Javascript

Datatypes, Variables and Operators

How is `null` different from `undefined` in Node.js?

Answer:

- `undefined` means a variable has been declared but has not yet been assigned a value.
- `null` is an intentional absence of any object value, often used to indicate "no value."

What is the difference between `var`, `let`, and `const` in Node.js?

Answer:

Feature	`var`	`let`	`const`
Scope	Function-scoped	Block-scoped	Block-scoped
Redeclaration	Allowed	Not allowed	Not allowed
Reassignment	Allowed	Allowed	**Not allowed** (value is constant)

Example:

```
function testScope() {
  if (true) {
    var x = 1;
    let y = 2;
    const z = 3;
  }
  console.log(x); // 1
  console.log(y); // ReferenceError
  console.log(z); // ReferenceError
```

What is the difference between `==` and `===` when comparing different data types in Node.js?

Answer:

- `==` is the **loose equality operator**. It converts operands to the same type before comparing.

- `===` is the **strict equality operator**. It checks for both **value and type** without conversion.

Example:

```
console.log(5 == '5');  // true  → type coercion occurs
console.log(5 === '5'); // false → different types: number vs string
```

Is a `const` variable really immutable in Node.js?

Answer:

Not necessarily. `const` **only makes the binding immutable**, not the contents of the variable if it's an object or array.

Example:

```
const arr = [1, 2, 3];
arr.push(4); // Allowed
console.log(arr); // [1, 2, 3, 4]

arr = [5, 6]; // TypeError: Assignment to constant variable
```

So, while the **reference cannot be changed**, the **data inside** the object or array **can still be modified**.

What is lexical scope in JavaScript?

Answer:

Lexical scope means that **a function's scope is determined by its physical placement in the code**. Inner functions have access to variables declared in outer functions or global scope, but not the other way around.

Example:

```
function outer() {
  let a = 10;
  function inner() {
    console.log(a); // ✅ 10 — inner has access to outer's scope
  }
  inner();
}
outer();
```

What is hoisting? How are `var`, `let`, and `const` affected by it?

Answer:

Hoisting is JavaScript's behavior of moving **variable and function declarations to the top** of their scope **at compile time**, but not the initializations.

- `var` declarations are hoisted **and initialized as `undefined`**

- `let` and `const` are hoisted but **not initialized**, leading to a **temporal dead zone (TDZ)**

Example:

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

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

What is variable shadowing in JavaScript?

Answer:

Variable shadowing occurs when a variable declared within a certain scope (e.g., function or block) **has the same name as a variable in an outer scope**, thereby "shadowing" or overriding the outer variable **within the inner scope only**.

Example:

```
let a = 10;

function test() {
  let a = 5; // shadows the outer 'a'
  console.log(a); // ✅ 5
}

test();
console.log(a); // ✅ 10
```

What is the difference between a shallow copy and a deep copy in JavaScript?

Answer:

- **Shallow Copy** copies only the **first level** of the object. Nested objects still share references.
- **Deep Copy** recursively copies **all levels**, creating entirely independent clones.

Deep copy creates a new memory and points the new variable to it.

Shallow copy references the new variable to the memory of the variable to be copied.

Example:

```
let original = { a: 1, b: { c: 2 } };
let shallow = Object.assign({}, original);
shallow.b.c = 99;

console.log(original.b.c); //  99 (because of shared reference)
```

How do you create a deep copy of an object in Node.js?

Answer:

You can use:

- `JSON.parse(JSON.stringify(obj))` (simple but can't copy functions, `undefined`, etc.)
- `structuredClone(obj)` (modern & native)
- Deep cloning libraries like **Lodash**: `_.cloneDeep(obj)`

Example:

```
let obj = { a: 1, b: { c: 2 } };
let deep = JSON.parse(JSON.stringify(obj));
deep.b.c = 50;

console.log(obj.b.c); // 2
```

What will be the output of the following code?

```
const user = {
  name: "Alice",
  address: {
    city: "Delhi"
  }
};

const copy = { ...user };
copy.address.city = "Mumbai";

console.log(user.address.city);
```

Answer: `Mumbai`

Explanation:
The spread operator `{ ...user }` creates a **shallow copy**. The nested `address` object is still **shared by reference**, so changes to `copy.address.city` affect `user.address.city`.

What is the difference between Rest Operator and Spread Operator?

Answer:

Though both use the `...` **(three dots)** syntax, their **purpose and behavior** differ based on **context**:

Feature	Rest Operator	Spread Operator
Purpose	Collect multiple elements into one array or object	Expand or unpack elements from an array/object
Usage Context	Function parameters	Function arguments, array/object literals
Position	Always used at the **end** of parameter list	Used anywhere an expression is expected

Examples:

✅ **Rest (gathers):**

```
function sum(...nums) {
  return nums.reduce((a, b) => a + b, 0);
}
sum(1, 2, 3); // 6
```

✅ **Spread (expands):**

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

What is the spread operator and why do we use it?

Answer:

The **spread operator (`...`)** is used to **expand iterable elements** (like arrays or objects) into individual elements.

✅ **Use Cases:**

- **Clone arrays/objects** (shallow copy)
- **Merge arrays/objects**

- **Pass multiple values as arguments**

Examples:

🔹 **Clone an array:**

```
const arr = [1, 2];
const copy = [...arr]; // [1, 2]
```

🔹 **Merge objects:**

```
const obj1 = { a: 1 };
const obj2 = { b: 2 };
const merged = { ...obj1, ...obj2 }; // { a: 1, b: 2 }
```

🔹 **Function arguments:**

```
function greet(a, b, c) {
  console.log(a, b, c);
}
const args = ["Hi", "Hello", "Hey"];
greet(...args); // Hi Hello Hey
```

Functions and Closures

How does the value of `this` differ between a regular function and an arrow function?

Answer:

- In a **regular function**, `this` is **dynamic**—its value depends on how the function is called.
- In an **arrow function**, `this` is **lexically bound**—it uses `this` from its **surrounding scope**.

Example:

```
const obj = {
  value: 10,
  regular: function () {
    return this.value;
  },
  arrow: () => {
    return this.value;
  }
};

console.log(obj.regular()); // ✅ 10
console.log(obj.arrow());   // ❌ undefined (or global `this.value`)
```

What is the value of `this` in a method when it's detached and called separately?

```
const obj = {
  name: "NodeJS",
  showName() {
    return this.name;
  }
};

const detached = obj.showName;
console.log(detached());
```

Answer: `undefined` (or `global.name` if defined)

Explanation:
When the method is **detached**, it **loses its binding to `obj`**, and `this` defaults to `undefined` (in strict mode) or `global` (in non-strict mode).

What are First-Class Functions and Higher-Order Functions?

Answer:

✅ **First-Class Functions**

First class functions are those which can be treated like other variables.

- **They accept non-functions** as arguments
- **AND**
- **Return non-functions.**

```
const greet = () => console.log("Hello");
const executor = (fn) => fn();
executor(greet); // ✅ Hello
```

✅ **Higher-Order Functions (HOFs)**

Functions that:

- **Take one or more functions** as arguments,
- **OR**
- **Return a function**

Example:

```
function higherOrder(fn) {
  return function () {
    console.log("Before");
    fn();
    console.log("After");
  };
}
```

What is a callback function? State some situations where callback functions are commonly used.

Answer:

A **callback function** is a function passed as an **argument to another function**, to be **executed later**, often after an asynchronous operation completes.

Example:

```
function fetchData(callback) {
  setTimeout(() => {
    callback("Data received");
  }, 1000);
}

fetchData((data) => console.log(data)); // ✅ Data received
```

Common Use Cases:

- `setTimeout`, `setInterval`
- Event listeners: `button.addEventListener('click', callback)`
- Array methods: `map`, `filter`, `forEach`
- Asynchronous operations: `fs.readFile(callback)` in Node.js

When should one not use arrow functions?

Answer:

You should **not use arrow functions** when you need a **dynamic `this` binding**, because arrow functions **inherit `this`** from their surrounding scope and **do not have their own `this`**.

❌ **Avoid arrow functions in:**

- **Object methods**
- **Constructor functions**
- **Event handlers** (if you need to access the element via `this`)

What is function chaining? How do you create a chainable function?

Answer:

Function chaining is a technique where multiple methods are **called sequentially on the same object**, often on a single line, by returning `this` from each method.

✅ **Use Case:**

- Improves readability
- Common in libraries like jQuery, Lodash, and builder patterns

Example of a chainable object:

```
class Calculator {
  constructor(value = 0) {
    this.value = value;
  }

  add(n) {
    this.value += n;
    return this;
  }

  multiply(n) {
    this.value *= n;
    return this;
  }

  print() {
    console.log(this.value);
    return this;
  }
}

new Calculator()
  .add(5)
  .multiply(2)
  .print(); // ✅ 10
```

What is a Closure?

Answer:

A **closure** is a function that **remembers and has access to variables from its lexical scope**, even after the outer function has finished executing.

Closures are formed when:

- An **inner function** is returned or used outside its **parent function**, and
- It still retains access to the **variables defined in the outer function's scope**.

Example:

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

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

Even though `outer()` has finished running, `inner()` still has access to `count`. That's a closure.

What are the Use Cases of Closures?

Answer:

Closures are powerful tools for **encapsulation** and **stateful programming**.

✅ **Common Use Cases:**

- **Data privacy** (emulating private variables)
- **API Middleware Functions**
- **Function factories** (returning customized functions)

- **Event handlers with access to outer values**
- **Memoization/caching**
- **Callbacks and asynchronous programming**

Example:

```
// Middleware factory using closure

function authorizeRole(requiredRole) {
  return function (req, res, next) {
    // assume req.user is set by an auth middleware
    const user = req.user;

    if (user && user.role === requiredRole) {
      next();
    } else {
      res.status(403).json({ message: "Forbidden: Access denied" });
    }

  };
}
```

What all scopes does a function inside a closure have access to?

Answer:

A function inside a closure has access to:

1. **Its own local scope**
2. **The scope of its enclosing (outer) function(s)**
3. **The global scope**

Example:

```
let globalVar = "Global";

function outer() {
  let outerVar = "Outer";

  function inner() {
    let innerVar = "Inner";
    console.log(globalVar); // ✅ Global
    console.log(outerVar);  // ✅ Outer
    console.log(innerVar);  // ✅ Inner
```

```
  }
  inner();
}
outer();
```

What is Currying and Why is it Used?

Answer:

Currying is a functional programming technique where a function with **multiple arguments** is transformed into a **series of nested functions**, each taking **one argument at a time**.

Instead of:

```
f(a, b, c)
```

You write:

```
f(a)(b)(c)
```

Why is Currying Useful? (Advantages)

Benefits of Currying:

- ✅ **Function reusability** – You can create specialized versions by partially applying arguments.
- ✅ **Cleaner code** – Especially in functional pipelines or higher-order functions.
- ✅ **Avoids redundancy** – Useful when the first few arguments remain the same.
- ✅ **Improved readability** in some functional patterns.

Implement Infinite Currying to Add Multiple Numbers

Goal:
Enable syntax like this:

```
add(2)(3)(4)(5)...(); // Output: 14
```

Implementation:

```
function add(a) {
  return function inner(b) {
    if (b === undefined) {
      return a;
    }
    a += b;
    return inner;
  };
}

// Usage:
console.log(add(2)(3)(4)(5)()); // ✅ 14
```

Array Functions - Map, Reduce, Filter

What are the array methods: `map()`, `filter()`, and `reduce()`?

- `map()`: Transforms each element of an array and returns a **new array** of the same length.
- `filter()`: Filters elements based on a condition and returns a **new array** with matching elements.
- `reduce()`: Reduces an array to a **single value** (number, object, etc.) by accumulating results using a callback.

Use Cases in React:

🔹 **`map()` in React:**

Used to **render lists of components** dynamically.

```
{users.map(user => <UserCard key={user.id} user={user} />)}
```

🔹 **`filter()` in React:**

Used to **conditionally display elements** based on state/props.

```
const visibleTodos = todos.filter(todo => !todo.completed);
```

🔹 **`reduce()` in React:**

Used to **aggregate state**, like **calculating total price** in a cart.

```
const total = cart.reduce((sum, item) => sum + item.price *
item.quantity, 0);
```

What is the difference between `map()` and `forEach()`?

Feature	`map()`	`forEach()`
Returns	**New array**	**Undefined**
Chainable	✅ Yes	❌ No
Use Case	Data transformation	Side effects (e.g., logging)
Mutates Original	❌ No	❌ No

Example:

```
const nums = [1, 2, 3];
const doubled = nums.map(n => n * 2);  // [2, 4, 6]
nums.forEach(n => console.log(n * 2)); // prints 2, 4, 6
```

Does the `map()` function modify the original array?

Answer: ❌ No

`map()` does **not mutate** the original array — it returns a **new transformed array**. The original remains unchanged.

What are the parameters of the `reduce()` function?

Answer:

```
arr.reduce(callbackFn, initialValue)
```

- **callbackFn** has 4 parameters:
    1. `accumulator` – the running total/result
    2. `currentValue` – current element in the array
    3. `currentIndex` – index of current element
    4. `array` – the original array
- **initialValue** – starting value for the accumulator

Example:

```
[1, 2, 3].reduce((acc, curr, i, arr) => {
  return acc + curr;
}, 0); // 6
```

Async Programming

What is Asynchronous Program Execution?

Answer:

Asynchronous execution allows JavaScript to **perform non-blocking operations**, enabling it to continue executing code **without waiting** for long-running tasks (e.g., network requests, file I/O) to complete.

What are the Ways to Implement Asynchronous Code?

Answer:

- **Callbacks**
- **Promises**
- **Async/Await**
- **Event Listeners**
- **setTimeout/setInterval**
- **Streams and Observables (RxJS)**

What is Callback Hell and How to Solve It?

Callback Hell refers to deeply nested callbacks that make code **hard to read and maintain**.

Example:

```
login(user, () => {
  fetchProfile(() => {
    getMessages(() => {
      // and so on...
    });
```

```
  });
});
```

Solutions:

- Use **Promises**
- Use **Async/Await**
- Break logic into **named functions**

How Would You Execute Multiple Async Functions in Parallel?

Use `Promise.all()`:

```
async function runParallel() {
  const [res1, res2] = await Promise.all([fetchA(), fetchB()]);
}
```

This executes `fetchA()` and `fetchB()` **in parallel**, not sequentially.

What Happens if Any Function Throws an Error in `Promise.all()`?

Answer:
If **any promise rejects**, `Promise.all()` **immediately rejects** with that error, and **others are ignored**, even if they are still running.

```
Promise.all([Promise.resolve(1), Promise.reject("Error")])
  .catch(console.error); // ❌ Logs: "Error"
```

`Promise.all()` vs `allSettled()` vs `race()` vs `any()`

`Promise.all()`	✅ Resolves if all pass, ❌ rejects if any fail
`Promise.allSettled()`	Always resolves, returns array of `{ status, value/reason }`
`Promise.race()`	Resolves/rejects with the **first settled** promise
`Promise.any()`	✅ Resolves with the **first fulfilled**, ❌ rejects only if **all fail**

If `async/await` is asynchronous, why does execution wait at `await`?

Answer:
Although `async/await` is built on Promises and is asynchronous, **`await` pauses execution within the async function** until the Promise resolves. The long running task happens on non-primary thread and does not block the program execution.

✅ This gives asynchronous code a **synchronous-like flow**, making it easier to read.

```
async function example() {
  const data = await fetchData(); // waits here
  console.log(data); // executes after data is resolved
}
```

`setTimeout()` vs `setInterval()`

Function	Behavior
`setTimeout()`	Runs **once** after a delay
`setInterval()`	Runs **repeatedly** at intervals

Example:

```
setTimeout(() => console.log("Run once"), 1000);
setInterval(() => console.log("Repeat"), 1000);
```

Advantages of Promise over Callback

Promises	Callbacks
Avoids callback hell	Nesting makes it hard to read
Easy error handling via `.catch`	Need to manually pass errors
Chainable `.then()`	No native chaining
Clean async flow with `await`	Hard to manage async sequencing

Summary: Promises (and async/await) offer **cleaner syntax**, **easier error handling**, and **better code structure**.

Event Loop and Runtime Execution

What is the Difference Between Concurrency and Parallelism?

Feature	Concurrency	Parallelism
Definition	Multiple tasks **start, run, and complete in overlapping time**	Multiple tasks **run at the same time** (literally)
Focus	Efficient **task switching**	Efficient **simultaneous execution**
Environment	Achievable on a **single core** (via event loop)	Requires **multiple cores or threads**
Example in JS	`setTimeout`, `Promises`, `async/await` (Event Loop-based)	Not natively supported in single-threaded JS
Analogy	A single waiter **juggling tables**	Multiple waiters **serving tables simultaneously**

JavaScript Context:

JavaScript is **single-threaded**, so it supports **concurrency** using:

- Event loop
- Callback queue and microtask queue
- Asynchronous APIs (e.g., `fetch`, `setTimeout`, `fs.readFile`)

For **parallelism**, JavaScript uses:

- **Web Workers** (in browsers)
- **Worker Threads** (in Node.js)
- **Child Processes** or **Cluster module** (Node.js)

What is the Purpose of the Call Stack?

Answer:

The **Call Stack** is a data structure that keeps track of **function calls** in JavaScript. It helps the JavaScript engine:

- Track **where the code is in execution**
- Know **what to return to** after a function finishes
- Handle **nested function calls** in order

Example:

```
function a() {
  b();
}
function b() {
  console.log("Hello");
}
a(); // Call Stack: a → b → log → return → return
```

Global Execution Context vs Function Execution Context

Feature	Global Execution Context	Function Execution Context
Scope	Entire script	Specific function
Created	Once (when script starts)	Every time a function is called
`this`	Refers to global object (`window`/`global`)	Depends on how the function is called
Contains	Global variables, functions	Function's local variables, arguments

---

What are Callback Queue and Microtask Queue?

🔹 **Callback Queue (Macrotask Queue):**

- Stores **tasks like**:
    - `setTimeout()`
    - `setInterval()`
    - DOM events
    - I/O operations

🔹 **Microtask Queue:**

- Stores **faster-priority tasks** like:
    - `.then()` callbacks from Promises
    - `queueMicrotask()`
    - `MutationObserver`

What is the Purpose of the Event Loop?

Answer:

The **Event Loop** is the mechanism that:

- **Continuously checks** the Call Stack and Task Queues
- Moves tasks from **queues to the stack** when the stack is empty
- **Ensures non-blocking behavior** in JavaScript

It enables JavaScript to execute asynchronous code.

Which Queue Has Higher Priority for the Event Loop?

Answer: ✅ Microtask Queue

The Event Loop will:

1. Execute all **microtasks** (Promises, `queueMicrotask`) after the current stack is clear,
2. Then move to the **macrotasks** (like `setTimeout`).

Precedence Priority: `setTimeout()` vs `Promise.then()`

Answer:
✅ `Promise.then()` is executed **before** `setTimeout()`.

Example:

```
setTimeout(() => console.log("Timeout"), 0);
Promise.resolve().then(() => console.log("Promise"));
```

Output:

```
Promise
Timeout
```

Why?
Because the `.then()` callback is queued in the **microtask queue**, which has higher priority over `setTimeout`, which is placed in the **macrotask queue**.

What is Starvation of the Callback Queue in JavaScript?

Starvation of the callback queue occurs when **microtasks (like Promises)** keep executing continuously and prevent **macrotasks (like `setTimeout`, `setInterval`, or I/O callbacks)** in the **callback queue** from ever being executed.

Objects and Classes

How to Add Dynamic Properties to an Object?

Answer:
You can use **bracket notation** to add a property using a **variable key**:

```
const key = "name";
const obj = {};
obj[key] = "NodeJS";

console.log(obj); // { name: "NodeJS" }
```

You can also add properties using:

- **Dot notation**: `obj.prop = value` (static keys)
- **Computed property names** in object literals:

```
const key = "role";
const obj = { [key]: "admin" }; // { role: "admin" }
```

How to Iterate Through All Keys in an Object?

You can use:

🔹 **`for...in` loop:**

```
for (let key in obj) { console.log(key, obj[key]); }
```

🔹 **`Object.keys()`:**

```
Object.keys(obj).forEach(key => { console.log(key, obj[key]); });
```

🔹 **`Object.entries()`:**

```
for (let [key, value] of Object.entries(obj)) {
  console.log(key, value);
}
```

What are `JSON.stringify()` and `JSON.parse()` Methods?

Method	Purpose
`JSON.stringify()`	Converts a **JavaScript object to JSON string**
`JSON.parse()`	Converts a **JSON string to JavaScript object**

Example:

```
const obj = { name: "NodeJS" };
const str = JSON.stringify(obj); // '{"name":"NodeJS"}'

const parsed = JSON.parse(str); // { name: "NodeJS" }
```

🔸 These are commonly used for **data transfer**, **localStorage**, and **deep cloning** (with limitations).

In Variable Assignment, What Type of Copy Happens?

Type	Copy Type	Behavior
`String`	Primitive	Copied by value
`Number`	Primitive	Copied by value
`Boolean`	Primitive	Copied by value
`Object`	Reference	Copied by reference
`Array`	Reference	Copied by reference

Example:

```
let a = 10;
let b = a;
b = 20;
console.log(a); // 10 ✅ (copied by value)

let obj1 = { name: "Node" };
let obj2 = obj1;
obj2.name = "JS";
console.log(obj1.name); // "JS" ❌ (copied by reference)
```
