**1. Variables (var, let, const)**

var: Function-scoped or globally scoped; can be re-declared and updated.

let: Block-scoped; can be updated but not re-declared in the same scope.

const: Block-scoped; must be initialized at declaration and cannot be updated or re-declared.

**2. Data Types (Primitive & Reference Types)**

Primitive Types: Include String, Number, Boolean, null, undefined, Symbol, and BigInt. These hold their value directly.

Reference Types: Include Object, Array, and Function. These store a reference to the value rather than the value itself.

**3. Type Coercion & Type Conversion**

Type Coercion: Automatic conversion of types (e.g., adding a number to a string).

Type Conversion: Explicit conversion (e.g., using Number() to convert a string to a number).

**4. Scope (Global, Local, Block)**

Global Scope: Accessible anywhere in the code.

Local Scope: Accessible only within a function.

Block Scope: Restricted to the block defined by {} (used with let and const).

**5. Closures**

A function that retains access to its lexical scope, even when the function is executed outside that scope. Closures can capture variables from their outer function.

**6. Hoisting**

The JavaScript engine moves declarations to the top of their containing scope. This means variables and functions can be used before they are declared. However, only the declarations are hoisted, not initializations.

**7. Functions (Function Declarations, Expressions, Arrow Functions)**

Function Declarations: Defined with the function keyword and hoisted.

Function Expressions: Defined within a variable and not hoisted.

Arrow Functions: Shorter syntax for functions; do not have their own this.

**8. IIFE (Immediately Invoked Function Expressions)**

A function that is executed immediately after its definition. Useful for creating a private scope.

javascript

Copy code

(function() {

// code here

})();

**9. Higher-Order Functions**

Functions that take other functions as arguments or return them. Commonly used in functional programming.

**10. Callback Functions**

Functions passed as arguments to other functions, executed at a later time, often used in asynchronous operations.

**11. Promises (then, catch, finally)**

Promises represent the eventual completion (or failure) of an asynchronous operation:

then(): Handles successful completion.

catch(): Handles errors.

finally(): Executes after promise is settled, regardless of outcome.

**12. Async/Await**

Syntactic sugar over promises, allowing asynchronous code to be written in a more synchronous manner. async functions return a promise, and await pauses execution until the promise resolves.

**13. Event Loop and Call Stack**

The event loop handles asynchronous operations. The call stack manages the execution context of functions. When the call stack is empty, the event loop pushes callbacks from the event queue to the stack.

**14. Error Handling (try, catch, throw)**

try: Block of code to test for errors.

catch: Block of code to handle errors.

throw: Used to create custom errors.

**15. Spread and Rest Operators (...)**

Spread Operator: Expands iterable objects (arrays, strings) into individual elements.

Rest Operator: Collects multiple elements into a single array.

**16. Destructuring (Arrays & Objects)**

A syntax for unpacking values from arrays or properties from objects into distinct variables.

javascript

Copy code

const arr = [1, 2, 3];

const [a, b] = arr;

const obj = { x: 1, y: 2 };

const { x, y } = obj;

**17. Array Methods (map, filter, reduce, forEach, find)**

map(): Creates a new array with results of calling a function on every element.

filter(): Creates a new array with elements that pass a test.

reduce(): Executes a reducer function on each element and returns a single value.

forEach(): Executes a provided function once for each array element.

find(): Returns the first element that satisfies a condition.

**18. Object Methods (Object.assign, Object.keys, Object.values, etc.)**

Object.assign(): Copies values from one or more source objects to a target object.

Object.keys(): Returns an array of a given object's property names.

Object.values(): Returns an array of a given object's property values.

**19. Prototype & Prototypal Inheritance**

JavaScript uses prototypes to enable inheritance. Each object has a prototype object, and if a property or method is not found on the object, JavaScript looks up the prototype chain.

**20. this Keyword**

Refers to the context in which a function is called. Its value depends on how a function is invoked, not where it is defined.

**21. Arrow Functions and this**

Arrow functions do not bind their own this. Instead, they inherit this from the parent scope, making them useful for preserving the context in callbacks.

**22. Classes (ES6 Classes, inheritance)**

Introduced in ES6, classes provide a clear syntax for creating objects and handling inheritance. They encapsulate properties and methods.

**23. Modules (import/export)**

JavaScript modules allow for code separation and reusability. Use export to expose functions or variables from a module and import to bring them into another module.

**24. Event Delegation**

A technique that involves attaching a single event listener to a parent element rather than multiple listeners to child elements, improving performance.

**25. Debouncing and Throttling**

Debouncing: Ensures a function is not called too often by delaying its execution until a certain time has passed since it was last called.

Throttling: Ensures a function is called at most once in a specified time interval.

**26. Closures and Lexical Environment**

Closures retain access to their outer function's scope even after the outer function has returned, allowing for private variables and encapsulation.

**27. Currying**

A technique where a function is transformed into a sequence of functions, each taking a single argument, enabling partial application.

**28. JSON (Parsing and Stringifying)**

JSON.parse(): Converts a JSON string into a JavaScript object.

JSON.stringify(): Converts a JavaScript object into a JSON string.

**29. Local Storage, Session Storage, and Cookies**

Local Storage: Stores data with no expiration, accessible across sessions.

Session Storage: Stores data for one session, not shared across tabs.

Cookies: Small pieces of data sent to and from the server; have expiration dates.

**30. DOM Manipulation (querySelector, createElement, etc.)**

Interacting with and modifying the Document Object Model (DOM):

querySelector(): Returns the first matching element.

createElement(): Creates a new HTML element.

**31. Fetch API & AJAX**

Fetch API: A modern way to make network requests, returning promises.

AJAX: Asynchronous JavaScript and XML; allows asynchronous data loading without page reloads.

**32. Timers (setTimeout, setInterval, requestAnimationFrame)**

setTimeout(): Executes a function after a specified delay.

setInterval(): Repeatedly calls a function at specified intervals.

requestAnimationFrame(): Optimizes animations by syncing updates to the browser's repaint rate.

**33. Immutable vs Mutable Data**

Immutable: Data that cannot be changed after creation (e.g., strings, frozen objects).

Mutable: Data that can be changed (e.g., arrays, regular objects).

**34. Shallow Copy vs Deep Copy**

Shallow Copy: A copy of an object that duplicates only the top-level properties. Nested objects are still references to the original.

Deep Copy: A complete copy of an object, including all nested objects.

**35. Generators and Iterators**

Generators: Functions that can be paused and resumed, using the function\* syntax and yield keyword.

Iterators: Objects that define a sequence and provide a way to access elements without exposing the underlying structure.

**36. WeakMap and WeakSet**

WeakMap: A collection of key-value pairs where keys are objects and are held weakly, meaning they can be garbage-collected if no other references exist.

WeakSet: A collection of objects where the objects are held weakly.

**37. Symbol (ES6)**

A unique and immutable primitive type, often used to create unique property keys for objects.

**38. Strict Mode**

A way to opt into a restricted variant of JavaScript, helping catch common coding errors and "unsafe" actions (e.g., using undeclared variables).

**39. Optional Chaining (?.)**

A feature that allows safe access to deeply nested properties without having to check for the existence of each reference.

**40. Nullish Coalescing (??)**

A logical operator that returns the right-hand operand when the left-hand operand is null or undefined, providing a way to set default values.