1. What are the different data types in JavaScript?

Answer:
JavaScript has 8 data types:

Primitive Types:

- String
- Number
- BigInt
- Boolean
- Undefined
- Null
- Symbol

Non-Primitive:

- Object (includes arrays, functions, etc.)

Example:

```
let name = "Alice";        // String
let age = 25;              // Number
let big = 123456789012345n; // BigInt
let isActive = true;       // Boolean
let x;                     // Undefined
let y = null;              // Null
let sym = Symbol("id");    // Symbol
let obj = { id: 1 };       // Object
```

---

2. What is the difference between == and ===?

Answer:

- == compares **value** after type coercion.
- === compares **value and type** (strict equality).

Example:

```
"5" == 5   // true (type coercion)
"5" === 5  // false (different types)
```

---

3. What is hoisting in JavaScript?

Answer:
Hoisting moves declarations (not initializations) to the top of their scope before code execution.

Example:

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

Variables declared with let and const are also hoisted but not initialized.

---

4. What are closures?

Answer:
A closure is a function that retains access to its outer scope even after the outer function has finished executing.

Example:

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    console.log(count);
  }
}

const counter = outer();
counter(); // 1
counter(); // 2
```

---

5. What is the difference between var, let, and const?

Answer:

Keyword	Scope	Hoisting	Reassignable
var	Function	Yes	Yes
let	Block	No	Yes
const	Block	No	No

Example:

```
var a = 1;
let b = 2;
```

```
const c = 3;
```

Would you like me to continue with the rest in this format (question, detailed answer, practical example)? I can provide all 50, or send them in batches (e.g., 10 at a time) for easier reading.

---

6. What is the difference between `null` and `undefined`?

Answer:

- `undefined`: A variable declared but not assigned a value.
- `null`: An assignment value that represents "no value".

Example:

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

let b = null;
console.log(b); // null
```

---

7. What are template literals?

Answer:
Template literals allow embedding expressions inside string literals using backticks `` ` `` and `${}`.

Example:

```
let name = "John";
let greeting = `Hello, ${name}!`;
console.log(greeting); // Hello, John!
```

---

8. What is the difference between `function` declaration and `function` expression?

Answer:

- Function **declaration** is hoisted.
- Function **expression** is not hoisted.

Example:

```
// Function Declaration
```

```
function greet() {
  console.log("Hello");
}

// Function Expression
const sayHi = function() {
  console.log("Hi");
};
```

---

9. What is a callback function?

Answer:
A function passed as an argument to another function to be executed later.

Example:

```
function greet(name, callback) {
  console.log("Hi " + name);
  callback();
}

function afterGreet() {
  console.log("Greeting completed.");
}

greet("Alice", afterGreet);
```

---

10. What are arrow functions?

Answer:
Arrow functions are shorter function syntax introduced in ES6. They don't bind their own `this`.

Example:

```
const add = (a, b) => a + b;
console.log(add(2, 3)); // 5
```

---

11. What is the `this` keyword in JavaScript?

Answer:
`this` refers to the object that is executing the current function.

Example:

```
const obj = {
  name: "Alice",
  greet: function() {
    console.log(this.name);
  }
};

obj.greet(); // Alice
```

---

12. What is event bubbling and event capturing?

Answer:

- **Bubbling**: Event propagates from child to parent.
- **Capturing**: Event propagates from parent to child.

You can control this using the addEventListener's third parameter (useCapture).

Example:

```
element.addEventListener('click', function() {
  console.log('clicked');
}, false); // false = bubbling
```

---

13. What is the spread operator (…)?

Answer:
It allows an iterable (like an array) to be expanded.

Example:

```
let nums = [1, 2, 3];
let moreNums = [...nums, 4, 5];
console.log(moreNums); // [1, 2, 3, 4, 5]
```

---

14. What is the rest parameter?

Answer:
It allows a function to accept an indefinite number of arguments as an array.

Example:

```
function sum(...args) {
  return args.reduce((a, b) => a + b, 0);
```

```
}

console.log(sum(1, 2, 3, 4)); // 10
```

---

15. What are higher-order functions?

Answer:
Functions that take other functions as arguments or return them.

Example:

```
function greet(message) {
  return function(name) {
    console.log(`${message}, ${name}`);
  }
}

const hello = greet("Hello");
hello("Bob"); // Hello, Bob
```

---

16. What is the difference between synchronous and asynchronous JavaScript?

Answer:

- **Synchronous**: Code is executed line-by-line. Each operation waits for the previous one to finish.
- **Asynchronous**: Allows non-blocking code execution using callbacks, promises, or async/await.

Example:

```
// Synchronous
console.log("1");
console.log("2");

// Asynchronous
console.log("1");
setTimeout(() => console.log("2"), 1000);
console.log("3");
```

---

17. What is a Promise in JavaScript?

Answer:
A **Promise** is an object that represents the eventual completion (or failure) of an asynchronous operation.

States: pending, fulfilled, rejected.

Example:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Done!"), 1000);
});

promise.then(result => console.log(result)); // Done!
```

---

18. What is the difference between map(), filter(), and reduce()?

Answer:

- map(): Transforms each element.
- filter(): Filters elements based on condition.
- reduce(): Reduces array to a single value.

Example:

```
let arr = [1, 2, 3, 4];

let mapped = arr.map(x => x * 2); // [2, 4, 6, 8]
let filtered = arr.filter(x => x % 2 === 0); // [2, 4]
let reduced = arr.reduce((acc, val) => acc + val, 0); // 10
```

---

19. What is the difference between call(), apply(), and bind()?

Answer:

- call(): Calls a function with a given this and arguments.
- apply(): Same as call(), but arguments are an array.
- bind(): Returns a new function with bound this.

Example:

```
function greet(city) {
  console.log(this.name + " from " + city);
}

let person = { name: "Alice" };
```

```
greet.call(person, "Paris");     // Alice from Paris
greet.apply(person, ["London"]); // Alice from London
let bound = greet.bind(person);
bound("Tokyo");                  // Alice from Tokyo
```

---

20. What is event delegation?

Answer:
Event delegation uses a single event listener on a parent element to handle events on child elements via event.target.

Example:

```
document.getElementById("list").addEventListener("click", function(e) {
  if (e.target.tagName === "LI") {
    console.log("Item clicked:", e.target.textContent);
  }
});
```

---

21. What are the different ways to create objects in JavaScript?

Answer:

- Using object literals
- Using new Object()
- Using constructor functions
- Using Object.create()
- Using classes (ES6)

Example:

```
let obj1 = { name: "Alice" };
let obj2 = new Object({ name: "Bob" });

function Person(name) {
  this.name = name;
}
let obj3 = new Person("Carol");

let obj4 = Object.create(obj1);

class User {
  constructor(name) {
    this.name = name;
  }
```

```
}
let obj5 = new User("Dave");
```

---

22. What is destructuring in JavaScript?

Answer:
Destructuring allows unpacking values from arrays or properties from objects.

Example:

```
// Array destructuring
let [a, b] = [1, 2];

// Object destructuring
let { name, age } = { name: "Alice", age: 25 };
```

---

23. What are default parameters?

Answer:
Function parameters can have default values if no argument is passed.

Example:

```
function greet(name = "Guest") {
  console.log("Hello " + name);
}
greet(); // Hello Guest
```

---

24. What is the `typeof` operator?

Answer:
It returns the data type of a variable.

Example:

```
console.log(typeof 123);      // number
console.log(typeof "hello");  // string
console.log(typeof undefined); // undefined
console.log(typeof null);     // object (quirk)
```

---

25. What is the difference between deep copy and shallow copy?

Answer:

- **Shallow copy**: Copies only references of nested objects.
- **Deep copy**: Recursively copies all nested values.

Example:

```
let obj1 = { a: 1, b: { c: 2 } };

// Shallow copy
let shallow = { ...obj1 };
shallow.b.c = 10;
console.log(obj1.b.c); // 10

// Deep copy
let deep = JSON.parse(JSON.stringify(obj1));
deep.b.c = 20;
console.log(obj1.b.c); // 10
```

Would you like to continue with questions **26–35**?

---

26. What is the difference between `==` and `Object.is()`?

Answer:

- `==`: Checks for equality with type coercion.
- `Object.is()`: Checks for strict equality, similar to `===`, but handles `NaN` and `-0` correctly.

Examples:

```
console.log(NaN === NaN);       // false
console.log(Object.is(NaN, NaN)); // true

console.log(+0 === -0);         // true
console.log(Object.is(+0, -0));   // false
```

---

27. What are Immediately Invoked Function Expressions (IIFE)?

Answer:
IIFE is a function that runs as soon as it's defined.

Syntax:

```
(function() {
  console.log("IIFE runs immediately");
})();
```

Use Case: Avoid polluting global scope.

---

28. What is the difference between mutable and immutable data types in JavaScript?

Answer:

- **Immutable**: Cannot be changed after creation (e.g., String, Number, Boolean).
- **Mutable**: Can be changed (e.g., Object, Array).

Example:

```
let str = "hello";
str[0] = "H";
console.log(str); // still "hello" (immutable)

let arr = [1, 2, 3];
arr[0] = 99;
console.log(arr); // [99, 2, 3] (mutable)
```

---

29. What is a pure function?

Answer:
A pure function:

- Returns the same output for the same input
- Has no side effects

Example:

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

Not Pure Example:

```
let count = 0;
function increment() {
  count++;
}
```

---

30. What are modules in JavaScript?

Answer:
Modules allow code to be split into reusable files.

Example:

```
// math.js
export function add(x, y) {
  return x + y;
}

// main.js
import { add } from './math.js';
console.log(add(2, 3)); // 5
```

---

31. What is the setTimeout() and setInterval() function?

Answer:

- setTimeout(): Executes code once after a delay.
- setInterval(): Executes code repeatedly after a delay.

Example:

```
setTimeout(() => console.log("Delayed"), 1000);

let intervalId = setInterval(() => console.log("Repeated"), 1000);
clearInterval(intervalId); // To stop it
```

---

32. What is the difference between for...in and for...of?

Answer:

- for...in: Iterates over **keys** in an object.
- for...of: Iterates over **values** of iterable objects like arrays.

Example:

```
let arr = ["a", "b", "c"];

for (let i in arr) {
  console.log(i); // 0, 1, 2
```

```
}

for (let val of arr) {
  console.log(val); // a, b, c
}
```

---

33. What is the event loop in JavaScript?

Answer:
The event loop handles asynchronous callbacks. It takes tasks from the queue and pushes them to the call stack when it's empty.

Example:

```
console.log("Start");
setTimeout(() => console.log("Timeout"), 0);
console.log("End");

// Output: Start → End → Timeout
```

---

34. What is memory leak in JavaScript?

Answer:
Memory leaks occur when objects are no longer needed but not garbage collected due to lingering references.

Example:

```
let cache = {};
function saveData(key, value) {
  cache[key] = value; // If not cleared, can lead to memory leak
}
```

---

35. What are weak references (WeakMap and WeakSet)?

Answer:

- WeakMap and WeakSet allow garbage collection if there are no other references to the object.
- They do **not prevent** objects from being collected.

Example:

```
let obj = { name: "John" };
let wm = new WeakMap();
wm.set(obj, "info");

obj = null; // `obj` can now be garbage collected
```

---

36. What is the difference between Array.slice() and Array.splice()?

Answer:

Feature	slice()	splice()
Mutates array	✗ No	✓ Yes
Return value	New array	Removed elements
Use case	To copy part of an array	To remove or replace items in-place

Example:

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

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

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

---

37. What are falsy values in JavaScript?

Answer:
Falsy values are treated as false in boolean contexts.

Falsy values:

- false
- 0
- "" (empty string)
- null
- undefined
- NaN

Example:

```
if (!0) console.log("0 is falsy"); // Output
```

---

38. What is the difference between `window`, `document`, and `this`?

Answer:

- `window`: Global object in browsers.
- `document`: DOM of the loaded web page.
- `this`: Contextual reference to the object executing the function.

Example:

```
console.log(window === this); // true (in global scope)
console.log(document.title);  // Gets the page title
```

---

39. What is debouncing in JavaScript?

Answer:
Debouncing limits the rate at which a function is executed. It ensures a function runs only **after a delay** since the last time it was invoked.

Use Case: Prevent multiple API calls on keypress.

Example:

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

---

40. What is throttling in JavaScript?

Answer:
Throttling limits a function to run at most once every X milliseconds, regardless of how many times it's triggered.

Use Case: Scroll or resize event handling.

Example:

```
function throttle(fn, limit) {
  let lastCall = 0;
  return function (...args) {
```

```
    const now = new Date().getTime();
    if (now - lastCall >= limit) {
      lastCall = now;
      fn.apply(this, args);
    }
  };
}
```

---

41. What are generators in JavaScript?

Answer:
Generators are functions that can be paused and resumed using function* and yield.

Example:

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

let g = gen();
console.log(g.next()); // { value: 1, done: false }
console.log(g.next()); // { value: 2, done: false }
```

---

42. What is optional chaining (?.) in JavaScript?

Answer:
Optional chaining prevents errors when accessing deeply nested properties.

Example:

```
let user = {};
console.log(user?.address?.street); // undefined (no error)
```

---

43. What is nullish coalescing (??) operator?

Answer:
Returns the right-hand operand only if the left-hand operand is null or undefined.

Example:

```
let value = null ?? "default";
console.log(value); // "default"
```

---

44. What is the difference between `localStorage`, `sessionStorage`, and `cookies`?

Feature	localStorage	sessionStorage	cookies
Lifetime	Until manually cleared	Until tab is closed	Set expiration
Size Limit	~5MB	~5MB	~4KB
Sent with HTTP	✗ No	✗ No	✓ Yes

---

45. What is the use of `Object.freeze()`?

Answer:
Prevents modification of an object — no new properties, no deletions, and no changes.

Example:

```
const obj = Object.freeze({ name: "Alice" });
obj.name = "Bob";
console.log(obj.name); // Alice (not changed)
```

Would you like me to continue with the **final set (46–50)**?

---

46. What is the difference between `==` and `===` in JavaScript?

Answer:

- `==` (Loose Equality): Compares values after type coercion.
- `===` (Strict Equality): Compares both value and type (no coercion).

Example:

```
console.log(5 == "5");   // true (coerces string to number)
console.log(5 === "5");  // false (different types)
```

---

47. What are template literals in JavaScript?

Answer:

Template literals allow:

- Multi-line strings
- String interpolation using ${}

Syntax:

```
const name = "Alice";
const message = `Hello, ${name}!`;
console.log(message); // Hello, Alice!
```

---

48. What is the `new` keyword in JavaScript?

Answer:

The `new` keyword:

- Creates a new empty object
- Binds `this` to that object
- Links it to a constructor's prototype
- Returns the object

Example:

```
function Person(name) {
  this.name = name;
}
const p = new Person("John");
console.log(p.name); // John
```

---

49. How does JavaScript handle errors? What is `try...catch`?

Answer:

JavaScript uses `try...catch` to handle exceptions gracefully.

Syntax:

```
try {
  // Code that may throw
  throw new Error("Oops!");
} catch (error) {
  console.error("Caught:", error.message);
}
```

Use Case: Prevent program crashes during runtime errors.

50. What are higher-order functions in JavaScript?

Answer:

Higher-order functions:

- Take other functions as arguments **or**
- Return functions

Examples:

```
function greet(name) {
  return `Hello, ${name}`;
}

function higherOrder(fn, value) {
  return fn(value);
}

console.log(higherOrder(greet, "Alice")); // Hello, Alice
```

Common higher-order functions: map, filter, reduce, forEach.