("inputText");

  /* Select the text field */
  copyText.select();
  copyText.setSelectionRange(0, 99999); /*For mobile devices*/

  /* Copy the text inside the text field */
  document.execCommand("copy");

  /* Alert the copied text */
  alert("Copied the text: " + copyText.value);
}
```

Q 18.10. What is a service worker?

A Service worker is basically a JavaScript file that runs in background, separate from a web page and provide features that don't need a web page or user interaction.

Some of the major features of service workers are

- Offline first web application development
- Periodic background syncs, push notifications
- Intercept and handle network requests
- Programmatically managing a cache of responses

Lifecycle of a Service Worker

It consists of the following phases:

- Download
- Installation
- Activation

Registering a Service Worker

To register a service worker we first check if the browser supports it and then register it.

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/ServiceWorker.js')
  .then(function(response) {

    // Service worker registration done
    console.log('Registration Successful', response);
  }, function(error) {
    // Service worker registration failed
    console.log('Registration Failed', error);
  }
```

Installation of service worker:

After the controlled page that takes care of the registration process, we come to the service worker script that handles the installation part.

Basically, you will need to define a callback for the install event and then decide on the files that you wish to cache. Inside a callback, one needs to take of the following three points –

- Open a cache
- Cache the files
- Seek confirmation for the required caches and whether they have been successful.

```
var CACHENAME = 'My site cache';
var urlstocache = [
    '/',
  '/styles/main1.css',
    '/script/main1.js'
];
self.addEventListener('install', function(event) {
    // Performing installation steps
    event.waitUntil(
        caches.open(CACHENAME)
```

```
        .then(function(cache) {
            console.log('Opening of cache');
            return cache.addAll(urlstocache);
        })
);
```

Cache and return requests:

After a service worker is installed and the user navigates to a different page or refreshes, the service worker will begin to receive fetch events, an example of which is below.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request)
      .then(function(response) {
        // Cache hit - return response
        if (response) {
          return response;
        }
        return fetch(event.request);
      }
    )
  );
});
```

Q 18.11. How do you manipulate DOM using service worker?

Service worker can't access the DOM directly. But it can communicate with the pages it controls by responding to messages sent via the `postMessage` interface, and those pages can manipulate the DOM.
Example: service-worker.html

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>Service Worker</title>
</head>
<body>
(Look in the console.)
<script>
(function() {
    "use strict";

    if (!navigator.serviceWorker || !navigator.serviceWorker.register) {
        console.log("This browser doesn't support service workers");
```

```
        return;
    }

    // Listen to messages from service workers.
    navigator.serviceWorker.addEventListener('message', function(event) {
        console.log("Got reply from service worker: " + event.data);
    });

    // Are we being controlled?
    if (navigator.serviceWorker.controller) {
        // Yes, send our controller a message.
        console.log("Sending 'hi' to controller");
        navigator.serviceWorker.controller.postMessage("hi");
    } else {
        // No, register a service worker to control pages like us.
        // Note that it won't control this instance of this page, it only takes
effect
        // for pages in its scope loaded *after* It is installed.
        navigator.serviceWorker.register("service-worker.js")
            .then(function(registration) {
                console.log("Service worker registered, scope: " +
registration.scope);
                console.log("Refresh the page to talk to it.");
                // If we want to, we might do `location.reload();` so that we'd be
controlled by it
            })
            .catch(function(error) {
                console.log("Service worker registration failed: " +
error.message);
            });
    }
})();
</script>
</body>
</html>
```

service-worker.js

```
self.addEventListener("message", function(event) {
    //event.source.postMessage("Responding to " + event.data);
    self.clients.matchAll().then(all => all.forEach(client => {
        client.postMessage("Responding to " + event.data);
    }));
});
```

Q 18.12. How to use Web Workers in javascript?

Step 01: Create a Web Workers file: Write a script to increment the count value.

```
// counter.js
let i = 0;
```

```
function timedCount() {
  i = i + 1;
  postMessage(i);
  setTimeout("timedCount()",500);
}

timedCount();
```

Here `postMessage()` method is used to post a message back to the HTML page.
Step 02: Create a Web Worker Object: Create a web worker object by checking for browser support.

```
if (typeof(w) == "undefined") {
  w = new Worker("counter.js");
}
```

and we can receive messages from web workers

```
w.onmessage = function(event){
  document.getElementById("message").innerHTML = event.data;
};
```

Step 03: Terminate a Web Workers: Web workers will continue to listen for messages (even after the external script is finished) until it is terminated. You can use terminate() method to terminate listening the messages.

```
w.terminate();
```

Step 04: Reuse the Web Workers: If you set the worker variable to undefined you can reuse the code

```
w = undefined;
```

Example:

```
<!DOCTYPE html>
<html>
<body>
  <p>Count numbers: <output id="result"></output></p>
  <button onclick="startWorker()">Start</button>
  <button onclick="stopWorker()">Stop</button>

  <script>
    var w;

    function startWorker() {
      if (typeof(Worker) !== "undefined") {
        if (typeof(w) == "undefined") {
          w = new Worker("counter.js");
```

```
        }
        w.onmessage = function(event) {
          document.getElementById("result").innerHTML = event.data;
        };
      } else {
        document.getElementById("result").innerHTML = "Sorry! No Web Worker
support.";
      }
    }

    function stopWorker() {
      w.terminate();
      w = undefined;
    }
  </script>
</body>
</html>
```

Q 18.13. What are the restrictions of web workers on DOM?

WebWorkers do not have access to below javascript objects since they are defined in an external files

1. Window object
2. Document object
3. Parent object

Q 18.14. What is rendering in JavaScript?

JavaScript-powered content needs to be rendered before it can output meaningful code and be displayed for the client. These are the different steps involved in the JavaScript rendering process:

1. **JavaScript**: Typically JavaScript is used to handle work that will result in visual changes.
2. **Style calculations**: This is the process of figuring out which CSS rules apply to which elements. They are applied and the final styles for each element are calculated.

3. **Layout**: Once the browser knows which rules apply to an element it can begin to calculate how much space it takes up and where it is on screen.
4. **Paint**: Painting is the process of filling in pixels. It involves drawing out text, colors, images, borders, and shadows, essentially every visual part of the elements.
5. **Compositing**: Since the parts of the page were drawn into potentially multiple layers they need to be drawn to the screen in the correct order so that the page renders correctly.

The main responsibility of the rendering engine is to display the requested page on the browser screen. Rendering engines can display HTML and XML documents and images.

Similar to the JavaScript engines, different browsers use different rendering engines as well. These are some of the popular ones:

- **Gecko** — Firefox
- **WebKit** — Safari
- **Blink** — Chrome, Opera (from version 15 onwards)

The process of rendering

The rendering engine receives the contents of the requested document from the networking layer.

Constructing the DOM tree

The first step of the rendering engine is parsing the HTML document and converting the parsed elements to actual DOM nodes in a DOM tree.

```
<html>
  <head>
    <meta charset="UTF-8">
    <link rel="stylesheet" type="text/css" href="theme.css">
  </head>
  <body>
    <p> Hello, <span> World! </span> </p>
    <div>
      <img src="smiley.gif" alt="Smiley face" height="42" width="42">
    </div>
  </body>
</html>
```

The DOM tree for this HTML will look like this:

Basically, each element is represented as the parent node to all of the elements, which are directly contained inside of it. And this is applied recursively.

Constructing the CSSOM tree

CSSOM refers to the CSS Object Model. While the browser was constructing the DOM of the page, it encountered a link tag in the head section which was referencing the external theme.css CSS style sheet. Anticipating that it might need that resource to render the page, it immediately dispatched a request for it. Let's imagine that the theme.css file has the following contents:

```
body {
  font-size: 16px;
}

p {
  font-weight: bold;
}

span {
  color: red;
}

p span {
  display: none;
}

img {
  float: right;
}
```

As with the HTML, the engine needs to convert the CSS into something that the browser can work with — the CSSOM. Here is how the CSSOM tree will look like:

When computing the final set of styles for any object on the page, the browser starts with the most general rule applicable to that node (for example, if it is a child of a body element, then all body styles apply) and then recursively refines the computed styles by applying more specific rules.

Painting the render tree

In this stage, the renderer tree is traversed and the renderer's paint() method is called to display the content on the screen. Painting can be global or incremental (similar to layout):

- **Global** — the entire tree gets repainted.

- **Incremental** — only some of the renderers change in a way that does not affect the entire tree. The renderer invalidates its rectangle on the screen. This causes the OS to see it as a region that needs repainting and to generate a paint event. The OS does it in a smart way by merging several regions into one.

Q 18.15. What is the difference between HTMLCollection and NodeList?

HTMLCollection

An HTMLCollection is a list of nodes. An individual node may be accessed by either ordinal index or the node's name or id attributes. Collections in the HTML DOM are assumed to be live meaning that they are automatically updated when the underlying document is changed.

NodeList

A NodeList object is a collection of nodes. The NodeList interface provides the abstraction of an ordered collection of nodes, without defining or constraining how this collection is implemented. NodeList objects in the DOM are live or static based on the interface used to retrieve them

Difference

An HTMLCollection is a list of webpage elements (div, p, ul, li, img objects, etc...) which form part of the DOM, and are of a specific node type usually referred to as an element node. A NodeList is also a list of nodes, but it can contain a list not only of element nodes, but other types of nodes as well. So it is a more generic list of nodes than HTMLCollection. HTMLCollection tells you that what it contains are webpage elements, specifically.

Both interfaces are collections of DOM nodes. They differ in the methods they provide and in the type of nodes they can contain. While a NodeList can contain any node type, an HTMLCollection is supposed to only contain Element nodes. An HTMLCollection provides the same methods as a NodeList and additionally a method called namedItem.

Collections are always used when access has to be provided to multiple nodes, e.g. most selector methods (such as getElementsByTagName) return multiple nodes or getting a reference to all children (element.childNodes).

Attribute Node

Refers to the attributes of an element node.

```
// html: <div id="my-id" />
let element = document.getElementById("my-id");
let myIdAttribute = element.getAttributeNode("id");
console.log(myIdAttribute); // output: my-id
```

Text Node

Refers to the text of an element.

```
// html: <div id="my-id"></div>
let element = document.getElementById("my-id");
let text = document.createTextNode("Some Text");
element.appendChild(text);
// updated html: <div id="my-id">Some Text</div>
```

Comment Node

```
<!-- This is what a comment node looks like -->
```

Q 18.16. What is a trampoline function? What is it used for?

The trampoline is just a technique to optimize **recursion** and prevent `stack-overflow` exceptions in languages that don't support tail call optimization like Javascript ES5 implementation. However, ES6 will probably have support for tail call optimization.
The problem with regular recursion is that every recursive call adds a stack frame to the call stack, which you can visualize as a **pyramid** of calls. Here is a visualization of calling a factorial function recursively:

```
(factorial 3)
(* 3 (factorial 2))
(* 3 (* 2 (factorial 1)))
(* 3 (* 2 (* 1 (factorial 0))))
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

Here is a visualization of the stack where each vertical dash is a stack frame:

```
        ---|---
      ---|     |---
   ---|           |---
---                   ---
```

The problem is that the stack has limited size, and stacking up these stack frames can overflow the stack. Depending on the stack size, a calculation of a larger factorial would overflow the stack. That is why regular recursion in Javascript could be considered dangerous.

An optimal execution model would be something like a **trampoline** instead of a **pyramid**, where each recursive call is executed in place, and does not stack up on the call stack. That execution in languages supporting tail call optimization could look like:

```
(fact 3)
(fact-tail 3 1)
(fact-tail 2 3)
(fact-tail 1 6)
(fact-tail 0 6)
6
```

You can visualize the stack like a bouncing trampoline:

```
   ---|---   ---|---   ---|---
---      ---       ---
```

This is clearly better since the stack has always only one frame, and from the visualization you can also see why it is called a trampoline. This prevents the stack from overflowing.

Since we don't have the luxury of `tail call optimization` in Javascript, we need to figure out a way to turn regular recursion into an optimized version that will execute in trampoline-fashion.
One obvious way is to **get rid of recursion**, and rewrite the code to be iterative.

When that is not possible we need a bit more complex code where instead of executing directly the recursive steps, we will utilize `higher order functions` to return a wrapper function instead of executing the recursive step directly, and let another function control the execution.
In your example, the **repeat** function wraps the regular recursive call with a function, and it returns that function instead of executing the recursive call:

```
function repeat(operation, num) {
    return function() {
       if (num <= 0) return
```

```
        operation()
        return repeat(operation, --num)
    }
}
```

The returned function is the next step of recursive execution and the trampoline is a mechanism to execute these steps in a controlled and iterative fashion in the while loop:

```
function trampoline(fn) {
    while(fn && typeof fn === 'function') {
        fn = fn()
    }
}
```

So the sole purpose of the trampoline function is to control the execution in an iterative way, and that ensures the stack to have only a single stack frame on the stack at any given time.

Using a trampoline is obviously less performant than simple recursion, since you are "blocking" the normal recursive flow, but it is much safer.

Q 18.17. What is throttling and debouncing in javascript?

Debouncing and throttling techniques are used to limit the number of times a function can execute. These are two widely-used techniques to improve the performance of code that gets executed repeatedly within a period of time.

Throttling enforces a maximum number of times a function can be called over time. As in "execute this function at most once every 100 milliseconds."

Example:

```
$("body").on('scroll', _.throttle(function() {
  // Do expensive things
}, 100));
```

Debouncing enforces that a function not be called again until a certain amount of time has passed without it being called. As in "execute this function only if 100 milliseconds have passed without it being called."

Example:

```
$(window).on('resize', _.debounce(function() {
  // Do expensive things
}, 100));
```

Use Case

- Throttling a button click so we can't spam click
- Throttling an API call
- Throttling a mousemove event handler
- Debouncing a resize event handler
- Debouncing a scroll event handler
- Debouncing a save function in an autosave feature

Q 18.18. What is same-origin policy?

The same-origin policy is a policy that prevents JavaScript from making requests across domain boundaries. An origin is defined as a combination of URI scheme, hostname, and port number. If you enable this policy then it prevents a malicious script on one page from obtaining access to sensitive data on another web page using Document Object Model(DOM).

Q 18.19. What is server-sent events?

Server-sent events (SSE) is a server push technology enabling a browser to receive automatic updates from a server via HTTP connection without resorting to polling. These are a one way communications channel - events flow from server to client only. This is been used in Facebook/Twitter updates, stock price updates, news feeds etc.

The `EventSource` object is used to receive server-sent event notifications. For example, we can receive messages from server as below,

```
if(typeof(EventSource) !== "undefined") {
  var source = new EventSource("sse_generator.js");
  source.onmessage = function(event) {
    document.getElementById("output").innerHTML += event.data + "<br>";
  };
}
```

Below are the list of events available for server sent events

Event	Description
onopen	It is used when a connection to the server is opened
onmessage	This event is used when a message is received
onerror	It happens when an error occurs

Q 18.20. How do you get the image width and height using JS?

You can programmatically get the image and check the dimensions(width and height) using Javascript.

```
var img = new Image();
img.onload = function() {
  console.log(this.width + 'x' + this.height);
}
img.src = 'http://www.google.com/intl/en_ALL/images/logo.gif';
```

Q 18.21. What is a browser engine?

The browser engine is to take the HTML, CSS and other code of a web page - the text you can see in the page source or open in a text editor, setting out layouts, page content, and styling - and convert it into what you actually see on screen.

The browser engine, rendering engine, and JavaScript engine are all essentially working together to get raw web code into a viewable and usable-form inside your browser.

Browser Engines:

1. Gecko:

It's a Mozilla's browser engine. It is used in the Firefox web browser, the Thunderbird email client, and the SeaMonkey internet suite. Goanna also is a fork of Gecko used in the Pale Moon browser.

2. WebKit:

This engine created by Apple for its Safari browser, by forking the KHTML engine of the KDE project. Google also used WebKit for its Chrome browser, but eventually forked it to create the Blink engine.

3. Blink:

All Chromium-based browsers use Blink, as do applications built with CEF, Electron, or any other framework that embeds Chromium.

4. Trident and EdgeHTML:

Microsoft formerly developed its own proprietary browser engines - Trident and EdgeHTML, though now uses Blink for its Edge browser.

Q 18.22. What is a Polyfill?

Polyfill

A polyfill is a piece of code (usually a JavaScript library) that provides modern functionality on older browsers that do not support it. It is a way to bring new features to old browsers by mimicking the behavior of modern JavaScript APIs.

For example, `Object.values()` was introduced in ES2017 and is not supported in some older browsers such as Internet Explorer and Safari 9. However, you can use a polyfill to add support for it in older browsers.

Example

```
// polyfill for the Object.values()
if (!Object.values) {
  Object.values = function(obj) {
    var values = [];
    for (var key in obj) {
      if (obj.hasOwnProperty(key)) {
        values.push(obj[key]);
      }
    }
    return values;
  };
}

// Now you can use Object.values() even in older browsers that don't support it
natively
const obj = { a: 1, b: 2, c: 3 };
const values = Object.values(obj);
```

```
console.log(values); // Output: [1, 2, 3]
```

This code checks if the `Array.prototype.includes()` method is available in the current environment, and if not, it provides an implementation of the method that emulates the behavior of the modern API. This allows you to use the `Array.prototype.includes()` method in older browsers, even though it is not natively supported.

Q 18.23. What is optional chaining in JavaScript?

Optional chaining

Optional chaining is a feature in JavaScript that allows you to safely access nested object properties or functions without worrying about whether the intermediate properties exist or not. It uses the `?.` operator to check for nullish (`null` or `undefined`) values and short-circuits the expression if it encounters one. Optional chaining was introduced in ECMAScript 2020 (ES11) and is supported in most modern browsers, but not in older ones. To use it in older browsers, you can use a transpiler like Babel, or write your code with alternative techniques like conditional statements or the `&&` operator.

Example

```
const obj = {
  a: {
    b: {
      c: 123
    }
  }
};

// Without optional chaining
if (obj && obj.a && obj.a.b && obj.a.b.c) {
  console.log(obj.a.b.c); // output: 123
}

// With optional chaining
console.log(obj?.a?.b?.c); // output: 123

// Optional chaining with method
console.log(obj?.a?.b?.c.toString()); // output: "123"
```

Q 18.24. How could you make sure a const value is garbage collected?

In JavaScript, garbage collection is automatically performed by the browser or the JavaScript engine. When a value is no longer being used or referenced, it becomes eligible for garbage collection.

In the case of a const value, the same rules apply as with any other value. As long as the const value is not referenced by any other variables or objects in the program, it will become eligible for garbage collection when the variable goes out of scope.

Here are some best practices to follow when working with const variables:

1. Only use `const` when you know that the value should not be reassigned.
2. If you ever want to change the contents of the variable for any reason in the future, then don't declare it as const.
3. Use `let` or `var` instead of `const` if you need to reassign the value.
4. To ensure that a const value is garbage collected, remove all references to it.

Example

```
const myObj = {name: 'John', age: 30};

// Use myObj...

// Remove all references to myObj
myObj = null;
```

In this example, setting `myObj` to `null` removes the only reference to the object, allowing it to be garbage collected.

Q 18.25. How Garbage Collection works in JavaScript?

JavaScript has an automatic garbage collector that periodically frees up memory that is no longer being used by the program. The garbage collector works by identifying "garbage" values that are no longer accessible or needed by the program and freeing up the memory they occupy.

Example

```
let a = { b: { c: { d: "Hello" } } };
let e = a.b.c;
a = null;
```

In this code, the object `{ b: { c: { d: "Hello" } } }` is created and assigned to the variable a. The variable e is then assigned a reference to the nested object `{ c: { d: "Hello" } }`. Finally, the variable a is set to null, which means that the original object `{ b: { c: { d: "Hello" } } }` is no longer accessible by the program.
At this point, the garbage collector will identify the object `{ b: { c: { d: "Hello" } } }` as "garbage" because it is no longer accessible by the program. The garbage collector will then free up the memory occupied by this object.
7799933