

Basics

1. What are the data types in JavaScript?

- JavaScript has primitive data types such as `number`, `string`, `boolean`, `null`, `undefined`, and `symbol` (added in ES6). Objects (`object`) and functions (`function`) are also types in JavaScript.

2. Explain the difference between `undefined` and `null`.

`undefined` means a variable has been declared but has not yet been assigned a value. `null` is an assignment value that represents a deliberate non-value.

```
javascript
Copy code
let a; // undefined
let b = null; // null
```

3. How does JavaScript handle types?

- JavaScript is dynamically typed, meaning variables can hold values of any type without any type enforcement. Type coercion may occur during operations.

4. What are truthy and falsy values in JavaScript?

Truthy values are values that coerce to `true` in boolean contexts. Falsy values are values that coerce to `false`.

```
javascript
Copy code
// Falsy values
let falseValue = false;
let zero = 0;
let emptyString = '';
let notANumber = NaN;
let nullValue = null;
let undefinedValue = undefined;
```

```
// Truthy values
let trueValue = true;
let anyNonEmptyString = 'Hello';
let anyNonZeroNumber = 42;
let anyObject = {};
```

5. What is the difference between `==` and `===`?

`==` checks for equality after doing type conversion. `===` checks for equality without type conversion (strict equality).

```
javascript
Copy code
1 == '1'; // true (type coercion)
1 === '1'; // false (no type coercion)
```

6. Explain hoisting in JavaScript.

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

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

7. What is strict mode in JavaScript and how is it enabled?

Strict mode catches common coding mistakes and throws exceptions. It is enabled by placing `'use strict';` at the beginning of a script or a function.

```
javascript
Copy code
'use strict';
```

8. What are global variables and how are they declared?

Global variables are accessible from any part of the program.

They are declared without the `var`, `let`, or `const` keywords.

```
javascript
Copy code
globalVariable = 10;
```

9. Explain the difference between `var`, `let`, and `const`.

`var` declares variables scoped to the nearest function block, `let` declares variables scoped to the nearest enclosing block, and `const` declares constants scoped to the nearest enclosing block.

```
javascript
Copy code
var a = 1;
let b = 2;
const c = 3;
```

10. How do you define constants in JavaScript?

Constants are declared using the `const` keyword and must be initialized with a value that cannot be changed.

```
javascript
Copy code
const PI = 3.14;
```

Functions

11. What are the different ways to create functions in JavaScript?

Functions can be created using function declarations, function expressions, arrow functions, and methods (functions defined on objects).

```
javascript
Copy code
// Function declaration
function greet() {
  return 'Hello!';
}
```

```
// Function expression
let greet = function() {
  return 'Hello!';
};
```

```
// Arrow function
let greet = () => 'Hello!';
```

12. Explain the difference between function declaration and function expression.

Function declarations are hoisted and can be used before they are declared. Function expressions are not hoisted and cannot be used before they are defined.

```
javascript
Copy code
sayHello(); // Works
```

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

```
sayHi(); // Error: sayHi is not a function
```

```
let sayHi = function() {
  console.log('Hi!');
};
```

-
- 13. **What are arrow functions? How do they differ from regular functions?**

Arrow functions are a concise way to write function expressions in JavaScript, using `=>`. They do not have their own `this` and `arguments` bindings.

```
javascript
Copy code
let multiply = (x, y) => x * y;
```

-
- 14. **What is function hoisting? Does it work with arrow functions?**

Function hoisting moves function declarations to the top of the current scope. Arrow functions are not hoisted because they are part of expressions.

```
javascript
Copy code
sayHello(); // Works
```

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

```
sayHi(); // Error: sayHi is not a function
```

```
let sayHi = () => {
  console.log('Hi!');
};
```

-
- 15. **How do you pass arguments to a function in JavaScript?**

Arguments are passed to functions as comma-separated values inside parentheses during function invocation.

```
javascript
Copy code
function add(a, b) {
  return a + b;
}
```

```
add(3, 5); // Returns 8
```

-
- 16. **Explain function closures and how they are used.**

A closure is the combination of a function and the lexical environment within which that function was declared. It allows a function to access variables from its outer scope

even after the outer function has finished executing.

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

```
let increment = outer();
increment(); // Returns 1
increment(); // Returns 2
```

-
- 17. **What is a callback function? Provide an example of its usage.**

A callback function is a function passed into another function as an argument to be executed later. It allows asynchronous actions and is common in event handling and AJAX requests.

```
javascript
Copy code
function fetchData(callback) {
  setTimeout(() => {
    let data = 'Data fetched!';
    callback(data);
  }, 2000);
}
```

```
function displayData(data) {
  console.log(data);
}
```

```
fetchData(displayData); // Logs 'Data fetched!'
                           after 2 seconds
```

-
- Explain the concept of higher-order functions and provide an example.**

Higher-order functions are functions that take other functions as arguments or return functions as results. They enable functional programming paradigms.

```
javascript
Copy code
function operateOnArray(arr, operation) {
  let result = [];
  for (let element of arr) {
    result.push(operation(element));
  }
  return result;
}
```

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

```
// Example of higher-order function using map
let squaredNumbers = operateOnArray(numbers, x
=> x * x);
```

-
- 18. **How does `this` keyword work in JavaScript?**

`this` refers to the object to which a method belongs or the context in which a function is called. Its value is determined

by how a function is invoked.

```
javascript
Copy code
let person = {
  name: 'John',
  greet: function() {
    console.log(`Hello, ${this.name}!`);
  }
};

person.greet(); // Logs 'Hello, John!'
```

○

19. **What is the difference between call, apply, and bind methods?**

call and **apply** are used to invoke functions with a specific context (**this** value) and arguments. **bind** creates a new function with a fixed context (**this** value) and optional arguments.

```
javascript
Copy code
let person1 = { name: 'Alice' };
let person2 = { name: 'Bob' };

function greet() {
  console.log(`Hello, ${this.name}!`);
}

greet.call(person1); // Logs 'Hello, Alice!'
greet.apply(person2); // Logs 'Hello, Bob!'

let greetBob = greet.bind(person2);
greetBob(); // Logs 'Hello, Bob!'
```

○

This covers the Basics and Functions categories. Let me know if you'd like to continue with Objects and Prototypes, Arrays, or any other topic!

3.5

continue

ChatGPT

Certainly! Let's continue with Objects and Prototypes, and then we'll move on to Arrays:

Objects and Prototypes

21. How do you create objects in JavaScript?

Objects in JavaScript can be created using object literals, the **new** keyword with a constructor function, or with **Object.create()** method.

```
javascript
Copy code
// Using object literals
let person = {
  name: 'John',
  age: 30,
  greet: function() {
    console.log(`Hello, my name is ${this.name}.`);
  }
};
```

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

```
let person1 = new Person('Alice', 25);
```

```
// Using Object.create()
let obj = Object.create(null);
obj.name = 'Bob';
obj.age = 20;
```

○

22. Explain the concept of prototypal inheritance in JavaScript.

In JavaScript, objects can inherit properties and methods from other objects. Every JavaScript object has a prototype object, which acts as a template object that it inherits methods and properties from.

```
javascript
Copy code
// Example of prototypal inheritance
let animal = {
  makeSound: function() {
    console.log('Some sound');
  }
};
```

```
let cat = Object.create(animal);
cat.makeSound(); // Logs 'Some sound'
```

○

23. What is a prototype chain?

The prototype chain is a series of objects linked together by their prototypes. It is used to implement inheritance and shared properties among objects.

```
javascript
Copy code
let animal = {
  makeSound: function() {
    console.log('Some sound');
  }
};
```

```
let cat = Object.create(animal);
let kitten = Object.create(cat);

kitten.makeSound(); // Logs 'Some sound'
```

○

24. How do you add properties to an object in JavaScript?

Properties can be added to an object using dot notation or bracket notation.

```
javascript
Copy code
let person = {};
person.name = 'Alice';
person['age'] = 30;
```

-
- 25. **What are object methods? Provide an example.**

Object methods are functions defined on objects. They can access and manipulate object properties.

```
javascript
Copy code
let person = {
  name: 'John',
  greet: function() {
    console.log(`Hello, my name is
    ${this.name}.`);
  }
};

person.greet(); // Logs 'Hello, my name is
John.'
```

-
- 26. **Explain object destructuring in JavaScript.**

Object destructuring allows you to extract properties from objects and bind them to variables.

```
javascript
Copy code
let person = {
  name: 'Alice',
  age: 25
};

let { name, age } = person;
console.log(name); // Logs 'Alice'
console.log(age); // Logs 25
```

-
- 27. **How do you clone an object in JavaScript?**

Object cloning can be done using various methods like spread syntax (...), `Object.assign()`, or using libraries like Lodash.

```
javascript
Copy code
// Using spread syntax
let obj = { a: 1, b: 2 };
let clone = { ...obj };

// Using Object.assign()
let clonedObj = Object.assign({}, obj);
```

-
- 28. **What are ES6 classes? How do they differ from constructor functions?**

ES6 classes are a syntactical sugar over JavaScript's existing prototype-based inheritance. They provide a more familiar and cleaner syntax for defining classes and their relationships.

```
javascript
Copy code
// Using constructor function
function Person(name, age) {
  this.name = name;
  this.age = age;
}

// Using ES6 class
class Person {
  constructor(name, age) {
    this.name = name;
```

```
    this.age = age;
  }
}
```

-
- 29. **How do you inherit from a class in JavaScript?**

Inheritance in JavaScript can be achieved using the `extends` keyword to create a subclass (child class) from a superclass (parent class).

```
javascript
Copy code
// Superclass
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a
    sound.`);
  }
}

// Subclass
class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }

  speak() {
    console.log(`${this.name} barks.`);
  }
}
```

```
let myDog = new Dog('Buddy', 'Golden
Retriever');
myDog.speak(); // Logs 'Buddy barks.'
```

-
- 30. **Explain the difference between `hasOwnProperty` and `in` operator.**

`hasOwnProperty` is a method that checks if an object has a property defined on itself, not inherited from its prototype chain. The `in` operator checks if a property exists anywhere in the prototype chain of the object.

```
javascript
Copy code
let obj = { a: 1 };
```

```
obj.hasOwnProperty('a'); // true
'a' in obj; // true
```

○

Arrays

- 31. **How do you create an array in JavaScript?**

Arrays can be created using array literals or the `Array` constructor.

```
javascript
Copy code
// Using array literal
let fruits = ['apple', 'banana', 'cherry'];
```

```
// Using Array constructor
let numbers = new Array(1, 2, 3, 4, 5);
```

-
- 32. **Explain the various methods to add and remove elements from an array.**

Elements can be added to arrays using `push()`, `unshift()`, or `splice()`, and removed using `pop()`, `shift()`, or `splice()`.

```
javascript
Copy code
let fruits = ['apple', 'banana'];
```

```
// Adding elements
fruits.push('cherry'); // ['apple', 'banana', 'cherry']
fruits.unshift('orange'); // ['orange', 'apple', 'banana', 'cherry']
fruits.splice(2, 0, 'grape'); // ['orange', 'apple', 'grape', 'banana', 'cherry']
```

```
// Removing elements
fruits.pop(); // ['orange', 'apple', 'grape', 'banana']
fruits.shift(); // ['apple', 'grape', 'banana']
fruits.splice(1, 1); // ['apple', 'banana']
```

-
- 33. **What are array methods like `map`, `filter`, and `reduce`? Provide examples of their usage.**

`map()` creates a new array with the results of calling a provided function on every element. `filter()` creates a new array with elements that pass a test. `reduce()` applies a function to each element of the array to reduce it to a single value.

```
javascript
Copy code
let numbers = [1, 2, 3, 4, 5];
```

```
// Using map()
let doubledNumbers = numbers.map(x => x * 2);
// [2, 4, 6, 8, 10]

// Using filter()
let evenNumbers = numbers.filter(x => x % 2 === 0); // [2, 4]

// Using reduce()
let sum = numbers.reduce((acc, curr) => acc + curr, 0); // 15
```

-
- 34. **Explain the difference between `slice` and `splice` methods.**

`slice()` returns a shallow copy of a portion of an array into a new array object. `splice()` changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

```
javascript
Copy code
let fruits = ['apple', 'banana', 'cherry', 'date'];
```

```
// Using slice()
let slicedFruits = fruits.slice(1, 3); // ['banana', 'cherry']
```

```
// Using splice()
fruits.splice(2, 1, 'grape'); // ['apple', 'banana', 'grape', 'date']
```

-
- 35. **How do you iterate over an array in JavaScript?**

Arrays can be iterated using various methods such as `for` loops, `forEach()`, `map()`, `filter()`, `reduce()`, etc.

```
javascript
Copy code
let numbers = [1, 2, 3, 4, 5];
```

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

```
// Using forEach()
numbers.forEach(num => console.log(num));
```

```
// Using map()
let doubledNumbers = numbers.map(x => x * 2);
```

```
// Using for...of loop
for (let num of numbers) {
  console.log(num);
}
```

○

These explanations and examples should cover the Objects and Prototypes and Arrays sections. If you have any more questions or need further clarification, feel free to ask!

Arrays

- 36. **What are array-like objects? Provide examples.**

Array-like objects are objects that have indexed properties and a `length` property, but do not inherit from `Array.prototype`. Examples include DOM `NodeList` objects and the `arguments` object.

```
javascript
Copy code
// NodeList example (from DOM)
let nodeList = document.querySelectorAll('p');
```

```
// Arguments object example (from function parameters)
function sum() {
  console.log(arguments.length); // Logs number of arguments passed
}
```

-
- 37. **Explain the `forEach` method and how it differs from a regular `for` loop.**

`forEach` is a method on arrays that iterates over each element and executes a callback function. It provides a more concise way to iterate compared to a traditional `for` loop.

```
javascript
```

Copy code

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

```
// Using forEach
numbers.forEach(function(number) {
  console.log(number);
});

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

-
- 38. **How do you check if a variable is an array in JavaScript?**

You can check if a variable is an array using

`Array.isArray()` method or by checking its constructor.

javascript

Copy code

```
let arr = [1, 2, 3];
```

```
Array.isArray(arr); // true
```

```
// or
arr.constructor === Array; // true
```

-
- 39. **What are typed arrays in JavaScript?**

Typed arrays are array-like objects that provide a mechanism for reading and writing raw binary data in memory. They offer better performance for handling numerical data compared to regular arrays.

javascript

Copy code

```
// Example of a typed array
```

```
let buffer = new ArrayBuffer(16); // 16 bytes
```

```
let view = new Int32Array(buffer); // 4
```

```
elements, each 4 bytes (32-bit)
```

```
view[0] = 1;
```

```
view[1] = 2;
```

-
- Asynchronous JavaScript**

- 40. **What is asynchronous programming in JavaScript?**

Asynchronous programming in JavaScript allows operations to be executed asynchronously, meaning the code can continue to run without waiting for an operation to finish.

javascript

Copy code

```
// Example of asynchronous operation
```

```
(setTimeout)
```

```
console.log('Before timeout');
```

```
setTimeout(function() {
```

```
  console.log('Inside timeout');
```

```
}, 1000);
```

```
console.log('After timeout');
```

-
- 41. **Explain the event loop in JavaScript.**

The event loop is a mechanism that handles asynchronous operations in JavaScript. It continuously checks the call stack and task queue, moving tasks from the queue to the stack when the stack is empty.

javascript

Copy code

```
console.log('Start');
```

```
setTimeout(function() {
```

```
  console.log('Timeout');
```

```
}, 0);
```

```
console.log('End');
```

-
- 42. **What are promises in JavaScript? How do they work?**

Promises are objects that represent the eventual completion or failure of an asynchronous operation and its resulting value.

javascript

Copy code

```
// Example of a promise
```

```
let promise = new Promise(function(resolve,
```

```
reject) {
```

```
  setTimeout(function() {
```

```
    resolve('Success!');
```

```
  }, 1000);
```

```
});
```

```
promise.then(function(value) {
  console.log(value); // Logs 'Success!'
  after 1 second
});
```

-
- 43. **Explain `async` and `await` keywords in JavaScript.**

`async` keyword is used to define asynchronous functions.

`await` keyword is used to pause execution within `async` functions until a promise settles and returns its result.

javascript

Copy code

```
async function getData() {
```

```
  let response = await
```

```
  fetch('https://api.example.com/data');
```

```
  let data = await response.json();
```

```
  return data;
```

```
}
```

-
- 44. **What is the difference between callbacks and promises?**

Callbacks are functions passed as arguments to another function to be executed later, while promises represent the eventual result of an asynchronous operation.

javascript

Copy code

```
// Callback example
```

```
function fetchData(callback) {
```

```
  setTimeout(function() {
```

```
    callback('Data fetched');
```

```
  }, 1000);
```

```
}
```

```
fetchData(function(data) {
```

```
  console.log(data); // Logs 'Data fetched'
```

```
  after 1 second
```

```
});
```

```
// Promise example
```

```
let promise = new Promise(function(resolve,
```

```
reject) {
```

```

        setTimeout(function() {
            resolve('Data fetched');
        }, 1000);
    });

    promise.then(function(data) {
        console.log(data); // Logs 'Data fetched'
        // after 1 second
    });

```

45. How do you handle errors in asynchronous code?

Errors in promises can be handled using `.catch()` method or `try...catch` blocks with `async/await`.

```

javascript
Copy code
async function fetchData() {
    try {
        let response = await
fetch('https://api.example.com/data');
        if (!response.ok) {
            throw new Error('Failed to fetch
data');
        }
        let data = await response.json();
        return data;
    } catch (error) {
        console.error('Error:', error.message);
    }
}

```

46. Explain the concept of chaining in promises.

Chaining promises allows you to execute asynchronous operations sequentially by returning a new promise from `.then()` or `.catch()` handlers.

```

javascript
Copy code
fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('Error:',
error));

```

47. What are generator functions and iterators in JavaScript?

Generator functions are special functions that can be paused and resumed, producing a sequence of values using `yield` keyword. Iterators are objects that implement the iterator protocol, used in `for...of` loops.

```

javascript
Copy code
function* generator() {
    yield 1;
    yield 2;
    yield 3;
}

let iterator = generator();
console.log(iterator.next()); // { value: 1,
done: false }
console.log(iterator.next()); // { value: 2,
done: false }

```

```

console.log(iterator.next()); // { value: 3,
done: false }
console.log(iterator.next()); // { value:
undefined, done: true }

```

48. How do you handle multiple asynchronous operations in JavaScript?

Multiple asynchronous operations can be handled using `Promise.all()` to wait for all promises to settle, or using nested `.then()` calls for sequential operations.

```

javascript
Copy code
let promise1 =
fetch('https://api.example.com/data1');
let promise2 =
fetch('https://api.example.com/data2');

Promise.all([promise1, promise2])
    .then(responses =>
Promise.all(responses.map(response =>
response.json())))
    .then(data => console.log(data))
    .catch(error => console.error('Error:',
error));

```

•

DOM Manipulation and Events

49. How do you access and manipulate the DOM in JavaScript?

DOM can be accessed using methods like

```

getElementById(), querySelector(), and
manipulated using properties and methods like
innerHTML, setAttribute(), appendChild().
javascript
Copy code
// Accessing and manipulating DOM
document.getElementById('myElement').innerHTML
= 'Hello, World!';

```

•

50. Explain event handling in JavaScript.

Event handling involves attaching event listeners to DOM elements to respond to user interactions or other events.

```

javascript
Copy code
document.getElementById('myButton').addEvent
tListener('click', function() {
    console.log('Button clicked');
});

```

•

51. What are event bubbling and capturing?

Event bubbling is the propagation of events from the target element up through its ancestors to the root of the document. Event capturing is the opposite, where events are captured from the root down to the target element.

```

javascript
Copy code
// Event bubbling example
document.getElementById('myDiv').addEventList
ener('click', function() {
    console.log('Div clicked');
});

```



```
});
```

```
// Event capturing example
document.addEventListener('click', function() {
  console.log('Document clicked');
}, true);
```

-

52. How do you add and remove event listeners in JavaScript?

Event listeners are added using `addEventListener()` and removed using `removeEventListener()`.

```
javascript
Copy code
function handleClick() {
  console.log('Button clicked');
}
```

```
document.getElementById('myButton').addEventListener('click', handleClick);
```

```
// Remove event listener
document.getElementById('myButton').removeEventListener('click', handleClick);
```

-

53. How do you prevent default behavior in an event?

Default behavior of an event can be prevented using `event.preventDefault()` inside the event handler function.

```
javascript
Copy code
document.getElementById('myLink').addEventListener('click', function(event) {
  event.preventDefault();
  console.log('Link clicked, but default behavior prevented.');
```

```
});
```

-

54. Explain event delegation and its benefits.

Event delegation is a technique where a single event listener is attached to a parent element to manage events for its child elements. It helps to handle dynamically added elements efficiently.

```
javascript
Copy code
document.getElementById('myList').addEventListener('click', function(event) {
  if (event.target.tagName === 'LI') {
    console.log('List item clicked:', event.target.textContent);
  }
});
```

-

55. How do you create and trigger custom events in JavaScript?

- Custom events can be created using `CustomEvent`

3.5

continue

ChatGPT

Certainly! Let's continue with the remaining questions:

DOM Manipulation and Events (continued)

55. How do you create and trigger custom events in JavaScript?

Custom events can be created using `CustomEvent` constructor and triggered using `dispatchEvent()` method on DOM elements.

```
javascript
Copy code
// Creating a custom event
let customEvent = new
  CustomEvent('myCustomEvent', {
    detail: { key: 'value' }
  });
```

```
// Triggering the custom event on an element
document.getElementById('myElement').dispatchEvent(customEvent);
```

```
// Listening for the custom event
document.getElementById('myElement').addEventListener('myCustomEvent', function(event) {
  console.log('Custom event triggered with detail:', event.detail);
});
```

-

56. Explain the difference between `document.getElementById` and `document.querySelector`.

`document.getElementById` is used to select an element by its `id` attribute, returning a single element, while `document.querySelector` allows you to select elements using CSS selectors, returning the first matching element.

```
javascript
Copy code
// Example of getElementById
let elementById =
  document.getElementById('myId');
```

```
// Example of querySelector
let elementByQuery =
  document.querySelector('.myClass');
```

-

ES6+ Features

57. What are the new features introduced in ES6 (ECMAScript 2015)?

ES6 introduced significant enhancements such as arrow functions, `let` and `const` for variable declarations, classes, template literals, destructuring, promises, and more.

```
javascript
Copy code
// Example of ES6 features
const name = 'Alice';
let age = 30;
```

```
class Person {
  constructor(name, age) {
```



```

        this.name = name;
        this.age = age;
    }

    greet() {
        return `Hello, my name is ${this.name}
and I am ${this.age} years old.`;
    }
}

let alice = new Person(name, age);
console.log(alice.greet());

```

58. Explain destructuring assignment in ES6.

Destructuring assignment allows you to extract values from arrays or objects into variables using syntax that mirrors the structure of the array or object.

```

javascript
Copy code
// Example of array destructuring
let numbers = [1, 2, 3];
let [a, b, c] = numbers;
console.log(a, b, c); // Outputs: 1 2 3

// Example of object destructuring
let person = { name: 'Bob', age: 25 };
let { name, age } = person;
console.log(name, age); // Outputs: Bob 25

```

59. What are template literals in ES6?

Template literals are string literals that allow embedded expressions and multi-line strings. They are enclosed by backticks (```) instead of single or double quotes.

```

javascript
Copy code
// Example of template literals
let name = 'Charlie';
let greeting = `Hello, ${name}!`;
console.log(greeting); // Outputs: Hello,
Charlie!

```

60. How do you use `let` and `const` in ES6?

`let` and `const` are used for variable declarations. `let` allows reassignment of variable values, while `const` declares constants whose values cannot be changed once assigned.

```

javascript
Copy code
let count = 10;
count = 20; // Valid with let

const PI = 3.14;
PI = 3.14159; // Error: Assignment to constant
variable

```

61. Explain arrow functions and their advantages.

Arrow functions are a concise syntax for writing function expressions in JavaScript, with implicit return and lexical `this` binding.

javascript

```

Copy code
// Example of arrow function
const add = (a, b) => a + b;
console.log(add(3, 5)); // Outputs: 8

```

62. What are default parameters in ES6?

Default parameters allow you to initialize function parameters with default values if no argument or `undefined` is passed.

```

javascript
Copy code
// Example of default parameters
function greet(name = 'Guest') {
    console.log(`Hello, ${name}!`);
}

greet(); // Outputs: Hello, Guest!
greet('Alice'); // Outputs: Hello, Alice!

```

63. How do you use classes in ES6?

Classes in ES6 provide a cleaner syntax for creating objects and implementing inheritance using `extends` keyword.

```

javascript
Copy code
// Example of ES6 class
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(`${this.name} makes a
noise.`);
    }
}

class Dog extends Animal {
    speak() {
        console.log(`${this.name} barks.`);
    }
}

```

```

let dog = new Dog('Buddy');
dog.speak(); // Outputs: Buddy barks.

```

64. Explain the spread (`...`) operator and its uses.

The spread operator `...` allows an iterable (like arrays or strings) to be expanded into individual elements. It is useful for array manipulation, function arguments, and object literals.

```

javascript
Copy code
// Example of spread operator with arrays
let arr1 = [1, 2, 3];
let arr2 = [...arr1, 4, 5];
console.log(arr2); // Outputs: [1, 2, 3, 4, 5]

// Example of spread operator with function
arguments
function sum(a, b, c) {
    return a + b + c;
}

```

```

}
let numbers = [1, 2, 3];
console.log(sum(...numbers)); // Outputs: 6

```

65. What are modules in ES6? How do you export and import modules?

Modules in ES6 provide a way to organize code into reusable components. Modules are exported using `export` and imported using `import`.

```

javascript
Copy code
// Example of exporting and importing modules
// math.js (exporting module)
export function sum(a, b) {
  return a + b;
}

```

```

// app.js (importing module)
import { sum } from './math.js';
console.log(sum(5, 3)); // Outputs: 8

```

Error Handling and Debugging

66. How do you handle errors in JavaScript?

Errors in JavaScript can be handled using `try...catch` blocks to catch and handle exceptions.

```

javascript
Copy code
try {
  // Code that may throw an error
  throw new Error('Something went wrong!');
} catch (error) {
  console.error(error);
}

```

67. What are `try`, `catch`, and `finally` blocks?

`try` block contains code that may throw an error. `catch` block handles the error if one occurs. `finally` block contains code that executes regardless of whether an error occurred or not.

```

javascript
Copy code
try {
  // Code that may throw an error
  throw new Error('Something went wrong!');
} catch (error) {
  console.error(error);
} finally {
  console.log('Execution complete.');
```

68. How do you throw custom errors in JavaScript?

Custom errors can be thrown using `throw` keyword with an instance of `Error` or its subclasses.

```

javascript
Copy code
function validateNumber(number) {
  if (typeof number !== 'number') {

```

```

    throw new TypeError('Invalid type:
    Expected a number.');
```

```

}
return number;
}

try {
  validateNumber('abc');
} catch (error) {
  console.error(error.message);
}

```

69. What are some common debugging techniques in JavaScript?

- Common debugging techniques include using `console.log()`, `console.error()`, breakpoints in browser developer tools, `debugger` statement, and logging stack traces.

70. How do you use the JavaScript console for debugging?

The JavaScript console in browser developer tools (e.g., Chrome DevTools) is used for debugging by logging messages, evaluating expressions, and inspecting variables.

```

javascript
Copy code
console.log('Debugging message');
console.error('Error message');
```

Web APIs and Browser Environment

71. How do you use `localStorage` and `sessionStorage` in JavaScript?

`localStorage` and `sessionStorage` provide mechanisms to store key-value pairs in the browser. `localStorage` persists data across browser sessions, while `sessionStorage` persists data within the session.

```

javascript
Copy code
// Example of using localStorage
localStorage.setItem('key', 'value');
let storedValue = localStorage.getItem('key');
console.log(storedValue); // Outputs: 'value'

```

```

// Example of using sessionStorage
sessionStorage.setItem('key', 'value');
```

72. What is the difference between `localStorage` and cookies?

`localStorage` stores data locally in the browser, has larger storage capacity (5MB+), and is not sent with every HTTP request. Cookies are sent with every request, have smaller storage capacity (4KB), and can be set with expiration.

```

javascript
Copy code
// Example of setting a cookie
document.cookie = 'name=Alice; expires=Sun, 20
Jun 2024 12:00:00 UTC; path=/';

```

73. How do you make HTTP requests in JavaScript? Explain `fetch` API.

`fetch()` API is used to make HTTP requests in JavaScript, supporting modern promises and providing a more flexible and powerful interface for fetching resources from the network.

javascript
Copy code

```
// Example of using fetch API
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response
was not ok');
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('Fetch
error:', error));
```

•

74. How do you handle CORS issues in JavaScript?

Cross-Origin Resource Sharing (CORS) issues can be handled by configuring server-side headers to allow requests from specific origins (`Access-Control-Allow-Origin`) and methods (`Access-Control-Allow-Methods`).

javascript
Copy code

```
// Example of CORS-enabled fetch request
fetch('https://api.example.com/data', {
  method: 'GET',
  headers: {
    'Content-Type': 'application/json',
    'Access-Control-Allow-Origin': '*'
  }
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Fetch error:',
error));
```

•

75. Explain the difference between synchronous and asynchronous XMLHttpRequest.

XMLHttpRequest (XHR) is an older API for making HTTP requests. Synchronous requests block the execution of code until the request completes, while asynchronous requests allow the code to continue executing while waiting for the request to complete.

javascript
Copy code

```
// Example of synchronous XHR
let xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.example.com/data',
false); // false for synchronous
xhr.send(null);
console.log(xhr.responseText);

// Example of asynchronous XHR
let xhrAsync = new XMLHttpRequest();
xhrAsync.open('GET',
'https://api.example.com/data', true); //
true for asynchronous
xhrAsync.onload = function() {
  if (xhrAsync.status === 200) {
    console.log(xhrAsync.responseText);
  } else {
```

```
    console.error('Request failed.
Status:', xhrAsync.status);
  }
};
```

```
xhrAsync.send(null);
```

•

Testing and Performance

76. How do you write unit tests for JavaScript code?

Unit tests in JavaScript are typically written using testing frameworks like Jest, Mocha, or Jasmine. Tests verify the behavior of individual functions or modules.

javascript
Copy code

```
// Example of Jest test
function sum(a, b) {
  return a + b;
}
```

```
test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

•

77. What are some popular testing frameworks for JavaScript?

Popular testing frameworks for JavaScript include Jest, Mocha, Jasmine, and QUnit, each providing features for writing and executing tests.

javascript
Copy code

```
// Example of using Mocha and Chai
const assert = require('chai').assert;
```

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

```
describe('Multiply function', function() {
  it('should multiply two numbers correctly',
function() {
    assert.equal(multiply(2, 3), 6);
  });
});
```

•

78. How do you optimize JavaScript code for better performance?

JavaScript code can be optimized for performance by minimizing DOM manipulation, using efficient algorithms, caching variables, reducing function calls, and leveraging browser developer tools for profiling and optimization.

javascript
Copy code

```
// Example of optimizing loop performance
let sum = 0;
for (let i = 0; i < array.length; i++) {
  sum += array[i];
}
```

```
// Optimized version using array.reduce()
let sum = array.reduce((acc, curr) => acc +
curr, 0);
```

-
- 79. **What tools would you use to analyze and improve website performance?**
- Tools like Google Lighthouse, Chrome DevTools (Performance tab), WebPageTest, and GTmetrix are used to analyze website performance by measuring metrics such as load times, rendering speed, and resource utilization.

Security and Best Practices

-
- 80. **What are some common security issues in JavaScript applications?**
- Common security issues include XSS (Cross-Site Scripting), CSRF (Cross-Site Request Forgery), insecure data storage, improper input validation, and insecure third-party dependencies.
- 81. **How do you prevent XSS (Cross-Site Scripting) attacks in JavaScript?**
- XSS attacks can be prevented by sanitizing user input, escaping characters in output to prevent script execution, and using frameworks/libraries that handle XSS vulnerabilities.
- 82. **What is Content Security Policy (CSP) and how do you implement it?**

Content Security Policy (CSP) is a security feature that helps prevent XSS attacks by defining approved sources for content. It is implemented using HTTP headers or `<meta>` tags.

html
Copy code
`<!-- Example of CSP meta tag -->`
`<meta http-equiv="Content-Security-Policy"`
`content="default-src 'self'">`

-
- 83. **How do you sanitize user input in JavaScript?**

User input can be sanitized by removing or escaping potentially harmful characters (like `<`, `>`, `'`, `"`), validating input against expected formats, and using libraries/frameworks for input sanitization.

javascript
Copy code
`// Example of sanitizing user input`
`function sanitizeInput(input) {`
 `return input.replace(/[<>'"]/g, ''); //`
 `Remove HTML and SQL injection characters`
`}`

-
- 84. **What are some best practices for writing secure JavaScript code?**
- Best practices include validating input data, escaping output to prevent XSS attacks, using HTTPS for secure communication, avoiding `eval()`, and keeping libraries/frameworks updated to avoid known vulnerabilities.

Miscellaneous

-
- 85. **How do you create a timer in JavaScript?**

A timer in JavaScript can be created using `setTimeout()` or `setInterval()` functions to execute code after a delay or at regular intervals.

javascript
Copy code
`// Example of setTimeout`
`setTimeout(function() {`

```
  console.log('Timer executed after 1
  second.');
```

```
}, 1000);
```

```
// Example of setInterval
let count = 0;
let interval = setInterval(function() {
  console.log('Interval count:', ++count);
  if (count === 5) {
    clearInterval(interval);
  }
}, 1000);
```

-
- 86. **Explain the difference between synchronous and asynchronous code execution.**
- Synchronous code executes line by line, blocking further execution until the current operation is completed. Asynchronous code allows multiple operations to be executed concurrently, without blocking the main thread.
- 87. **How do you implement a carousel/slider in JavaScript?**

A carousel/slider can be implemented using HTML, CSS for layout, and JavaScript for handling transitions and user interactions (e.g., clicking arrows or swiping).

html
Copy code
`<!-- Example of HTML structure for carousel -->`
`<div class="carousel">`
 `<div class="carousel-inner">`
 `<div class="carousel-item">Slide`
 `1</div>`
 `<div class="carousel-item">Slide`
 `2</div>`
 `<div class="carousel-item">Slide`
 `3</div>`
`</div>`
`<button class="carousel-control`
`prev">Prev</button>`
`<button class="carousel-control`
`next">Next</button>`
`</div>`

-
- 88. **Explain the concept of lazy loading images in JavaScript.**

Lazy loading delays loading of non-visible images until they are needed, typically as the user scrolls down the page, improving initial page load performance and reducing bandwidth usage.

javascript
Copy code
`// Example of lazy loading images`
`document.addEventListener('DOMContentLoaded',`
`function() {`
 `let lazyImages =`
 `document.querySelectorAll('img.lazy');`
 `lazyImages.forEach(function(lazyImage) {`
 `lazyImage.src = lazyImage.dataset.src;`
 `});`
`});`

-
- 89. **How do you implement pagination using JavaScript?**

Pagination can be implemented by fetching a subset of data from a server/API and dynamically rendering page links or buttons to navigate through pages.

javascript

Copy code

```
// Example of pagination
const pageSize = 10; // Items per page
let currentPage = 1;

function fetchData(page) {
  // Fetch data for specified page from API
}

function renderPagination(totalPages) {
  // Render pagination buttons or links
}

// Example usage
fetchData(currentPage);
```

90. How do you detect the browser and its version using JavaScript?

Browser detection can be done using `navigator.userAgent` or feature detection (checking for specific browser APIs or behaviors).

javascript

Copy code

```
// Example of browser detection
if (navigator.userAgent.includes('Chrome')) {
  console.log('User is using Chrome browser.');
```

91. Explain the concept of functional programming in JavaScript.

Functional programming in JavaScript emphasizes the use of pure functions, immutability, and higher-order functions to perform computations by composing functions together.

javascript

Copy code

```
// Example of functional programming
const numbers = [1, 2, 3, 4, 5];

// Using map and reduce for computation
const sum = numbers.map(num => num * 2).reduce((acc, curr) => acc + curr, 0);
console.log(sum); // Outputs: 30
```

92. What are the benefits of using immutable data structures in JavaScript?

- Immutable data structures, where data cannot be changed after creation, offer several benefits:
 - **Consistency:** Immutable data ensures that once a value is set, it cannot be modified, avoiding unexpected changes.
 - **Concurrency:** Immutable data is inherently thread-safe, simplifying concurrent programming as data cannot be mutated.
 - **Predictability:** Immutability reduces bugs caused by unintended side effects, making code easier to reason about and debug.
 - **Performance:** Immutable data can optimize performance by allowing data to be shared safely across components or threads without deep copying.

93. How do you handle routing in a single-page application (SPA) using JavaScript?

Routing in SPAs is typically handled using client-side routing libraries like React Router, Vue Router, or by implementing custom routing logic using the History API.

javascript

Copy code

```
// Example of routing with React Router
import { BrowserRouter as Router, Switch, Route } from 'react-router-dom';

function App() {
  return (
    <Router>
      <Switch>
        <Route exact path="/"
          component={Home} />
        <Route path="/about"
          component={About} />
        <Route path="/products/:id"
          component={ProductDetail} />
        <Route component={NotFound} />
      </Switch>
    </Router>
  );
}
```

94. How do you implement animations in JavaScript?

Animations can be implemented in JavaScript using CSS animations/transitions or JavaScript libraries like GSAP (GreenSock Animation Platform) for more complex animations.

javascript

Copy code

```
// Example of animating with CSS transitions
const element = document.getElementById('animatedElement');
element.classList.add('animate');

// Example of animating with GSAP library
gsap.to('.animate', { duration: 1, x: 100, opacity: 0.5 });
```

95. How do you implement drag-and-drop functionality in JavaScript?

Drag-and-drop functionality can be implemented using native HTML5 Drag and Drop API or JavaScript libraries like interact.js for more advanced interactions.

javascript

Copy code

```
// Example of native HTML5 Drag and Drop API
const draggableElement = document.getElementById('draggable');
draggableElement.addEventListener('dragstart', function(event) {
  event.dataTransfer.setData('text/plain', 'Drag me!');
});

const droppableElement = document.getElementById('droppable');
```

```
draggableElement.addEventListener('dragover',
  function(event) {
    event.preventDefault();
  });
```

```
draggableElement.addEventListener('drop',
  function(event) {
    const data =
    event.dataTransfer.getData('text/plain');
    event.target.textContent = data;
  });
```

-
- 96. **Explain the concept of memoization and provide a practical example.**

Memoization is a technique used to optimize functions by caching the results of expensive function calls and returning the cached result when the same inputs occur again.

```
javascript
Copy code
// Example of memoization
function memoize(func) {
  const cache = {};
  return function(...args) {
    const key = JSON.stringify(args);
    if (!cache[key]) {
      cache[key] = func.apply(this,
args);
    }
    return cache[key];
  };
}

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

const memoizedFibonacci = memoize(fibonacci);
console.log(memoizedFibonacci(10)); // Outputs:
55 (result cached)
```

-
- 97. **How do you handle responsive design with JavaScript?**

Responsive design in JavaScript can be handled using CSS media queries for basic layout changes and JavaScript for more dynamic changes based on viewport size or device orientation.

```
javascript
Copy code
// Example of responsive design with
JavaScript
window.addEventListener('resize', function() {
  if (window.innerWidth < 768) {
    console.log('Mobile view');
  } else {
    console.log('Desktop view');
  }
});
```

-
- 98. **How do you implement internationalization (i18n) in JavaScript applications?**

Internationalization (i18n) can be implemented using libraries like `i18next`, `FormatJS`, or frameworks like `React Intl` to handle translations and locale-specific formatting.

```
javascript
Copy code
// Example of using i18next for
internationalization
import i18n from 'i18next';
import { initReactI18next } from
'react-i18next';
```

```
i18n
.use(initReactI18next)
.init({
  resources: {
    en: { translation: { welcome: 'Welcome!'
} },
    fr: { translation: { welcome:
'Bienvenue!' } }
  },
  lng: 'en',
  fallbackLng: 'en',
  interpolation: { escapeValue: false }
});
```

```
// Usage in components
function App() {
  const { t } = useTranslation();
  return <h1>{t('welcome')}</h1>;
}
```

-
- 99. **What are the differences between progressive enhancement and graceful degradation?**
- **Progressive enhancement** involves starting with a baseline of usable functionality and progressively enhancing it for browsers that support advanced features. **Graceful degradation** starts with a full-featured experience and gracefully adapts to support less capable browsers.

Progressive Enhancement Example:

```
javascript
Copy code
// Example of progressively enhanced
feature
if (document.querySelector) {

  document.querySelector('.container').classL
ist.add('enhanced');
}
```

○

Graceful Degradation Example:

```
javascript
Copy code
// Example of gracefully degrading feature
try {
  // Use modern API
} catch (error) {
  // Fallback to alternative approach
}
```

○

-
- 100. **How do you integrate third-party libraries and plugins in JavaScript applications?** - Third-party libraries and

plugins can be integrated into JavaScript applications by including their scripts or modules using `<script>` tags, `import` statements (for ES modules), or using package managers like npm or yarn.

javascript

Copy code

```
// Example of integrating a third-party
library
import { gsap } from 'gsap';

gsap.to('.element', { duration: 1, x: 100 });
```