Created by -Swadhin Nayak

Question 1: Event Loop and SetTimeout

```javascript
console.log('Start');
setTimeout(() => {
  console.log('Timeout');
}, 0);
console.log('End');
```

Output:

```sql
Start
End
Timeout
```

Explanation:

- The `setTimeout` with `0ms` delay doesn't execute immediately. It is pushed to the event loop's queue, meaning it will be executed after the synchronous code is done.
- First, "Start" is logged, followed by "End". Once the stack is clear, the `setTimeout` callback is executed, logging "Timeout".

Question 2: Closure with Looping

```javascript
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000);
}
```

Output:

```
3
3
3
```

Explanation:

- The `var` keyword has function scope, meaning by the time the `setTimeout` callbacks run, the loop has completed, and `i` is `3`.
- The solution would be to use `let` (which has block scope) or an IIFE (Immediately Invoked Function Expression) to capture the value of `i` at each iteration.

Question 3: Object Freezing and Mutation

```javascript
const obj = Object.freeze({a: 1});
obj.a = 2;
console.log(obj.a);
```

Output:

```
1
```

Explanation:

- `Object.freeze()` makes an object immutable. Any attempts to modify its properties are silently ignored (in non-strict mode).
- Since the object is frozen, `obj.a` remains `1`.

Question 4: Array Destructuring with Rest

```javascript
const [a, ...b] = [1, 2, 3, 4];
console.log(a, b);
```

Output:

```css
1 [2, 3, 4]
```

Explanation:

- Array destructuring allows extracting values from arrays. The first value `1` is assigned to `a`.
- The rest of the array `[2, 3, 4]` is captured by the `...b` rest operator.

Question 5: This Binding in Arrow Functions

```javascript
const person = {
  name: 'John',
  greet: function() {
    setTimeout(() => {
      console.log(`Hello, ${this.name}`);
    }, 1000);
  }
};
person.greet();
```

Output:

```
Hello, John
```

Explanation:

- Arrow functions don't have their own `this` context. They capture `this` from the surrounding lexical scope, which is the `person` object in this case.
- Therefore, `this.name` correctly refers to `John`.

Question 6: Falsy Values in JavaScript

```javascript
const result = 0 || null || undefined || '' || false || 'Hello';
console.log(result);
```

Output:

```
Hello
```

Explanation:

- The `||` operator returns the first truthy value it encounters. All the values before `'Hello'` are falsy, so `'Hello'` is returned.

Question 7: Prototypal Inheritance

```javascript
function Animal(name) {
  this.name = name;
}
Animal.prototype.speak = function() {
  console.log(`${this.name} makes a noise`);
};

const dog = new Animal('Dog');
dog.speak();
```

Output:

```css
Dog makes a noise
```

Explanation:

- The `speak` method is attached to the prototype of `Animal`, so all instances of `Animal` inherit it.
- When `dog.speak()` is called, the method logs "Dog makes a noise".

Question 8: Promise Chaining

```javascript
Promise.resolve(1)
  .then(x => x + 1)
  .then(x => { throw new Error('Oops'); })
  .catch(err => console.log(err.message))
  .then(() => console.log('Done'));
```

Output:

```
Oops
Done
```

Explanation:

- The first two `then()` methods modify the value and throw an error. The `catch()` catches the error and logs the message.
- The final `then()` executes after the `catch()`, logging "Done".

Question 9: Function Hoisting

```javascript
hoisted();
function hoisted() {
  console.log('Hoisted function');
}
```

Output:

```bash
Hoisted function
```

Explanation:

- Function declarations are hoisted to the top of their scope, meaning you can call the function before it is defined in the code.

Question 10: Function Scope vs Block Scope

```javascript
if (true) {
  var x = 5;
}
console.log(x);
```

Output:

```
5
```

Explanation:

- Variables declared with `var` are function-scoped, not block-scoped. Even though `x` is inside the `if` block, it's accessible outside of it.

Question 11: Default Parameters

```javascript
function greet(name = 'Stranger') {
  console.log(`Hello, ${name}`);
}
greet();
greet('John');
```

Output:

```
Hello, Stranger
Hello, John
```

Explanation:

- Default parameters allow you to define default values for function parameters. If no value is passed to `greet()`, it uses `'Stranger'`. When `'John'` is passed, it overrides the default.

Question 12: Async/Await with Error Handling

```javascript
async function fetchData() {
  throw new Error('Data not found');
}
async function getData() {
  try {
    await fetchData();
  } catch (error) {
    console.log(error.message);
  }
}
getData();
```

Output:

```
Data not found
```

Explanation:

- The `fetchData()` function throws an error. Inside `getData()`, the `try-catch` block catches the error, and the `error.message` is logged.

Question 13: Rest Parameters in Functions

```javascript
function sum(...numbers) {
  return numbers.reduce((acc, curr) => acc + curr, 0);
}
console.log(sum(1, 2, 3, 4, 5));
```

Output:

```
15
```

Explanation:

- The rest parameter `...numbers` gathers all passed arguments into an array. The `reduce()` method then sums them up, starting from `0`.

Question 14: Map vs Object

```javascript
const map = new Map();
map.set('key', 'value');
console.log(map.get('key'));

const obj = {};
obj['key'] = 'value';
console.log(obj['key']);
```

Output:

```
value
value
```

Explanation:

- Both `Map` and `Object` can store key-value pairs, but `Map` provides more flexible and optimized methods like `.set()` and `.get()` compared to an object's bracket notation.

Question 15: Strict Mode with `this`

```javascript
function show() {
  'use strict';
  console.log(this);
```

```
}
show();
```

Output:

```
javascript
```

```
undefined
```

Explanation:

- In strict mode, `this` inside a regular function (not bound to any object) is `undefined`. Without strict mode, `this` would default to the global object (`window` in browsers).

Question 16: Short-Circuit Evaluation with `&&`

```
javascript
```

```
const result = true && 'JavaScript' && 42 && null && 'End';
console.log(result);
```

Output:

```
csharp
```

```
null
```

Explanation:

- The `&&` operator returns the first falsy value it encounters, or the last truthy value if all are truthy. In this case, `null` is falsy, so it's returned.

Question 17: Destructuring with Default Values

```
javascript
```

```
const {a = 10, b = 20} = {a: 5};
console.log(a, b);
```

Output:

```
```

```
5 20
```

Explanation:

- Destructuring allows for setting default values. The value of `a` is already provided as `5`, so the default `10` is ignored. Since `b` is not provided, it defaults to `20`.

Question 18: Chaining Optional Properties

```javascript
const person = {
  name: 'Alice',
  address: {
    city: 'Wonderland'
  }
};
console.log(person.address?.city);
console.log(person.contact?.phone);
```

Output:

```javascript
Wonderland
undefined
```

Explanation:

- The optional chaining (`?.`) operator allows you to access deeply nested properties safely. If `contact` is `undefined`, it won't throw an error, and `undefined` will be returned.

Question 19: Array Flatting

```javascript
const arr = [1, [2, [3, [4]]]];
console.log(arr.flat(2));
```

Output:

```csharp
[1, 2, 3, [4]]
```

Explanation:

- The `flat()` method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth (`2` in this case).

Question 20: Custom Iterators

```javascript
const iterable = {
  [Symbol.iterator]() {
    let count = 0;
    return {
      next() {
        if (count < 3) {
          return { value: count++, done: false };
        } else {
          return { done: true };
        }
      }
    };
  }
};
for (const value of iterable) {
  console.log(value);
}
```

Output:

```
0
1
2
```

Explanation:

- Custom iterators use the `[Symbol.iterator]()` method to implement iteration behavior. In this example, the iterator returns values from `0` to `2`, then stops when `count` reaches `3`.

Question 21: Hoisting with `let` and `var`

```javascript
console.log(a); // undefined
var a = 5;

console.log(b); // ReferenceError
let b = 10;
```

Output:

```javascript
undefined
ReferenceError: Cannot access 'b' before initialization
```

Explanation:

- Variables declared with `var` are hoisted to the top of their scope and initialized as `undefined`, so `console.log(a)` works but logs `undefined`.
- Variables declared with `let` and `const` are also hoisted, but they are placed in the "temporal dead zone" (TDZ) until they are initialized. Therefore, `console.log(b)` throws a `ReferenceError` because `b` is in the TDZ when accessed.

Question 22: `typeof` with Different Types

```javascript
console.log(typeof null);
console.log(typeof []);
console.log(typeof function() {});
```

Output:

```php
object
object
function
```

Explanation:

- In JavaScript, `typeof null` is a known quirk and returns `"object"`, even though `null` is not an object.
- Arrays are also considered objects, so `typeof []` returns `"object"`.
- Functions, however, have their own type, so `typeof function() {}` returns `"function"`.

Question 23: Immediately Invoked Function Expression (IIFE)

```javascript
(function() {
  var message = 'Hello World';
  console.log(message);
})();
console.log(message);
```

Output:

```vbnet
Hello World
ReferenceError: message is not defined
```

Explanation:

- An IIFE is a function that runs as soon as it is defined. The function creates its own scope, so `message` is not accessible outside of the function. Inside the function, "Hello World" is logged, but outside, a `ReferenceError` is thrown because `message` is not defined in the global scope.

Question 24: Shadowing with `let` and `var`

```javascript
var x = 10;
if (true) {
  let x = 20;
  console.log(x);
}
console.log(x);
```

Output:

```
20
10
```

Explanation:

- The `let` keyword creates a block-scoped variable. Inside the `if` block, the new `x` shadows the outer `x` declared with `var`. Inside the block, `x` is `20`. Outside, the block-scoped `x` is inaccessible, so `x` refers to the `var`-declared `x`, which is `10`.

Question 25: Promise Resolution Timing

```javascript
console.log('Start');
const promise = new Promise((resolve) => {
  console.log('Promise');
  resolve();
});
promise.then(() => console.log('Resolved'));
console.log('End');
```

Output:

```sql
Start
Promise
End
Resolved
```

Explanation:

- The `Promise` constructor runs synchronously, so "Promise" is logged immediately after "Start".
- However, `.then()` callbacks are asynchronous and are placed in the microtask queue. Thus, "End" is logged first, and only after the synchronous code is finished, "Resolved" is logged.

Question 26: Object Property Descriptors

```javascript
const obj = {};
Object.defineProperty(obj, 'prop', {
  value: 42,
  writable: false
});
obj.prop = 100;
console.log(obj.prop);
```

Output:

```
42
```

Explanation:

- `Object.defineProperty()` allows control over the attributes of object properties. In this case, `writable` is set to `false`, meaning the `prop` value cannot be changed. Even though `obj.prop = 100` is attempted, the change does not occur, and `obj.prop` remains `42`.

Question 27: Recursive Function with Default Parameters

```javascript
function factorial(n, acc = 1) {
  if (n <= 1) return acc;
  return factorial(n - 1, n * acc);
}
console.log(factorial(5));
```

Output:

```
120
```

Explanation:

- This is a recursive function that calculates the factorial of a number. The `acc` parameter holds the accumulated value, and each recursive call multiplies `n` with `acc`. The recursion continues until `n` is `1`, at which point the accumulated result is returned. This method is tail-recursive, optimizing the function's memory usage by avoiding stack overflow.

Question 28: Symbol as Object Property Key

```javascript
const sym = Symbol('unique');
const obj = {
  [sym]: 'Secret value'
};
console.log(obj[sym]);
console.log(obj['unique']);
```

Output:

```javascript
Secret value
undefined
```

Explanation:

- Symbols are unique and can be used as property keys in objects. The property key `[sym]` refers to the symbol, so `obj[sym]` returns "Secret value". However, since `sym` is not a string, `obj['unique']` returns `undefined` because `'unique'` is not the key.

Question 29: Object Shallow Copy with `Object.assign`

```javascript
const original = {a: 1, b: {c: 2}};
const copy = Object.assign({}, original);
copy.b.c = 3;
console.log(original.b.c);
```

Output:

```
3
```

Explanation:

- `Object.assign()` creates a shallow copy of an object, meaning only the top-level properties are copied. Nested objects (like `b`) still reference the same object. Changing `copy.b.c` affects `original.b.c`, as both `copy.b` and `original.b` refer to the same object.

Question 30: Set vs Array Uniqueness

```javascript
const array = [1, 2, 2, 3, 4, 4, 5];
const uniqueSet = new Set(array);
console.log([...uniqueSet]);
```

Output:

```csharp
[1, 2, 3, 4, 5]
```

Explanation:

- A `Set` in JavaScript automatically removes duplicate values. When the array is passed to the `Set`, it only keeps unique values (`[1, 2, 3, 4, 5]`). Using the spread operator (`[...]`), the `Set` is converted back into an array, maintaining only unique values from the original array.

Question 31: Prototype Chain Lookup

```javascript
function Vehicle() {}
Vehicle.prototype.wheels = 4;

const car = new Vehicle();
console.log(car.wheels);

car.wheels = 2;
console.log(car.wheels);

delete car.wheels;
console.log(car.wheels);
```

Output:

```
4
2
4
```

Explanation:

- Initially, `car.wheels` is not a direct property of the `car` object but is inherited from the `Vehicle.prototype`. When `car.wheels` is set to `2`, it creates a direct property on the `car` object, shadowing the prototype property. When the property is deleted with `delete`, the lookup goes back to the prototype, so `car.wheels` refers to the inherited value `4` again.

Question 32: Promise.all and Error Handling

```javascript
const p1 = Promise.resolve(1);
const p2 = Promise.reject('Error');
const p3 = Promise.resolve(3);

Promise.all([p1, p2, p3])
  .then(values => console.log(values))
  .catch(error => console.log(error));
```

Output:

```javascript
Error
```

Explanation:

- `Promise.all` executes all promises in parallel and only resolves when all promises are fulfilled. If any of the promises reject, `Promise.all` immediately rejects with that error, and no further promises are considered. In this case, `p2` rejects with `'Error'`, so the `catch()` block catches and logs it.

Question 33: Optional Chaining and Function Calls

```javascript
const obj = {
  greet() {
    return 'Hello';
  }
};
console.log(obj.greet?.());
console.log(obj.nonExistentFunction?.());
```

Output:

```javascript
Hello
undefined
```

Explanation:

- Optional chaining (`?.`) can be used for safe function calls. In `obj.greet?.()`, since `greet` exists, the function is called, and "Hello" is returned. For `obj.nonExistentFunction?.()`, the function does not exist, so `undefined` is returned instead of throwing an error.

Question 34: Currying Function

```javascript
function multiply(a) {
  return function(b) {
    return a * b;
  };
}
const double = multiply(2);
console.log(double(5));
```

Output:

```
10
```

Explanation:

- Currying is a functional programming technique where a function with multiple arguments is broken down into a series of functions that each take one argument. Here, `multiply(2)` returns a new function that multiplies its argument by `2`. When `double(5)` is called, it returns `10`.

Question 35: `for...in` vs `for...of`

```javascript
const arr = ['a', 'b', 'c'];

for (const index in arr) {
  console.log(index);
}

for (const value of arr) {
  console.log(value);
}
```

Output:

```css
0
1
2
a
b
c
```

Explanation:

- `for...in` iterates over the *keys* (or indexes) of an object or array. In this case, it logs the indices (`0`, `1`, `2`).
- `for...of` iterates over the *values* of an iterable. For the array `arr`, it logs the values (`a`, `b`, `c`).

Question 36: Object Shallow vs Deep Copy

```javascript
const obj1 = {a: 1, b: {c: 2}};
const obj2 = JSON.parse(JSON.stringify(obj1));

obj2.b.c = 3;
console.log(obj1.b.c);
```

Output:

```
2
```

Explanation:

- `JSON.parse(JSON.stringify(obj1))` creates a deep copy of `obj1`. It converts the object to a JSON string and back, ensuring that nested objects are cloned. When `obj2.b.c` is changed to `3`, it doesn't affect `obj1.b.c` because they are no longer referencing the same object in memory.

Question 37: `reduce` with Initial Value

```javascript
const numbers = [10, 20, 30];
const sum = numbers.reduce((acc, curr) => acc + curr, 0);
console.log(sum);
```

Output:

```
60
```

Explanation:

- The `reduce()` method executes a reducer function on each element of the array, accumulating a result. In this case, it sums up the numbers in the array. The `0` is the initial value of `acc`. The reducer function adds each element to `acc`, resulting in a sum of `60`.

Question 38: Array `sort` with Numbers

```javascript
const arr = [10, 1, 5, 2];
arr.sort();
console.log(arr);
```

Output:

```csharp
[1, 10, 2, 5]
```

Explanation:

- By default, `sort()` converts array elements to strings and compares their UTF-16 code unit values. Hence, the order is based on the string representation (`'1'`, `'10'`, `'2'`, `'5'`). To sort numbers properly, you need to pass a compare function like `(a, b) => a - b`.

Question 39: Tagged Template Literals

```javascript
function tag(strings, ...values) {
  return strings[0] + values[0] + strings[1] + values[1];
}
```

```
const a = 5, b = 10;
console.log(tag`Sum of ${a} and ${b} is ${a + b}`);
```

Output:

```
csharp
```

```
Sum of 5 and 10 is 15
```

Explanation:

- Tagged template literals allow you to parse a template string with a function. The `tag` function receives two arguments: an array of string literals and an array of the interpolated values. It returns a combined string, where the placeholders (`${a}`, `${b}`, etc.) are replaced by the actual values.

Question 40: Function Generators

```
javascript
```

```
function* generator() {
  yield 1;
  yield 2;
  yield 3;
}

const gen = generator();
console.log(gen.next().value);
console.log(gen.next().value);
console.log(gen.next().value);
```

Output:

```
```

```
1
2
3
```

Explanation:

- A generator function is defined using `function*`. It produces a sequence of values by using `yield` to pause the function's execution. Each time `gen.next()` is called, the generator resumes execution until it hits the next `yield` statement, returning the value. When all `yield` statements are exhausted, `done: true` will be returned.

Question 41: Difference between `null` and `undefined`

```javascript
let a;
let b = null;

console.log(a == b);
console.log(a === b);
```

Output:

```arduino
true
false
```

Explanation:

- `a` is `undefined` because it has been declared but not initialized, while `b` is explicitly assigned the value `null`.
- In the loose equality comparison (`==`), `undefined` and `null` are considered equal. However, in the strict equality comparison (`===`), they are different types, so `false` is returned.

Question 42: Prototype vs `__proto__`

```javascript
function Animal() {}
Animal.prototype.speak = function() {
  return 'Roar';
};

const lion = new Animal();
console.log(lion.speak());
console.log(lion.__proto__ === Animal.prototype);
```

Output:

```arduino
Roar
true
```

Explanation:

- The `speak` method is defined on `Animal.prototype`, so when `lion.speak()` is called, the method is found on the prototype chain.
- `__proto__` is an internal property that refers to the prototype of the object. In this case, `lion.__proto__` points to `Animal.prototype`, so the comparison returns `true`.

Question 43: Closure with Lexical Environment

```javascript
function outer() {
  let count = 0;
  return function inner() {
    count++;
    return count;
  };
}

const counter = outer();
console.log(counter());
console.log(counter());
console.log(counter());
```

Output:

```
1
2
3
```

Explanation:

- Closures allow a function to remember and access variables from its lexical environment even after it has returned. The `inner` function maintains access to the `count` variable from `outer`, so each time `counter()` is called, `count` is incremented, and the new value is returned.

```
const obj = {
  value: 100,
  getValue() {
    return this.value;
  }

}
const getValue = obj.getValue;
```

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

Output:

```javascript
undefined
100
```

Explanation:

- In JavaScript, `this` refers to the object from which the method was called. In `obj.getValue()`, `this` refers to `obj`, so it correctly returns `100`.
- However, when `getValue()` is called as a standalone function, `this` defaults to the global object (`window` in browsers). Since `value` is not defined globally, it returns `undefined`. To fix this, you can bind `getValue` to `obj` using `.bind()` or use arrow functions, which do not have their own `this`.

Question 45: `Promise.race`

```javascript
const p1 = new Promise((resolve) => setTimeout(() => resolve('First'), 500));
const p2 = new Promise((resolve) => setTimeout(() => resolve('Second'), 100));

Promise.race([p1, p2])
  .then(value => console.log(value))
  .catch(err => console.log(err));
```

Output:

```sql
Second
```

Explanation:

- `Promise.race` runs multiple promises and resolves or rejects as soon as one of them settles. In this case, `p2` resolves faster (`100ms`), so `"Second"` is logged. Even though `p1` eventually resolves, it's not considered since `Promise.race` only returns the result of the first settled promise.

Question 46: Optional Chaining with Arrays

```javascript
const arr = [[1, 2], [3, 4]];
console.log(arr[1]?.[1]); // 4
console.log(arr[2]?.[1]); // undefined
```

Output:

```javascript
4
undefined
```

Explanation:

- Optional chaining (`?.`) allows you to safely access deeply nested properties without causing a runtime error if any part of the chain is `null` or `undefined`. In this example, `arr[1]?.[1]` evaluates to `4` because the value exists, but `arr[2]?.[1]` is `undefined` because `arr[2]` is `undefined` (no third element in the array).

Question 47: WeakMap for Garbage Collection

```javascript
let obj = {key: 'value'};
const weakMap = new WeakMap();
weakMap.set(obj, 'metadata');

obj = null; // Remove reference to the object

console.log(weakMap.has(obj)); // false
```

Output:

```arduino
false
```

Explanation:

- A `WeakMap` holds "weak" references to its keys, meaning the keys can be garbage-collected if there is no other reference to them. Once the object `obj` is set to `null`, the key in the `WeakMap` is no longer accessible, and it can be garbage collected. Therefore, `weakMap.has(obj)` returns `false`.

Question 48: `Array.prototype.reduce()` for Object Transformation

```javascript
const data = [
  {id: 1, name: 'John'},
  {id: 2, name: 'Jane'},
  {id: 3, name: 'Doe'}
];

const result = data.reduce((acc, item) => {
  acc[item.id] = item.name;
  return acc;
}, {});

console.log(result);
```

Output:

```css
{ 1: 'John', 2: 'Jane', 3: 'Doe' }
```

Explanation:

- The `reduce()` function is used here to transform an array of objects into an object, where the `id` becomes the key and the `name` becomes the value. The `acc` (accumulator) starts as an empty object (`{}`), and for each iteration, a new property is added, ultimately producing the result `{ 1: 'John', 2: 'Jane', 3: 'Doe' }`.

Question 49: `Proxy` for Object Interception

```javascript
const target = {message: 'Hello'};
const handler = {
  get: function(obj, prop) {
    if (prop === 'message') {
      return 'Intercepted';
    }
    return obj[prop];
  }
};

const proxy = new Proxy(target, handler);
console.log(proxy.message); // Intercepted
console.log(proxy.nonExistent); // undefined
```

Output:

```javascript
Intercepted
undefined
```

Explanation:

- A `Proxy` allows you to intercept and redefine basic operations on an object, such as property access (`get`). In this example, when the `message` property is accessed, the `get` trap returns `'Intercepted'`. For other properties, it defaults to returning the original property from the target object.

Question 50: Template Literal with Expression Evaluation

```javascript
const a = 5;
const b = 10;
console.log(`${a} + ${b} = ${a + b}`);
```

Output:

```
5 + 10 = 15
```

Explanation:

- Template literals (backticks ``) allow embedded expressions and multiline strings. The expression `${a + b}` is evaluated to `15` within the template string. This makes it easy to construct strings with dynamic values without needing concatenation.

Question 51: Recursive Fibonacci with Memoization

```javascript
function fibonacci(n, memo = {}) {
  if (n <= 1) return n;
  if (memo[n]) return memo[n];
  return memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
}

console.log(fibonacci(6));
```

Output:

```
8
```

Explanation:

- This is a memoized recursive implementation of the Fibonacci sequence. `memo` stores previously computed Fibonacci numbers to avoid redundant calculations. For `fibonacci(6)`, the function

computes `fibonacci(5) + fibonacci(4)` and so on, but memoization ensures that each Fibonacci value is computed only once.

Question 52: Destructuring Assignment with Default Values

```javascript
const [a = 5, b = 10] = [1];
console.log(a); // 1
console.log(b); // 10
```

Output:

```
1
10
```

Explanation:

- In array destructuring, default values can be provided for variables if there is no value in the array at that position. Here, `a` gets the value `1` from the array, but `b` does not have a corresponding value in the array, so it uses the default value of `10`.

Question 53: Promise Chaining with Error Handling

```javascript
Promise.resolve('Success')
  .then(value => {
    console.log(value);
    throw new Error('Something went wrong');
  })
  .catch(error => {
    console.log(error.message);
  });
```

Output:

```mathematica
Success
Something went wrong
```

Explanation:

- A promise chain executes sequentially. The first `then()` block logs "Success" and then throws an error. The error is caught by the `catch()` block, which logs the error message. Throwing an error in a `then()` block causes the promise to reject, which triggers the `catch()` block.

Question 54: Nested Objects with Spread Operator

```javascript
const obj1 = {a: 1, b: {c: 2}};
const obj2 = {...obj1};
obj2.b.c = 3;
console.log(obj1.b.c);
```

Output:

```
3
```

Explanation:

- The spread operator (`{...obj1}`) creates a shallow copy of `obj1`. This means that while the top-level properties are copied, nested objects (`b` in this case) still reference the same object in memory. Thus, modifying `obj2.b.c` also changes `obj1.b.c` because they share the same reference for `b`.

Question 55: String Reversal with `split`, `reverse`, `join`

```javascript
const str = 'JavaScript';
const reversed = str.split('').reverse().join('');
console.log(reversed);
```

Output:

```
tpircSavaJ
```

Explanation:

- The string is first split into an array of characters using `split('')`. Then, the array is reversed with `reverse()`, and the characters are joined back together into a string using `join('')`. This effectively reverses the original string.

Question 56: Nullish Coalescing Operator (`??`)

```javascript
let a = null;
let b = 'default';
```

```
  console.log(a ?? b);
```

Output:

```
arduino
```

```
  default
```

Explanation:

- The nullish coalescing operator (`??`) returns the right-hand side operand when the left-hand side operand is `null` or `undefined`. In this case, since `a` is `null`, `b` (`'default'`) is returned. This operator is useful when handling potential `null` or `undefined` values.

Question 57: Object Freezing with `Object.freeze()`

```
javascript
```

```
const obj = {name: 'John'};
Object.freeze(obj);
obj.name = 'Jane';
console.log(obj.name);
```

Output:

```
```

```
  John
```

Explanation:

- `Object.freeze()` makes an object immutable. Once an object is frozen, its properties cannot be modified, added, or deleted. In this case, trying to change `obj.name` to `'Jane'` has no effect, and `obj.name` remains `'John'`.

Question 58: Debouncing Function

```
javascript
```

```
function debounce(func, delay) {
  let timeout;
  return function(...args) {
    clearTimeout(timeout);
    timeout = setTimeout(() => func.apply(this, args), delay);
  };
}

const log = debounce(() => console.log('Debounced!'), 2000);
log();
```

```
log();
log();
```

Output:

```
Debounced!
```

Explanation:

- Debouncing is a technique to limit the rate at which a function is executed. In this example, `log()` is called multiple times in quick succession, but the function inside `debounce` is executed only once after `2000ms` have passed since the last call. This is achieved by resetting the timer each time `log()` is called, ensuring the function only executes after a pause in the rapid calls.

Question 59: Rest Operator in Function Arguments

```javascript
function sum(...numbers) {
  return numbers.reduce((acc, curr) => acc + curr, 0);
}

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

Output:

```
10
```

Explanation:

- The rest operator (`...`) collects all the arguments passed into the function into an array. In this case, `numbers` becomes `[1, 2, 3, 4]`, which is then summed up using `reduce()`. The rest operator allows handling a variable number of function arguments in a concise way.

Question 60: Event Loop and Microtasks (Promises)

```javascript
console.log('Start');
setTimeout(() => console.log('Timeout'), 0);
Promise.resolve().then(() => console.log('Promise'));
console.log('End');
```

Output:

```sql
Start
End
Promise
Timeout
```

Explanation:

- JavaScript's event loop handles execution order. First, synchronous code (`console.log('Start')` and `console.log('End')`) runs. Promises (`Promise.resolve().then(...)`) are added to the microtask queue, which has higher priority than the task queue (where `setTimeout` callbacks are placed). Therefore, `"Promise"` is logged before `"Timeout"`.

Question 61: Set Object and Uniqueness of Elements

```javascript
const mySet = new Set([1, 2, 2, 3, 4, 4]);
console.log(mySet);
```

Output:

```javascript
Set { 1, 2, 3, 4 }
```

Explanation:

- A `Set` is a collection of unique values. When you pass an array with duplicate elements to a `Set`, it automatically removes duplicates. In this example, the array `[1, 2, 2, 3, 4, 4]` has duplicate values (`2` and `4`), which are removed in the `Set`, leaving only `{ 1, 2, 3, 4 }`.

Question 62: Rest and Spread with Object Destructuring

```javascript
const person = {name: 'John', age: 30, job: 'Developer'};
const {name, ...rest} = person;
console.log(name); // John
console.log(rest); // {age: 30, job: 'Developer'}
```

Output:

```css
John
{ age: 30, job: 'Developer' }
```

Explanation:

- In object destructuring, the rest operator (`...rest`) allows you to collect the remaining properties of the object into a new object. In this case, `name` is extracted, and the rest of the properties (`age` and `job`) are gathered into a new object `rest`, which contains `{ age: 30, job: 'Developer' }`.

Question 63: Asynchronous Function with `async` and `await`

```javascript
async function fetchData() {
  const data = await new Promise(resolve => setTimeout(() => resolve('Fetched data'), 1000));
  console.log(data);
}
fetchData();
console.log('Waiting...');
```

Output:

```kotlin
Waiting...
Fetched data
```

Explanation:

- The `async` keyword makes a function return a promise, and the `await` keyword pauses the function's execution until the promise is resolved. In this example, `fetchData` logs "Fetched data" only after waiting for 1000ms. The `console.log('Waiting...')` runs first because `await` allows the JavaScript engine to handle other tasks while waiting for the promise.

Question 64: `Map` Object and Iterating Over Entries

```javascript
const myMap = new Map([
  ['key1', 'value1'],
  ['key2', 'value2']
]);

for (const [key, value] of myMap) {
  console.log(`${key}: ${value}`);
}
```

Output:

```makefile
key1: value1
key2: value2
```

Explanation:

- A `Map` object stores key-value pairs and preserves the insertion order of the keys. You can iterate over a `Map` using a `for...of` loop, which returns an array of `[key, value]` pairs. In this example, the map contains two pairs, which are logged in the order they were inserted.

Question 65: Default Parameters in Functions

```javascript
function greet(name = 'Guest') {
  console.log(`Hello, ${name}!`);
}

greet('John');
greet();
```

Output:

```
Hello, John!
Hello, Guest!
```

Explanation:

- Default parameters allow you to provide default values for function arguments. If the argument is not provided or is `undefined`, the default value will be used. In this case, when `greet()` is called without an argument, the `name` defaults to `'Guest'`.

Question 66: Function Overriding in Prototypes

```javascript
function Animal() {}
Animal.prototype.speak = function() {
  return 'Roar';
};

function Dog() {}
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.speak = function() {
  return 'Bark';
};

const dog = new Dog();
console.log(dog.speak());
```

Output:

```
Bark
```

Explanation:

- In this example, `Dog` is a constructor function that inherits from `Animal`. The `Dog.prototype` is set to an object created from `Animal.prototype`, establishing inheritance. However, `Dog.prototype.speak` is overridden to return `'Bark'`, which replaces the inherited `speak()` method from `Animal.prototype`. Therefore, when `dog.speak()` is called, it returns `'Bark'`.

Question 67: IIFE (Immediately Invoked Function Expression)

```javascript
(function() {
  console.log('IIFE executed!');
})();
```

Output:

```
IIFE executed!
```

Explanation:

- An IIFE is a function that is defined and executed immediately after its definition. It is enclosed in parentheses, and a set of parentheses at the end invokes the function. In this case, the function is executed immediately, logging `"IIFE executed!"`.

Question 68: Event Delegation

```html
<ul id="parent">
  <li>Item 1</li>
  <li>Item 2</li>
</ul>

<script>
document.getElementById('parent').addEventListener('click', function(event) {
  console.log('Clicked:', event.target.textContent);
});
</script>
```

Expected Output (upon clicking any `li` element):

```makefile
Clicked: Item 1
Clicked: Item 2
```

Explanation:

- Event delegation is a technique where a single event listener is added to a parent element to handle events on its child elements. In this example, the `click` event is attached to the `<ul>` element, but when an `<li>` is clicked, `event.target` refers to the `<li>` that was clicked, allowing the event handler to react to clicks on child elements.

Question 69: Currying with Multiple Functions

```javascript
function multiply(a) {
  return function(b) {
    return function(c) {
      return a * b * c;
    };
  };
}

console.log(multiply(2)(3)(4)); // 24
```

Output:

```
24
```

Explanation:

- Currying is a technique where a function with multiple arguments is transformed into a sequence of functions, each taking one argument. In this example, `multiply(2)(3)(4)`

successively applies the values `2`, `3`, and `4` to the curried functions, eventually returning the product `24`.

Question 70: Falsy Values in JavaScript

```javascript
const falsyValues = [false, 0, '', null, undefined, NaN];
falsyValues.forEach(value => {
  if (!value) {
    console.log(`${value} is falsy`);
  }
});
```

Output:

```csharp
false is falsy
0 is falsy
 is falsy
null is falsy
undefined is falsy
NaN is falsy
```

Explanation:

- In JavaScript, falsy values are values that are considered `false` when evaluated in a Boolean context. These include `false`, `0`, `''` (empty string), `null`, `undefined`, and `NaN`. The code iterates through the `falsyValues` array and logs each value that is falsy.

Question 71: Recursive Function for Factorial

```javascript
function factorial(n) {
  if (n === 0) return 1;
  return n * factorial(n - 1);
}

console.log(factorial(5));
```

Output:

```
120
```

Explanation:

- A recursive function calls itself until it reaches a base case. In this example, the factorial function calculates `n!` by recursively multiplying `n` with the factorial of `n - 1`. The base case occurs when `n === 0`, which returns `1`. For `factorial(5)`, it computes `5 * 4 * 3 * 2 * 1 = 120`.

Question 72: Combining Multiple Arrays Using `flatMap()`

```javascript
const arrays = [[1, 2], [3, 4], [5, 6]];
const flatArray = arrays.flatMap(arr => arr);
console.log(flatArray);
```

Output:

```csharp
[1, 2, 3, 4, 5, 6]
```

Explanation:

- The `flatMap()` method first applies a mapping function to each element and then flattens the resulting arrays into a single array. In this case, each subarray `[1, 2]`, `[3, 4]`, `[5, 6]` is flattened into one array `[1, 2, 3, 4, 5, 6]`. This is useful when dealing with nested arrays.

Question 73: Deep Cloning with JSON Methods

```
const obj = { name: 'John', address: { city: 'New York' } };
const clone = JSON.parse(JSON.stringify(obj));

clone.address.city = 'Los Angeles';
console.log(obj.address.city); // New York
```

Output:

```sql
New York
```

Explanation:

- Using `JSON.stringify()` and `JSON.parse()` is a quick way to create a deep copy of an object. A deep copy means that nested objects are also copied, not just references. In this case, when the `city` property in `clone.address` is modified, it does not affect the original `obj.address.city`, because they are separate objects.

Question 74: Symbol as a Unique Key

```javascript
const sym1 = Symbol('id');
const sym2 = Symbol('id');

const obj = {
  [sym1]: 'Symbol 1',
  [sym2]: 'Symbol 2'
};

console.log(obj[sym1]); // Symbol 1
console.log(obj[sym2]); // Symbol 2
```

Output:

```javascript
Symbol 1
Symbol 2
```

Explanation:

- Symbols are unique and immutable data types used as object keys in JavaScript. Even though `sym1` and `sym2` have the same description (`'id'`), they are different symbols and refer to different properties in the object. Each symbol is guaranteed to be unique, so the values in the object are not overwritten.

Question 75: Optional Chaining Operator (`?.`)

```javascript
const user = {
  name: 'Alice',
  details: { age: 25 }
};
```

```
console.log(user.details?.age); // 25
console.log(user.address?.city); // undefined
```

Output:

```
javascript
```

```
25
undefined
```

Explanation:

- The optional chaining operator (`?.`) allows you to safely access deeply nested properties without throwing an error if any of the intermediate properties are `null` or `undefined`. In this example, `user.details?.age` returns `25`, but `user.address?.city` returns `undefined` because `address` does not exist on `user`.

Question 76: Handling `this` in Arrow Functions

```
javascript
```

```
const person = {
  name: 'John',
  greet: function() {
    setTimeout(() => {
      console.log(`Hello, ${this.name}`);
    }, 1000);
  }
};

person.greet();
```

Output:

```
```

```
Hello, John
```

Explanation:

- Arrow functions do not have their own `this` context; instead, they inherit `this` from the surrounding lexical scope. In this case, the `this` inside the arrow function in `setTimeout` refers to the `person` object, allowing it to access `this.name` and log `'Hello, John'`.

Question 77: `Array.prototype.reduce()` for Flattening Arrays

```
javascript
```

```
const arrays = [[1, 2], [3, 4], [5, 6]];
const flatArray = arrays.reduce((acc, curr) => acc.concat(curr), []);
```

```
console.log(flatArray);
```

Output:

```
csharp
```

```
[1, 2, 3, 4, 5, 6]
```

Explanation:

- The `reduce()` method applies a function to each element in the array, accumulating the results into a single value. In this example, `reduce()` is used to flatten the nested arrays by concatenating each inner array to the accumulator, resulting in a single flattened array `[1, 2, 3, 4, 5, 6]`.

Question 78: `Array.prototype.find()`

```
javascript
```

```
const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
  { id: 3, name: 'Charlie' }
];

const user = users.find(user => user.id === 2);
console.log(user);
```

Output:

```
bash
```

```
{ id: 2, name: 'Bob' }
```

Explanation:

- The `find()` method returns the first element in the array that satisfies the provided testing function. In this example, it looks for a user object where `id === 2`. When it finds such a user (`{ id: 2, name: 'Bob' }`), it returns that object.

Question 79: String Template Literals and Expression Interpolation

```
javascript
```

```
const firstName = 'John';
const lastName = 'Doe';
```

```
const fullName = `${firstName} ${lastName}`;
console.log(fullName); // John Doe
```

Output:

```
John Doe
```

Explanation:

- Template literals (enclosed by backticks `` ``) allow you to embed expressions inside strings using `${expression}`. In this example, the values of `firstName` and `lastName` are interpolated into the template literal, resulting in the string `"John Doe"`.

Question 80: Sorting an Array of Objects

```javascript
const users = [
  { name: 'John', age: 30 },
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 35 }
];

users.sort((a, b) => a.age - b.age);
console.log(users);
```

Output:

```yaml
[
  { name: 'Alice', age: 25 },
  { name: 'John', age: 30 },
  { name: 'Bob', age: 35 }
]
```

Explanation:

- The `sort()` method sorts the elements of an array in place. In this case, it compares the `age` properties of the objects in the `users` array and sorts them in ascending order. The callback function `a.age - b.age` ensures that the array is sorted by the `age` values of the user objects.

Question 81: Object Property Shortcuts

```javascript
const name = 'Alice';
const age = 25;
```

```
const person = { name, age };
console.log(person);
```

Output:

```
css
```

```
{ name: 'Alice', age: 25 }
```

Explanation:

- When the variable name is the same as the object property name, you can use the shorthand syntax in object literals. Instead of writing `` `name: name` ``, you can simply write `` `{ name }` ``. This is a concise way to create objects when the property names match the variable names.

Question 82: `` `Array.prototype.every()` ``

```
javascript
```

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

const allPositive = numbers.every(num => num > 0);
console.log(allPositive); // true
```

Output:

```
arduino
```

```
true
```

Explanation:

- The `` `every()` `` method tests whether all elements in an array pass a provided test. It returns `` `true` `` if all elements satisfy the condition, and `` `false` `` otherwise. In this case, it checks if every number in the `` `numbers` `` array is greater than `` `0` ``, and since all elements are positive, it returns `` `true` ``.

Question 83: Chaining Promises

```
javascript
```

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve(1), 1000);
});

promise
  .then(result => {
    console.log(result); // 1
    return result * 2;
```

```
  })
  .then(result => {
    console.log(result); // 2
    return result * 3;
  })
  .then(result => {
    console.log(result); // 6
  });
```

Output:

```
1
2
6
```

Explanation:

- Promises can be chained using the `then()` method. Each `then()` call returns a new promise, which can be used to chain further operations. In this example, the promise resolves with the value `1`, and each subsequent `then()` modifies the result and passes it to the next handler. The chain produces the values `1`, `2`, and `6`.

Question 84: Object Shallow vs. Deep Copy with `Object.assign()`

```javascript
const obj = { name: 'Alice', details: { age: 25 } };
const shallowCopy = Object.assign({}, obj);
shallowCopy.details.age = 30;

console.log(obj.details.age); // 30
```

Output:

```
30
```

Explanation:

- `Object.assign()` creates a shallow copy of an object. This means that while the top-level properties are copied, any nested objects are still referenced by both the original and the copied object. In this case, modifying `shallowCopy.details.age` also modifies `obj.details.age`, as both objects share the same reference to the `details` object.

Question 85: Memoization with Functions

```javascript
function memoize(fn) {
  const cache = {};
  return function(...args) {
    const key = JSON.stringify(args);
    if (cache[key]) {
      return cache[key];
    }
    const result = fn(...args);
    cache[key] = result;
    return result;
  };
}

const add = memoize((a, b) => a + b);
console.log(add(1, 2)); // 3
console.log(add(1, 2)); // 3 (from cache)
```

Output:

```
3
3
```

Explanation:

- Memoization is a technique used to cache the results of expensive function calls. In this example, the `memoize` function wraps another function (`add`). When the function is called with the same arguments, the result is fetched from the cache instead of recomputing it. This improves performance by avoiding redundant calculations.

Question 86: Object Destructuring with Default Values

```javascript
const user = { name: 'John' };
const { name, age = 30 } = user;

console.log(name); // John
console.log(age); // 30
```

Output:

```
John
30
```

Explanation:

- When destructuring an object, you can assign default values to variables. In this case, `age` is not present in the `user` object, so it defaults to `30`. This provides a fallback value in case the property is `undefined` or missing.

Question 87: Handling Promises with `Promise.all()`

```javascript
const promise1 = Promise.resolve(1);
const promise2 = new Promise((resolve) => setTimeout(() => resolve(2), 1000));
const promise3 = Promise.resolve(3);

Promise.all([promise1, promise2, promise3]).then(values => {
  console.log(values);
});
```

Output:

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

Explanation:

- `Promise.all()` takes an array of promises and returns a new promise that resolves when all the promises in the array have resolved. The resolved value is an array of results from the input promises. In this example, once all three promises are resolved (after 1 second due to the second promise), the result `[1, 2, 3]` is logged.

Question 88: Function Binding with `bind()`

```javascript
const person = {
  name: 'Alice',
  greet: function() {
    console.log(`Hello, ${this.name}`);
  }
};

const greet = person.greet.bind({ name: 'Bob' });
greet();
```

Output:

```
Hello, Bob
```

Explanation:

- The `bind()` method creates a new function that, when called, has its `this` value set to the provided argument. In this case, the `greet` function is bound to the object `{ name: 'Bob' }`, so when `greet()` is called, it logs `"Hello, Bob"`, even though the original function was associated with `person`.

Question 89: Sorting Strings Alphabetically

```javascript
const fruits = ['banana', 'apple', 'orange'];
fruits.sort();
console.log(fruits);
```

Output:

```css
["apple", "banana", "orange"]
```

Explanation:

- The `sort()` method, when called on an array of strings, sorts the elements in lexicographical (alphabetical) order by default. In this case, `['banana', 'apple', 'orange']` is sorted as `['apple', 'banana', 'orange']`.

Question 90: Hoisting of Function Declarations

```javascript
console.log(foo()); // 'Hello'

function foo() {
  return 'Hello';
}
```

Output:

```
Hello
```

Explanation:

- In JavaScript, function declarations are hoisted to the top of their scope. This means the entire function `foo()` is available before it is invoked, even if it appears later in the code. Therefore, calling `foo()` before its declaration works as expected and returns `'Hello'`.

Question 91: Understanding `instanceof` Operator

```javascript
function Animal() {}
function Dog() {}

Dog.prototype = Object.create(Animal.prototype);

const myDog = new Dog();
console.log(myDog instanceof Dog); // true
console.log(myDog instanceof Animal); // true
```

Output:

```arduino
true
true
```

Explanation:

- The `instanceof` operator checks if an object is an instance of a constructor or its prototype chain. Since `Dog.prototype` is created from `Animal.prototype`, `myDog` is an instance of both `Dog` and `Animal`. Therefore, both `instanceof` checks return `true`.

Question 92: Optional Catch Binding

```javascript
try {
  throw new Error('Oops!');
} catch {
  console.log('Error caught');
}
```

Output:

```javascript

```

```
Error caught
```

Explanation:

- In modern JavaScript (starting from ES2019), you can omit the `catch` binding parameter. This allows you to catch errors without needing to specify the error object if you don't plan on using it. In this example, the `catch` block runs even without specifying the error variable, logging `'Error caught'`.

Question 93: Promise Rejection Handling with `catch()`

```javascript
const promise = new Promise((resolve, reject) => {
  reject('Something went wrong');
});

promise
  .then(() => console.log('Success'))
  .catch(error => console.log(error));
```

Output:

```

```

```
Something went wrong
```

Explanation:

- When a promise is rejected, the `catch()` method is used to handle the error. In this case, the promise is immediately rejected with the message `'Something went wrong'`, so the `catch()` block is executed, logging the error message.

Question 94: Using `Set` to Remove Duplicates from Array

```javascript
const numbers = [1, 2, 2, 3, 4, 4, 5];
const uniqueNumbers = [...new Set(numbers)];

console.log(uniqueNumbers);
```

Output:

```
csharp
```

```
[1, 2, 3, 4, 5]
```

Explanation:

- The `Set` object automatically removes duplicate values, as it only stores unique elements. In this example, creating a `Set` from the `numbers` array removes the duplicates. The result is then converted back into an array using the spread operator (`...`), giving `[1, 2, 3, 4, 5]`.

Question 95: Recursive Fibonacci Function

```
javascript
```

```
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

console.log(fibonacci(5));
```

Output:

```

```

```
5
```

Explanation:

- The Fibonacci sequence is defined such that each number is the sum of the two preceding ones. This recursive function computes the Fibonacci number for a given `n`. For `fibonacci(5)`, the sequence follows: `0, 1, 1, 2, 3, 5`, and the result is `5`.

Question 96: Using `Array.prototype.splice()`

```javascript
const fruits = ['apple', 'banana', 'cherry', 'date'];
fruits.splice(2, 1, 'blueberry');
console.log(fruits);
```

Output:

```css
["apple", "banana", "blueberry", "date"]
```

Explanation:

- The `splice()` method changes the contents of an array by removing or replacing existing elements and/or adding new elements in place. In this case, it starts at index `2`, removes `1` element (`'cherry'`), and adds `'blueberry'` in its place. The resulting array is `['apple', 'banana', 'blueberry', 'date']`.

Question 97: Using `Array.prototype.flat()` to Flatten Arrays

```javascript
const nestedArray = [1, [2, [3, 4]], 5];
const flatArray = nestedArray.flat(2);
console.log(flatArray);
```

Output:

```csharp
[1, 2, 3, 4, 5]
```

Explanation:

- The `flat()` method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth. Here, `nestedArray` is flattened to a depth of `2`, resulting in a single-level array: `[1, 2, 3, 4, 5]`.

Question 98: String Methods - `slice()` vs. `substring()`

```javascript
const str = 'Hello, World!';
console.log(str.slice(0, 5));    // Hello
console.log(str.substring(0, 5)); // Hello
```

```
console.log(str.slice(-6));       // World!
console.log(str.substring(-6));    // Hello
```

Output:

```
Hello
Hello
World!
Hello
```

Explanation:

- The `slice()` method extracts a section of a string and returns it as a new string. It can accept negative indices, which count back from the end of the string. The `substring()` method, on the other hand, does not support negative indices and treats them as `0`. In this example, `slice(-6)` correctly extracts `'World!'`, while `substring(-6)` returns the full string from the start, which is `'Hello, World!'`.

Question 99: Dynamic Object Keys

```javascript
const key = 'name';
const person = {
  [key]: 'Alice',
  age: 25
};

console.log(person.name); // Alice
```

Output:

```
Alice
```

Explanation:

- You can define object keys dynamically using computed property names. By wrapping an expression in square brackets, you can use the value of a variable (`key` in this case) as the property name. Here, `person` has a property named `'name'` with the value `'Alice'`, which is accessed using `person.name`.

Question 100: Using `Promise.race()`

```javascript
const promise1 = new Promise((resolve) => setTimeout(() => resolve('First!'), 1000));
const promise2 = new Promise((resolve) => setTimeout(() => resolve('Second!'), 500));

Promise.race([promise1, promise2]).then(result => {
  console.log(result);
});
```

Output:

```sql
Second!
```

Explanation:

- `Promise.race()` returns a promise that resolves or rejects as soon as one of the promises in the iterable resolves or rejects. In this case, `promise2` resolves first (after 500 ms), so the output is `'Second!'`. This method is useful when you want to get the result of the fastest promise.

Question 101: Using `Array.prototype.includes()`

```javascript
const numbers = [1, 2, 3, 4, 5];
console.log(numbers.includes(3));  // true
console.log(numbers.includes(6));  // false
```

Output:

```arduino
true
false
```

Explanation:

- The `includes()` method determines whether an array includes a certain value among its entries, returning `true` or `false` as appropriate. In this example, `numbers.includes(3)` returns `true`, while `numbers.includes(6)` returns `false`, indicating the absence of `6` in the array.

Question 102: Object Iteration with `for...in`

```javascript
const car = {
  make: 'Toyota',
  model: 'Camry',
  year: 2021
};

for (const key in car) {
  console.log(`${key}: ${car[key]}`);
}
```

Output:

```yaml
make: Toyota
model: Camry
year: 2021
```

Explanation:

- The `for...in` statement iterates over the enumerable properties of an object. In this case, it goes through each property in the `car` object, logging both the key and the corresponding value. It's important to note that `for...in` may also iterate over properties inherited through the prototype chain unless filtered with `hasOwnProperty()`.

Question 103: Creating and Using Prototypes

```javascript
function Person(name) {
  this.name = name;
}

Person.prototype.greet = function() {
  console.log(`Hello, my name is ${this.name}`);
};

const alice = new Person('Alice');
alice.greet();
```

Output:

```csharp
Hello, my name is Alice
```

Explanation:

- Prototypes allow you to define methods and properties that can be shared among all instances of a constructor function. In this example, `greet` is defined on `Person.prototype`, meaning all

instances of `Person` (like `alice`) have access to this method, enabling it to log the greeting.

Question 104: Event Delegation

```html
<ul id="list">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>

<script>
  document.getElementById('list').addEventListener('click', function(event) {
    if (event.target.tagName === 'LI') {
      console.log(`Clicked on: ${event.target.textContent}`);
    }
  });
</script>
```

Output:

```csharp
Clicked on: Item 1
Clicked on: Item 2
Clicked on: Item 3
```

Explanation:

- Event delegation is a technique used to handle events at a higher level in the DOM rather than on individual elements. Here, a click event listener is added to the `ul` element, which checks if the clicked element is an `li`. This way, it can handle clicks on any number of list items without needing separate event listeners for each one.

Question 105: Understanding Closures

```
function outerFunction() {
  let outerVariable = 'I am outside!';
```

```
    function innerFunction() {
      console.log(outerVariable);
    }

    return innerFunction;
  }

  const closureFunction = outerFunction();
  closureFunction();
```

Output:

```
css

  I am outside!
```

Explanation:

- A closure is a function that retains access to its lexical scope, even when the function is executed outside that scope. In this example, `**`innerFunction`**` is returned from `**`outerFunction`**`, allowing it to access `**`outerVariable`**`, which exists in the outer function's scope. Calling `**`closureFunction`**` logs `**`'I am outside!'`**` despite `**`outerFunction`**` having finished executing.

Question 106: Function Currying

```
javascript

  function multiply(a) {
    return function(b) {
      return a * b;
    };
  }

  const double = multiply(2);
  console.log(double(5)); // 10
```

Output:

```

  10
```

Explanation:

- Currying is a technique of transforming a function with multiple arguments into a sequence of functions, each taking a single argument. Here, `**`multiply`**` returns another function that takes `**`b`**` and multiplies it by `**`a`**`. The `**`double`**` variable is a partially applied function that doubles its input by fixing `**`a`**` to `**`2`**`.

Question 107: Array Reduction with `**`reduce()`**`

```javascript
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, current) => accumulator + current, 0);
console.log(sum);
```

Output:

```
15
```

Explanation:

- The `reduce()` method executes a reducer function (that you provide) on each element of the array, resulting in a single output value. In this case, it sums all the numbers in the array. The first argument to `reduce` is the accumulator (which accumulates the values), and the second argument `0` is the initial value.

Question 108: Using `this` in Arrow Functions

```javascript
const obj = {
  value: 42,
  getValue: function() {
    return () => this.value;
  }
};

const getValue = obj.getValue();
console.log(getValue()); // 42
```

Output:

```
42
```

Explanation:

- Arrow functions do not have their own `this` context; they inherit `this` from the parent scope where they are defined. In this example, the arrow function inside `getValue` captures `this` from the `getValue` method, which points to `obj`, thus allowing `getValue()` to access `obj.value` and return `42`.

Question 109: Event Bubbling vs. Event Capturing

```html
<div id="parent" style="padding: 20px; background: lightblue;">
  Parent
  <button id="child">Click me!</button>
</div>

<script>
  document.getElementById('parent').addEventListener('click', function() {
    console.log('Parent clicked!');
  }, true); // capturing phase

  document.getElementById('child').addEventListener('click', function(event) {
    console.log('Child clicked!');
    event.stopPropagation(); // prevents bubbling
  });
</script>
```

Output:

```
Child clicked!
```

Explanation:

- Event bubbling and capturing are two phases of event propagation in the DOM. Bubbling is when the event propagates from the target element up to the root, while capturing is when it goes from the root down to the target. In this example, the parent element has an event listener set for the capturing phase (`true`). When the button is clicked, the message from the child element is logged first, and `stopPropagation()` prevents the event from reaching the parent.

Question 110: Using `Promise.allSettled()`

```javascript
const promise1 = Promise.resolve(3);
const promise2 = new Promise((resolve, reject) => setTimeout(reject, 100, 'error'));
const promise3 = Promise.resolve(5);

Promise.allSettled([promise1, promise2, promise3]).then(results => {
  console.log(results);
});
```

Output:

```lua
[
  { status: 'fulfilled', value: 3 },
  { status: 'rejected', reason: 'error' },
  { status: 'fulfilled', value: 5 }
]
```

Explanation:

- `Promise.allSettled()` returns a promise that resolves after all of the given promises have either resolved or rejected. It provides an array of objects describing the outcome of each promise. In this example, the results include both fulfilled and rejected statuses, allowing you to handle all promises without failing early.

Question 111: Object.entries() and Object.fromEntries()

```javascript
const obj = { a: 1, b: 2, c: 3 };
const entries = Object.entries(obj);
console.log(entries); // [['a', 1], ['b', 2], ['c', 3]]

const newObj = Object.fromEntries(entries);
console.log(newObj); // { a: 1, b: 2, c: 3 }
```

Output:

```css
[['a', 1], ['b', 2], ['c', 3]]
{ a: 1, b: 2, c: 3 }
```

Explanation:

- `Object.entries()` converts an object into an array of its key-value pairs. `Object.fromEntries()` does the reverse, transforming an array of key-value pairs back into an object. In this case, `Object.entries(obj)` produces an array of entries, which can be converted back to an object using `Object.fromEntries()`.

Question 112: Understanding Symbols

```javascript
const sym1 = Symbol('description');
const sym2 = Symbol('description');
console.log(sym1 === sym2); // false
```

Output:

```arduino
false
```

Explanation:

- Symbols are a new primitive type in JavaScript introduced in ES6. They are unique and immutable, making them ideal for use as object property keys when you want to avoid name collisions. In this example, `sym1` and `sym2` have the same description but are different symbols, so the comparison returns `false`.

Question 113: Using `Object.freeze()`

```javascript
const obj = { name: 'Alice' };
Object.freeze(obj);

obj.name = 'Bob'; // No effect
console.log(obj.name); // Alice
```

Output:

```
Alice
```

Explanation:

- `Object.freeze()` prevents the modification of existing property attributes and values, and prevents the addition of new properties. In this example, the object `obj` cannot be modified after freezing, so trying to change `obj.name` has no effect. The original value remains unchanged.

Here

Question 114: Using `async`/`await` for Asynchronous Operations

```javascript
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('Data fetched!');
    }, 1000);
  });
}

async function getData() {
  const result = await fetchData();
```

```
  console.log(result);
}

getData();
```

Output:

```

Data fetched!
```

Explanation:

- The `async` function allows you to use `await` inside it, which pauses the execution of the function until the promise is resolved. In this case, `fetchData` simulates an asynchronous operation that resolves after 1 second. The `getData` function waits for this promise to resolve before logging the result.

Question 115: Using `Set` to Handle Unique Values

```javascript
const numbers = [1, 2, 3, 1, 2, 4, 5];
const uniqueNumbers = new Set(numbers);
console.log([...uniqueNumbers]);
```

Output:

```csharp
[1, 2, 3, 4, 5]
```

Explanation:

- The `Set` object lets you store unique values of any type, whether primitive values or references to objects. In this example, the `Set` constructor takes an array with duplicates and creates a set containing only unique values. Converting it back to an array with the spread operator `[...]` results in an array of unique numbers.

Question 116: Using `Map` for Key-Value Pairs

```javascript
const map = new Map();
map.set('name', 'Alice');
map.set('age', 25);
console.log(map.get('name')); // Alice
console.log(map.has('age'));  // true
```

Output:

```arduino
Alice
true
```

Explanation:

- A `Map` object holds key-value pairs where keys can be of any data type. In this example, we use `set()` to add entries to the map and `get()` to retrieve a value by key. The `has()` method checks for the existence of a key. This structure is particularly useful for maintaining the insertion order of elements.

Question 117: Using `filter()` Method

```javascript
const numbers = [1, 2, 3, 4, 5, 6];
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers);
```

Output:

```csharp
[2, 4, 6]
```

Explanation:

- The `filter()` method creates a new array with all elements that pass the test implemented by the provided function. In this example, the arrow function checks if each number is even, resulting in an array containing only the even numbers: `[2, 4, 6]`.

Question 118: Using `forEach()` Method

```javascript
const numbers = [1, 2, 3, 4, 5];
let sum = 0;
numbers.forEach(num => {
  sum += num;
});
console.log(sum);
```

Output:

```
15
```

Explanation:

- The `forEach()` method executes a provided function once for each array element. In this case, it iterates through the `numbers` array and accumulates the sum of all elements. The result is `15`, as it adds up `1 + 2 + 3 + 4 + 5`.

Question 119: Understanding the `bind()` Method

```javascript
const obj = {
  value: 42
};

function getValue() {
  return this.value;
}

const boundGetValue = getValue.bind(obj);
console.log(boundGetValue()); // 42
```

Output:

```
42
```

Explanation:

- The `bind()` method creates a new function that, when called, has its `this` keyword set to the provided value. In this example, `boundGetValue` is created by binding `getValue` to `obj`, allowing it to access `obj.value` and return `42`.

Question 120: Using `this` in Different Contexts

```javascript
const person = {
  name: 'Alice',
  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
};

const greet = person.greet;
greet(); // undefined
```

Output:

```csharp
Hello, my name is undefined
```

Explanation:

- When a method is called without an explicit object context (like in `greet()`), `this` defaults to the global object (or `undefined` in strict mode). In this case, `this.name` is `undefined` because `this` does not refer to `person`. This demonstrates how `this` is determined by how a function is called, not where it is defined.

Question 121: Using the Spread Operator with Function Calls

```javascript
const numbers = [1, 2, 3];
const maxNumber = Math.max(...numbers);
console.log(maxNumber);
```

Output:

```
3
```

Explanation:

- The spread operator `...` allows an iterable (like an array) to be expanded in places where zero or more arguments are expected. In this example, it spreads the elements of the `numbers` array into `Math.max()`, effectively finding the maximum number in the array, which is `3`.

Question 122: Using `Promise.any()`

```javascript
const promise1 = Promise.reject('Error 1');
const promise2 = new Promise((resolve) => setTimeout(resolve, 100, 'Result 2'));
const promise3 = new Promise((resolve) => setTimeout(resolve, 200, 'Result 3'));

Promise.any([promise1, promise2, promise3])
  .then(result => console.log(result));
```

Output:

```rust
Result 2
```

Explanation:

- `Promise.any()` takes an iterable of Promise objects and, as soon as one of the promises in the iterable fulfills, returns a single promise that resolves with the value from that promise. If no promises fulfill (i.e., all are rejected), it returns a promise that is rejected with an `AggregateError`, which is an error that groups together multiple errors. In this case, `promise2` fulfills first, logging `'Result 2'`.

Question 123: Using `Object.assign()` for Merging Objects

```javascript
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };
const mergedObj = Object.assign({}, obj1, obj2);
console.log(mergedObj);
```

Output:

```yaml
{ a: 1, b: 3, c: 4 }
```

Explanation:

- `Object.assign()` is used to copy the values of all enumerable own properties from one or more source objects to a target object. In this example, properties from `obj1` and `obj2` are merged into a new object. If a property exists in both source objects (like `b`), the last object's value is used, resulting in `b: 3`.

Question 124: Understanding the Prototype Chain

```javascript
function Animal(name) {
  this.name = name;
}

Animal.prototype.speak = function() {
  return `${this.name} makes a noise.`;
```

```
};

const dog = new Animal('Dog');
console.log(dog.speak());
```

Output:

```
css

Dog makes a noise.
```

Explanation:

- In JavaScript, every object has a prototype, and inheritance is achieved through the prototype chain. Here, `speak` is defined on `Animal.prototype`. When we create a `dog` instance, it inherits the `speak` method, allowing us to call `dog.speak()` and get the output.

Question 125: Using `Array.from()` to Create Arrays

```
javascript

const set = new Set(['a', 'b', 'c']);
const arr = Array.from(set);
console.log(arr);
```

Output:

```
css

['a', 'b', 'c']
```

Explanation:

- `Array.from()` creates a new Array instance from an array-like or iterable object. In this example, a `Set` is passed to `Array.from()`, converting it into an array. This method is particularly useful for transforming data structures into arrays.

Question 126: Using `Promise.race()`

```
javascript

const promise1 = new Promise((resolve) => setTimeout(resolve, 100, 'First'));
const promise2 = new Promise((resolve) => setTimeout(resolve, 50, 'Second'));

Promise.race([promise1, promise2]).then(result => console.log(result));
```

Output:

```sql
Second
```

Explanation:

- `Promise.race()` returns a promise that resolves or rejects as soon as one of the promises in the iterable resolves or rejects, with its value or reason. In this case, `promise2` resolves first, logging `'Second'`.

Question 127: Creating a Debounce Function

```javascript
function debounce(func, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => {
      func.apply(this, args);
    }, delay);
  };
}

const log = debounce(() => console.log('Debounced!'), 1000);
log();
log();
log(); // Only one log after 1 second
```

Output:

```
Debounced!
```

Explanation:

- Debouncing is a technique used to limit the rate at which a function can fire. In this example, `debounce` returns a function that resets the timeout every time it is called. The original function (`log`) will only execute once after 1 second of inactivity, even if called multiple times in quick succession.

Question 128: Using the `slice()` Method

```javascript
const arr = [1, 2, 3, 4, 5];
const slicedArr = arr.slice(1, 4);
console.log(slicedArr);
```

Output:

```
csharp
```

```
[2, 3, 4]
```

Explanation:

- The `slice()` method returns a shallow copy of a portion of an array into a new array object selected from `start` to `end` (end not included). In this example, it extracts elements from index `1` to `4`, producing a new array `[2, 3, 4]`.

Question 129: Using `includes()` for Array Searches

```
javascript
```

```javascript
const fruits = ['apple', 'banana', 'mango'];
console.log(fruits.includes('banana')); // true
console.log(fruits.includes('grape'));  // false
```

Output:

```
arduino
```

```
true
false
```

Explanation:

- The `includes()` method determines whether an array includes a certain value among its entries, returning `true` or `false`. In this case, it checks for the presence of `'banana'` and `'grape'`, returning `true` for `'banana'` and `false` for `'grape'`.

Question 130: Using `Promise.all()` with Multiple Promises

```
javascript
```

```javascript
const promise1 = Promise.resolve(5);
const promise2 = new Promise((resolve) => setTimeout(resolve, 100, 10));
const promise3 = new Promise((resolve) => setTimeout(resolve, 200, 15));

Promise.all([promise1, promise2, promise3]).then(results => {
  console.log(results);
});
```

Output:

```csharp
[5, 10, 15]
```

Explanation:

- `Promise.all()`takes an iterable of promises and returns a single promise that resolves when all of the promises have resolved, or rejects if any promise is rejected. In this example, all three promises are fulfilled, and the resulting array contains their resolved values.

Question 131: Understanding Object Destructuring

```javascript
const person = { name: 'Alice', age: 25, location: 'Wonderland' };
const { name, age } = person;
console.log(name, age);
```

Output:

```
Alice 25
```

Explanation:

- Object destructuring allows you to unpack properties from objects into distinct variables. In this example, `name`and `age`are extracted from the `person`object, allowing direct access to these properties without needing to reference the object itself.

Question 132: Creating a Simple Event Emitter

```javascript
class EventEmitter {
  constructor() {
    this.events = {};
```

```
    }

    on(event, listener) {
      if (!this.events[event]) {
        this.events[event] = [];
      }
      this.events[event].push(listener);
    }

    emit(event, data) {
      if (this.events[event]) {
        this.events[event].forEach(listener => listener(data));
      }
    }
  }

  const emitter = new EventEmitter();
  emitter.on('event', data => console.log(data));
  emitter.emit('event', 'Event triggered!');
```

Output:

```
vbnet
```

```
  Event triggered!
```

Explanation:

- This code defines a simple `EventEmitter` class that allows you to subscribe to events and trigger them. The `on` method registers a listener for a specific event, while the `emit` method calls all listeners for that event, passing any data along. Here, the event `'event'` is emitted, triggering the listener and logging `'Event triggered!'`.

Question 133: Using `find()` Method

```
javascript
```

```
  const users = [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' },
    { id: 3, name: 'Charlie' }
  ];

  const user = users.find(u => u.id === 2);
  console.log(user);
```

Output:

```
bash
```

```
  { id: 2, name: 'Bob' }
```

Explanation:

- The `find()` method returns the value of the first element in the array that satisfies the provided testing function. In this case, it finds the user object with `id` equal to `2`, returning `{ id: 2, name: 'Bob' }`. If no elements satisfy the condition, it returns `undefined`.

Question 134: Using `filter()` to Remove Duplicates

```javascript
const numbers = [1, 2, 2, 3, 4, 4, 5];
const uniqueNumbers = numbers.filter((value, index, self) => self.indexOf(value) === index);
console.log(uniqueNumbers);
```

Output:

```csharp
[1, 2, 3, 4, 5]
```

Explanation:

- This code snippet uses `filter()` to create a new array with unique values. The condition checks if the current index of the value matches the first occurrence of that value (`self.indexOf(value)`). This effectively filters out duplicates, resulting in an array of unique numbers.

Question 135: Understanding the `slice()` Method with Negative Indices

```javascript
const array = [1, 2, 3, 4, 5];
const newArray = array.slice(-3);
console.log(newArray);
```

Output:

```csharp
[3, 4, 5]
```

Explanation:

- The `slice()` method can accept negative indices, which count back from the end of the array. In this example, `slice(-3)` retrieves the last three elements of the `array`, resulting in `[3, 4, 5]`.

Question 136: Using `Object.entries()` for Iteration

```javascript
const obj = { x: 1, y: 2, z: 3 };
Object.entries(obj).forEach(([key, value]) => {
  console.log(`${key}: ${value}`);
});
```

Output:

```makefile
x: 1
y: 2
z: 3
```

Explanation:

- `Object.entries()` returns an array of a given object's own enumerable string-keyed property `[key, value]` pairs. The `forEach()` method is then used to iterate over these entries, logging each key-value pair in the format `key: value`.

Question 137: Understanding `async` Functions with Error Handling

```javascript
async function fetchData() {
  throw new Error('Data not found');
}

fetchData().catch(error => {
  console.log(error.message);
});
```

Output:

```
Data not found
```

Explanation:

- `async` functions always return a promise. If an error is thrown inside an `async` function, it returns a rejected promise. In this example, when `fetchData` is called, it throws an error which is caught in the `catch()` block, logging `'Data not found'`.

Question 138: Using `reduce()` to Flatten an Array

```javascript
const nestedArray = [[1, 2], [3, 4], [5]];
const flatArray = nestedArray.reduce((acc, curr) => acc.concat(curr), []);
console.log(flatArray);
```

Output:

```csharp
[1, 2, 3, 4, 5]
```

Explanation:

- The `reduce()` method can be used to flatten an array of arrays. In this example, it concatenates each sub-array into a single accumulator array. The initial value of the accumulator is an empty array, resulting in a flat array `[1, 2, 3, 4, 5]`.

Question 139: Using `String.prototype.split()`

```javascript
const str = 'Hello,World,JavaScript';
const result = str.split(',');
console.log(result);
```

Output:

```css
['Hello', 'World', 'JavaScript']
```

Explanation:

- The `split()` method splits a string into an array of substrings based on a specified separator. In this case, the string is split by commas, resulting in an array containing the substrings `'Hello'`, `'World'`, and `'JavaScript'`.

Here are the next 10 advanced JavaScript questions with detailed explanations, ensuring no repetitions:

Question 140: Using `Math` Methods

```javascript
const numbers = [1.5, 2.8, 3.2];
const roundedNumbers = numbers.map(Math.round);
console.log(roundedNumbers);
```

Output:

```csharp
[2, 3, 3]
```

Explanation:

- The **`Math.round()`** method rounds a number to the nearest integer. In this example, **`map()`** is used to apply **`Math.round`** to each element in the **`numbers`** array, resulting in a new array of rounded values: **`[2, 3, 3]`**.

Question 141: Understanding Closures

```javascript
function makeCounter() {
  let count = 0;
  return function() {
    count += 1;
    return count;
  };
}

const counter = makeCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

Output:

```
1
2
```

Explanation:

- A closure is a function that retains access to its lexical scope even when the function is executed outside that scope. Here, **`makeCounter`** creates a private variable **`count`**. Each call to **`counter()`**

increments and returns `count`, demonstrating how closures maintain state between function calls.

Question 142: Using `Object.keys()` to Retrieve Keys

```javascript
const obj = { a: 1, b: 2, c: 3 };
const keys = Object.keys(obj);
console.log(keys);
```

Output:

```css
['a', 'b', 'c']
```

Explanation:

- `Object.keys()` returns an array of a given object's own enumerable property names (keys). In this example, it retrieves the keys of the `obj` object, resulting in `['a', 'b', 'c']`.

Question 143: Using `String.prototype.replace()`

```javascript
const str = 'Hello, World!';
const newStr = str.replace('World', 'JavaScript');
console.log(newStr);
```

Output:

```
Hello, JavaScript!
```

Explanation:

- The `replace()` method returns a new string with some or all matches of a pattern replaced by a replacement. In this case, `'World'` is replaced with `'JavaScript'`, producing the new string `'Hello, JavaScript!'`.

Question 144: Using `Array.prototype.every()`

```javascript
const numbers = [2, 4, 6, 8];
const allEven = numbers.every(num => num % 2 === 0);
console.log(allEven);
```

Output:

```arduino
true
```

Explanation:

- The `every()` method tests whether all elements in the array pass the provided function's test. Here, it checks if all numbers are even, returning `true` since all elements in the array `[2, 4, 6, 8]` meet the condition.

Question 145: Using `Array.prototype.some()`

```javascript
const numbers = [1, 3, 5, 8];
const hasEven = numbers.some(num => num % 2 === 0);
console.log(hasEven);
```

Output:

```arduino
true
```

Explanation:

- The `some()` method tests whether at least one element in the array passes the provided function's test. In this case, it checks for even numbers. The presence of `8` makes the result `true`, indicating that at least one number satisfies the condition.

Question 146: Using `Array.prototype.reduce()` for Summation

```javascript
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((acc, curr) => acc + curr, 0);
console.log(sum);
```

Output:

```
15
```

Explanation:

- The `reduce()` method executes a reducer function (that you provide) on each element of the array, resulting in a single output value. In this case, it sums all the numbers in the array, starting from an initial value of `0`, resulting in `15`.

Question 147: Using `Array.prototype.unshift()`

```javascript
const arr = [2, 3, 4];
arr.unshift(1);
console.log(arr);
```

Output:

```csharp
[1, 2, 3, 4]
```

Explanation:

- The `unshift()` method adds one or more elements to the beginning of an array and returns the new length of the array. In this example, `1` is added to the front of `arr`, resulting in `[1, 2, 3, 4]`.

Question 148: Using `Array.prototype.shift()`

```javascript
const arr = [1, 2, 3, 4];
const firstElement = arr.shift();
console.log(firstElement);
console.log(arr);
```

Output:

```csharp
1
[2, 3, 4]
```

Explanation:

- The `shift()` method removes the first element from an array and returns that removed element. Here, `1` is removed from `arr`, and the modified array becomes `[2, 3, 4]`.

Question 149: Using `Promise.allSettled()`

javascript

```
const promise1 = Promise.resolve(1);
const promise2 = Promise.reject('Error occurred');
const promise3 = Promise.resolve(3);

Promise.allSettled([promise1, promise2, promise3]).then(results => {
  console.log(results);
});
```

Output:

lua

```
[
  { status: 'fulfilled', value: 1 },
  { status: 'rejected', reason: 'Error occurred' },
  { status: 'fulfilled', value: 3 }
]
```

Explanation:

- `Promise.allSettled()` takes an array of promises and returns a promise that resolves after all of the given promises have either resolved or rejected. The resulting array contains objects that describe the outcome of each promise, whether fulfilled or rejected, allowing you to handle multiple promises without failing if one or more rejects.

Question 150: Understanding Symbol and Unique Identifiers

javascript

```
const uniqueId = Symbol('id');
const obj = {
```

```
  [uniqueId]: 123,
};

console.log(obj[uniqueId]); // 123
console.log(Object.keys(obj)); // []
```

Output:

```
css

123
[]
```

Explanation:

- `Symbol` is a primitive data type that creates a unique and immutable identifier. In this example, a symbol `uniqueId` is used as a property key in the `obj` object. Unlike string keys, symbol keys do not appear in `Object.keys()`, highlighting their uniqueness and privacy.

Question 151: Using `Set` to Handle Unique Values

```
javascript

const set = new Set([1, 2, 2, 3, 4]);
console.log(set); // Set { 1, 2, 3, 4 }
```

Output:

```
javascript

Set { 1, 2, 3, 4 }
```

Explanation:

- `Set` is a built-in object that lets you store unique values of any type. Here, the `Set` constructor automatically removes duplicate values (like the second `2`), resulting in a collection of unique values: `1, 2, 3, 4`.

Question 152: Using `WeakMap` for Private Data

```
javascript

const privateData = new WeakMap();

class User {
  constructor(name) {
    privateData.set(this, { name });
  }

  getName() {
```

```
    return privateData.get(this).name;
  }
}

const user = new User('Alice');
console.log(user.getName());
```

Output:

```

Alice
```

Explanation:

- `WeakMap` is a collection of key-value pairs where keys are objects and values can be any value. The keys are weakly referenced, meaning they can be garbage collected if there are no other references to them. Here, `WeakMap` is used to store private data (the user's name) that is inaccessible from outside the class, demonstrating encapsulation.

Question 153: Using `String.prototype.includes()`

```javascript
const str = 'Hello, World!';
console.log(str.includes('World')); // true
console.log(str.includes('JavaScript')); // false
```

Output:

```arduino
true
false
```

Explanation:

- The `includes()` method determines whether one string can be found within another string, returning `true` or `false`. In this case, it checks for the presence of `'World'`, which returns `true`, and checks for `'JavaScript'`, which returns `false`.

Question 154: Using `Array.prototype.splice()`

```javascript
const arr = [1, 2, 3, 4, 5];
const removed = arr.splice(2, 1);
```

```
console.log(arr);    // [1, 2, 4, 5]
console.log(removed); // [3]
```

Output:

```
csharp
```

```
[1, 2, 4, 5]
[3]
```

Explanation:

- The `splice()` method changes the contents of an array by removing or replacing existing elements and/or adding new elements in place. In this example, it removes one element at index `2` (the value `3`), modifying `arr` to `[1, 2, 4, 5]` and returning the removed elements in a new array `[3]`.

Question 155: Using `Map` for Key-Value Pairs

```
javascript
```

```
const map = new Map();
map.set('a', 1);
map.set('b', 2);
console.log(map.get('a'));
console.log(map.has('b'));
console.log(map.size);
```

Output:

```
arduino
```

```
1
true
2
```

Explanation:

- `Map` is a built-in object that holds key-value pairs, where keys can be of any type. Here, `set()` is used to add key-value pairs, `get()` retrieves the value associated with a key, `has()` checks for the existence of a key, and `size` returns the number of entries in the map.

Question 156: Using `Array.prototype.concat()`

```
javascript
```

```
const arr1 = [1, 2];
const arr2 = [3, 4];
```

```
const combined = arr1.concat(arr2);
console.log(combined);
```

Output:

```
csharp
```

```
[1, 2, 3, 4]
```

Explanation:

- The `concat()` method is used to merge two or more arrays. It returns a new array containing the values from the original arrays without modifying them. In this case, it combines `arr1` and `arr2` into a new array `[1, 2, 3, 4]`.

Question 157: Using `Array.prototype.flat()`

```
javascript
```

```
const nestedArray = [1, [2, [3, 4]]];
const flatArray = nestedArray.flat(2);
console.log(flatArray);
```

Output:

```
csharp
```

```
[1, 2, 3, 4]
```

Explanation:

- The `flat()` method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth. Here, `flat(2)` flattens the `nestedArray` up to two levels deep, resulting in `[1, 2, 3, 4]`.
