

JavaScript (Day1-Day6) - Interview Questions

Compiled by Kumar Nayan

? Q #1 (Basic)

Tags: General Interview , JS Fundamentals , Data Types

Question:

What are the primitive data types in JavaScript?

Answer:

The primitive data types in JavaScript are fundamental values that are not objects and have no methods. They are immutable. The seven primitive types are: `String` , `Number` , `BigInt` , `Boolean` , `Undefined` , `Null` , and `Symbol` .

? Q #2 (Intermediate)

Tags: General Interview , JS Fundamentals , Objects , Prototypes

EXTREMELY IMPORTANT

Question:

If primitive types have no methods, why can I call methods like `.toUpperCase()` on a string? Explain the underlying mechanism.

✅ Answer:

This is possible due to a process called "auto-boxing." When you try to access a method or property on a primitive value (like a string), JavaScript temporarily wraps it in a corresponding object wrapper (`new String()` , `new Number()`). It then executes the method on this temporary object and discards the wrapper, returning the resulting primitive value.

? Q #3 (🟡 Intermediate)

Tags: 🌱 General Interview , JS Fundamentals , Equality

🔥 EXTREMELY IMPORTANT

🧠 Question:

Explain the difference between `==` (loose equality) and `===` (strict equality) in JavaScript.

✅ Answer:

The strict equality operator (`===`) checks for both value and type equality without performing any type conversion. If the types are different, it returns `false` .

The loose equality operator (`==`) performs type coercion if the operands are of different types, converting them to a common type before making the comparison. This can lead to unexpected results, like `5 == "5"` being `true` .

```
jsxconsole.log(5 == "5"); // true (string "5" is coerced to number 5)
console.log(5 === "5"); // false (number and string are different types)
```

? Q #4 (🟢 Basic)

Tags: 🌱 General Interview , JS Fundamentals , Data Types

🧠 Question:

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

✅ Answer:

`undefined` typically means a variable has been declared but has not yet been assigned a value. It's the default value for uninitialized variables.

`null` is an assignment value. It is used to intentionally represent the absence of any object value. It must be explicitly assigned by a developer.

? Q #5 (🟡 Intermediate)

Tags: 🚀 MAANG-level , JS Fundamentals , Bugs , Types

🧠 Question:

What does `typeof null` return, and why?

✅ Answer:

`typeof null` returns `"object"`. This is due to a historical bug in the first version of JavaScript that was never fixed to avoid breaking existing code on the web. Despite its type being reported as an object, `null` is a primitive value.

? Q #6 (🟡 Intermediate)

Tags: 🌱 General Interview , Objects , Data Structures

🧠 Question:

What is the difference between Pass by Value and Pass by Reference in JavaScript?

✅ Answer:

JavaScript passes primitives (String, Number, etc.) by value and objects (including Arrays, Functions) by reference.

- **Pass by Value:** When a primitive is assigned to another variable, the actual value is copied. The two variables are completely independent.
- **Pass by Reference:** When an object is assigned to another variable, a reference (or pointer) to the object's memory location is copied, not the object itself. Both variables point to the exact same object, so modifying it through one variable affects the other.

```
jsx // Pass by Value (Primitives)
let a = 10;
let b = a; // Value is copied
b = 20;
console.log(a); // 10 (a is unaffected) // Pass by Reference (Objects)
let obj1 = { name: "Alice" };
let obj2 = obj1; // Reference is copied
obj2.name = "Bob";
console.log(obj1.name); // "Bob" (obj1 is affected)
```

? Q #7 (🟡 Intermediate)

Tags: 🌱 General Interview , Variables , Scope , Hoisting

🔥 EXTREMELY IMPORTANT

🧠 Question:

Compare and contrast `var`, `let`, and `const` in terms of scope, hoisting, and re-declaration.

✅ Answer:

- `var`: Has function scope or global scope. It is hoisted to the top of its scope and initialized with `undefined`. It can be re-declared and updated.
- `let`: Has block scope (`{ }`). It is hoisted but not initialized, creating a "Temporal Dead Zone" (TDZ) where it cannot be accessed before declaration. It can be updated but not re-declared in the same scope.
- `const`: Also has block scope and is in the TDZ. It cannot be updated or re-declared. It must be initialized at the time of declaration.

? Q #8 (🟡 Intermediate)

Tags: 🚀 MAANG-level , Hoisting , Scope , TDZ

🔥 EXTREMELY IMPORTANT

🧠 Question:

What is the Temporal Dead Zone (TDZ) and which variable declarations does it affect?

✅ Answer:

The Temporal Dead Zone (TDZ) is the period from the start of a block until a `let` or `const` variable's declaration is processed. During this period, the variable exists but cannot be accessed. Attempting to access it results in a `ReferenceError`. This mechanism was introduced to prevent bugs that could occur with `var` being accessible as `undefined` before its declaration.

```
jsxconsole.log(city); // ReferenceError! We are in the TDZ for 'city'.
let city = "Delhi";
console.log(city); // "Delhi" (Access is fine after declaration)
```

? Q #9 (🔴 Advanced)

Tags: 🚀 MAANG-level , Hoisting , Functions

🧠 Question:

Explain the difference in hoisting behavior between a function declaration and a function expression.

✓ **Answer:**

- **Function Declaration:** The entire function, including its name and body, is hoisted to the top of its scope. This means you can call the function before it is physically written in the code.
- **Function Expression/Arrow Function:** Only the variable name (`var` , `let` , or `const`) is hoisted. The function body itself is not assigned until the line of code is executed. If declared with `var` , trying to call it before assignment results in a `TypeError` (`variable is not a function`). If declared with `let` or `const` , it results in a `ReferenceError` due to the TDZ.

```
jsx // Function Declaration
greet(); // "Hello!" (Works because of hoisting)
function greet() {
  console.log("Hello!");
}

// Function Expression
sayHi(); // TypeError: sayHi is not a function
var sayHi = function() {
  console.log("Hi!");
};
```

? Q #10 (● Basic)

Tags: 🌱 General Interview , Objects , Property Access

🧠 **Question:**

What are the two primary ways to access properties on a JavaScript object, and when would you use each?

✓ **Answer:**


The two primary ways are Dot Notation and Bracket Notation.

1. **Dot Notation (`obj.property`):** This is the most common method. It's used when the property key is a valid JavaScript identifier (no spaces, special characters, or starting with a number).
2. **Bracket Notation (`obj['property']`):** This is used when the property key is not a valid identifier (e.g., contains spaces) or when the key is stored in a variable and needs to be evaluated dynamically.

```
jsxconst person = {
  name: "Alice",
  "home city": "New York"
};

// Dot notation
console.log(person.name); // "Alice"// Bracket notation for special characters
console.log(person["home city"]); // "New York"// Bracket notation for dynamic access
let key = "name";
console.log(person[key]); // "Alice"
```

? Q #11 (Intermediate)

Tags:  General Interview , Prototypes , Inheritance

 EXTREMELY IMPORTANT

 Question:

What is the prototype chain and how does it enable inheritance in JavaScript?

 Answer:

Every object in JavaScript has a hidden internal property, `[[Prototype]]`, which links to another object. This other object is its prototype. The prototype object has its own prototype, and so on, forming a "prototype chain." The chain ends when a prototype is `null`.

When you try to access a property on an object, JavaScript first checks the object itself. If the property isn't found, it travels up the prototype chain, checking each linked object until the property is found or the chain ends. This mechanism is how JavaScript implements inheritance, allowing objects to access methods and properties from their "ancestor" objects.

? Q #12 (Intermediate)

Tags:  MAANG-level , Objects , Iteration

 Question:

What is the difference between `Object.keys()`, `Object.values()`, and `Object.entries()` ?

 Answer:

These are built-in methods for iterating over an object's own enumerable properties:

- `Object.keys(obj)` : Returns an array of the object's own enumerable property names (keys) as strings.
 - `Object.values(obj)` : Returns an array of the object's own enumerable property values.
 - `Object.entries(obj)` : Returns an array of the object's own enumerable string-keyed property `[key, value]` pairs. Each pair is itself an array. This is very useful for iterating over both keys and values at the same time, for example with a `for...of` loop or `Map` constructor.
-

? Q #13 (● Advanced)

Tags: 🚀 MAANG-level , Objects , Iteration , Prototypes

🧠 Question:

How does a `for...in` loop differ from using `Object.keys()` for iteration?

✅ Answer:

The key difference is that a `for...in` loop iterates over all *enumerable* properties of an object, including properties inherited from its prototype chain.

In contrast, `Object.keys()` returns an array containing only the names of the object's *own* enumerable properties, ignoring any properties from the prototype chain.

? Q #14 (● Intermediate)

Tags: 🌱 General Interview , JS Fundamentals , Equality , NaN

🧠 Question:

Why does `NaN === NaN` evaluate to `false` , and how should you correctly check if a value is `NaN` ?

✅ Answer:

In JavaScript, `NaN` (Not-a-Number) is unique because it is not equal to any other value, including itself. This is a standard part of the IEEE 754 specification for floating-point numbers.

Because `NaN === NaN` is `false` , you cannot use equality operators to check for it. The correct way to check if a value is `NaN` is to use the ES6 method `Object.is()` or

the global `isNaN()` function. `Object.is(value, NaN)` is the most reliable as it correctly identifies `NaN` without the type coercion pitfalls of the global `isNaN()`.

```
jsxlet result = 1 / "hello"; // This results in NaN
```

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

? Q #15 (● Advanced)

Tags: 🚀 MAANG-level , Objects , Properties

🧠 Question:

What is the difference between `Object.getOwnPropertyNames()` and `Object.keys()` ?

✅ Answer:

Both methods return an array of an object's own properties, but they differ in one key aspect: `Object.getOwnPropertyNames()` returns an array of *all* own properties (both enumerable and non-enumerable), whereas `Object.keys()` returns an array of only the *enumerable* own properties.

? Q #16 (● Intermediate)

Tags: 🧩 General Interview , Scope

🧠 Question:

What is variable shadowing in JavaScript?

✅ Answer:

Variable shadowing occurs when a variable declared in an inner scope (like a function or block) has the same name as a variable in an outer scope. The inner variable "shadows" or hides the outer one. Any reference to that variable name within the inner scope will refer to the inner variable. Once execution leaves the inner scope, the outer variable becomes accessible again.

```
jsxlet x = 10; // Outer variable
function example() {
  let x = 20; // Inner variable shadows the outer x
  console.log(x); // 20 (accessing the inner variable)
}
example();
console.log(x); // 10 (accessing the outer variable)
```

? Q #17 (● Intermediate)

Tags: MAANG-level , Functions , this

EXTREMELY IMPORTANT

Question:

Explain the fundamental difference in how `this` is determined in a traditional function versus an arrow function.

Answer:

- **Traditional Function (`function`):** `this` is determined **dynamically** based on *how the function is called*. If called as a method (`obj.myFunc()`), `this` is the object. If called as a standalone function (`myFunc()`), `this` is the global object (`window`) or `undefined` in strict mode.
- **Arrow Function (`⇒`):** `this` is determined **lexically** based on *where the function is defined*. It inherits `this` from its parent or surrounding scope and its value is fixed at creation time. It does not have its own `this` binding.

? Q #18 (Advanced)

Tags: MAANG-level , Functions , this , Objects

Question:

Why are arrow functions generally not suitable for object methods? Provide a code example.

Answer:

Arrow functions are not suitable for object methods because they do not have their own `this` binding. They inherit `this` from the surrounding (lexical) scope. When an arrow function is defined as a property of an object literal, its `this` will refer to the global object (e.g., `window`) or `undefined` , not the object instance itself. This prevents you from accessing other properties of the object using `this` .

```
jsxconst obj = {
  name: "Bob",
  // 'this' here refers to the outer scope (e.g., window), not `obj`
  greet: () ⇒ {
    console.log(this.name);
  },
  // This works correctly because it's a traditional function
  farewell: function() {
    console.log("Goodbye, " + this.name);
  }
};
```

```
obj.greet(); // undefined (or an error in strict mode)
obj.farewell(); // "Goodbye, Bob"
```

? Q #19 (Basic)

Tags:  General Interview , Functions

 Question:

What is the `arguments` object, and is it available in all function types?

 Answer:

The `arguments` object is an array-like object that is available inside traditional `function` declarations and expressions. It contains the values of all arguments passed to that function. It is **not** available in arrow functions. In arrow functions, you must use rest parameters (`...args`) to capture all passed arguments into a true array.

? Q #20 (Advanced)

Tags:  MAANG-level , Functions , Constructors

 Question:

Can arrow functions be used as constructors with the `new` keyword? Explain why or why not.

 Answer:

No, arrow functions cannot be used as constructors. Attempting to call an arrow function with `new` will throw a `TypeError`. This is because arrow functions do not have their own `this` binding, which is essential for the `new` operator to create a new object context and assign it to `this` within the constructor. They also lack an internal `[[Construct]]` method.

? Q #21 (Intermediate)

Tags:  MAANG-level , Closures , Scope

 EXTREMELY IMPORTANT

 Question:

What is a closure in JavaScript?

 Answer:

A closure is a feature where an inner function has access to the variables and parameters of its outer (enclosing) function, even after the outer function has finished executing. The inner function "remembers" the environment (the lexical scope) in which it was created. This allows for powerful patterns like data privacy and stateful functions.

? Q #22 (Advanced)

Tags:  MAANG-level , Closures , Loops , Async

 EXTREMELY IMPORTANT

 Question:

Explain the classic closure problem when using `var` inside a `for` loop with an asynchronous function like `setTimeout` .

 Answer:

The problem is that the `setTimeout` callback function creates a closure over the loop's `i` variable. However, because `var` is function-scoped, all callbacks created in the loop refer to the *same* `i` variable. By the time the `setTimeout` callbacks execute (after the loop has completed), the value of `i` will be its final value (e.g., 3 in a loop from 0 to 2). As a result, every callback logs the same final value instead of the value from its respective iteration.

```
jsxfor (var i = 0; i < 3; i++) {  
  setTimeout(function() {  
    // All three callbacks will log 3, because the loop finishes// and i is 3 before any callback runs.  
    console.log(i);  
  }, 1000);  
}  
// Output: 3, 3, 3
```

? Q #23 (Advanced)

Tags:  MAANG-level , Closures , IIFE

 Question:

How can an Immediately Invoked Function Expression (IIFE) be used to solve the closure problem in a `for` loop?

 Answer:

An IIFE can be used to create a new scope for each loop iteration. By wrapping the `setTimeout` call in an IIFE and passing the current value of `i` as an argument,

you effectively "capture" that value in a new variable local to the IIFE. Each callback now closes over its own unique, iteration-specific variable, preserving the correct value.

```
jsxfor (var i = 0; i < 3; i++) {  
  // The IIFE creates a new scope for each iteration  
  (function(j) {  
    setTimeout(function() {  
      // This function closes over 'j', which holds the value of 'i' for this iteration  
      console.log(j);  
    }, 1000);  
  })(i); // Immediately invoke it, passing the current 'i'  
}  
// Output: 0, 1, 2
```

? Q #24 (🟡 Intermediate)

Tags: 🌱 General Interview , Closures , Memory

🧠 Question:

How can closures lead to memory leaks in JavaScript?

✅ Answer:

A memory leak can occur if a closure holds a reference to a large object (like a DOM element or a large data structure) in its parent scope, and that closure remains accessible. Because the closure "remembers" its environment, the JavaScript garbage collector cannot reclaim the memory used by that large object, even if it's no longer needed in the application. The memory usage grows unnecessarily as long as the closure's reference is maintained.

? Q #25 (🟡 Intermediate)

Tags: 🌱 General Interview , Functions , this , Call-Applied-Bind

🔥 EXTREMELY IMPORTANT

🧠 Question:

What is the purpose of the `.call()` , `.apply()` , and `.bind()` methods?

✅ Answer:

All three methods are used to explicitly set the `this` context for a function.

- `.call(thisContext, arg1, arg2, ...)` : Invokes the function immediately with a specified `this` value and arguments provided individually.

- `.apply(thisContext, [arg1, arg2, ...])` : Invokes the function immediately with a specified `this` value and arguments provided as an array.
- `.bind(thisContext, arg1, ...)` : Does **not** invoke the function immediately. It returns a *new* function where the `this` value is permanently bound to the provided context. This new function can be called later.

? Q #26 (🟡 Intermediate)

Tags: 🌱 General Interview , JS Fundamentals , Coercion , Types

🧠 Question:

What are "truthy" and "falsy" values in JavaScript? List all the falsy values.

✅ Answer:

In a boolean context, such as an `if` statement, JavaScript performs type coercion to evaluate whether a value is `true` or `false`. A "truthy" value is any value that coerces to `true`. A "falsy" value coerces to `false`.

There are exactly six falsy values in JavaScript:

1. `false`
2. `0` (zero)
3. `""` (empty string)
4. `null`
5. `undefined`
6. `NaN` (Not-a-Number)

Every other value, including objects `{}` and arrays `[]`, is truthy.

? Q #27 (🟡 Intermediate)

Tags: 🌱 General Interview , JS Fundamentals , Coercion

🧠 Question:

Explain how JavaScript's type coercion differs when using the `+` operator versus other arithmetic operators like `-` or `*`.

✅ Answer:

The `+` operator is unique because it serves for both numeric addition and string concatenation. If either operand is a string, JavaScript defaults to string coercion, converting the other operand to a string and concatenating them.

In contrast, other arithmetic operators (`-`, `*`, `/`) always perform numeric coercion. They will attempt to convert any non-numeric operands into numbers before performing the calculation.

```
jsx // String coercion with '+'
let r1 = 5 + "10"; // "510" (5 is converted to a string)// Numeric coercion with '-'
let r2 = "10" - 5; // 5 (string "10" is converted to a number)
```

? Q #28 (🟡 Intermediate)

Tags: 🚀 MAANG-level , JS Fundamentals , Objects , Types

🧠 Question:

Differentiate between manual object wrapping (e.g., `new String("text")`) and automatic object wrapping (auto-boxing).

✅ Answer:

- **Auto-boxing** is a temporary, behind-the-scenes process. When you access a method on a primitive (e.g., `'hello'.toUpperCase()`), JavaScript temporarily creates a wrapper object (`new String('hello')`), calls the method, and then discards the wrapper. The original value remains a primitive.
- **Manual Wrapping** is an explicit action by the developer using a constructor like `new String()`, `new Number()`, or `new Boolean()`. This creates a permanent object that *wraps* the primitive value. This is generally discouraged because it can lead to confusion, as the created object will not behave like a primitive (e.g., `typeof new String('hi')` is `"object"`, and `new Boolean(false)` is `truthy`).

? Q #29 (🟡 Intermediate)

Tags: 🌱 General Interview , Objects , Properties , Prototypes

🧠 Question:

What is the purpose of the `Object.hasOwnProperty()` method?

✅ Answer:

`Object.hasOwnProperty()` is a method that returns a boolean indicating whether an object has a specified property as its own direct property. It returns `true` if the property exists directly on the object and `false` if the property does not exist or

is inherited from the object's prototype chain. This is crucial for distinguishing between an object's own properties and inherited ones, especially when iterating with a `for...in` loop.

```
jsxconst person = { name: "Alice" };
Object.prototype.age = 30;

console.log(person.hasOwnProperty('name')); // true (own property)
console.log(person.hasOwnProperty('age')); // false (inherited property)
```

? Q #30 (Advanced)

Tags:  MAANG-level , Prototypes , Inheritance

 EXTREMELY IMPORTANT

 Question:

Explain the critical difference between an object's `__proto__` property and a constructor function's `prototype` property.


 Answer:

This is a fundamental concept of JavaScript's inheritance model:

- `Function.prototype` : This is a property that exists on every **constructor function**. It is an object that will become the prototype for all instances created by that constructor using the `new` keyword. You place shared methods and properties on this object to make them available to all instances.
- `object.__proto__` : This is an internal property on every **object instance**. It is a direct link to the prototype object that the instance inherits from (i.e., the constructor's `prototype` object). It's the mechanism that allows the JavaScript engine to walk up the prototype chain.

In short: `Constructor.prototype` is the blueprint; `instance.__proto__` is the link to that blueprint.

? Q #31 (Intermediate)

Tags:  General Interview , Prototypes , Objects

 Question:

How does `Object.create()` work and how is it used for setting up prototypal inheritance?

✅ Answer:

`Object.create()` is a method that creates a new object, using an existing object as the prototype of the newly created object. It provides a direct and clean way to implement prototypal inheritance without needing constructor functions. The first argument to `Object.create()` becomes the `[[Prototype]]` (or `__proto__`) of the new object.

```
jsx // Base object (prototype)
const personProto = {
  greet: function() {
    console.log(`Hello, I'm ${this.name}`);
  }
};

// Create a new object that inherits directly from personProto
const alice = Object.create(personProto);
alice.name = "Alice";

alice.greet(); // "Hello, I'm Alice" (greet is inherited)
```

? Q #32 (🟡 Advanced)

Tags: 🚀 MAANG-level , Prototypes , Bugs , Architecture

🧠 Question:

Explain "Shared Mutable State" via prototypes and provide an example of a potential bug it can cause.

✅ Answer:

Shared Mutable State occurs when multiple object instances inherit a property from their prototype that is a mutable object (like an array or object). If one instance modifies this shared object, the change is visible across all other instances that share the same prototype. This can lead to unexpected side effects and bugs.

The problem arises because the instances don't get their own copy of the property; they all share a single reference to the same object on the prototype.

```
jsxconst familyProto = {
  shoppingList: [] // Mutable array on the prototype
};

const person1 = Object.create(familyProto);
const person2 = Object.create(familyProto);

// Person 1 adds an item to what they think is their list
person1.shoppingList.push("Milk");
```



```
// But it modifies the shared prototype's list  
console.log(person2.shoppingList); // ["Milk"] - Unexpected!
```

? Q #33 (Intermediate)

Tags:  General Interview , Classes , Prototypes

 EXTREMELY IMPORTANT

 Question:

Is JavaScript a class-based language? Explain the role of the `class` keyword introduced in ES6.

 Answer:

No, JavaScript is fundamentally a **prototype-based** language. The `class` keyword, introduced in ES6, is primarily **syntactic sugar** over JavaScript's existing prototypal inheritance mechanism. It provides a cleaner, more familiar syntax for developers coming from class-based languages like Java or C++, but it does not change the underlying model. A `class` definition still creates a constructor function and sets up the `.prototype` property for shared methods.

? Q #34 (Advanced)

Tags:  MAANG-level , this , Scope , Coercion

 Question:

What is "this substitution" in JavaScript and under what conditions does it occur?

 Answer:

"This substitution" is a behavior in non-strict mode where, if the `this` value of a function call is `null` or `undefined`, it is automatically replaced with the global object (`window` in browsers). This happens when a regular function is called without a specific context (e.g., `myFunc()` instead of `obj.myFunc()`). In strict mode (`"use strict";`), this substitution does not happen, and `this` remains `undefined`, which helps prevent bugs.

? Q #35 (Intermediate)

Tags:  MAANG-level , this , Scope

 Question:

How does `this` behave differently in strict mode vs. non-strict mode when a function is called in the global scope?

✅ Answer:

The difference is significant:

- **Non-Strict Mode:** When a regular function is called globally (e.g., `myFunction()`), `this` defaults to the global object (`window` in a browser). This is due to "this substitution."
- **Strict Mode ("use strict";):** When the same function is called globally, `this` will be `undefined`. This is a safety feature to prevent accidental modification of the global object.

```
function showThis() {  
  console.log(this);  
}  
showThis(); // In non-strict mode: Window object. In strict mode: undefined.
```

? Q #36 (🟢 Basic)

Tags: 🌱 General Interview , this , DOM

🧠 Question:

In an HTML file, what does `this` refer to inside an inline event handler attribute like `onclick` ?

✅ Answer:

Inside an inline event handler in HTML, `this` refers to the HTML element that the event handler is attached to. This allows for direct manipulation of the element that triggered the event.

```
xml <!-- `this` here refers to the button element itself →  
<button onclick="this.style.backgroundColor='red'">Click me</button>
```

? Q #37 (🟡 Intermediate)

Tags: 🌱 General Interview , Functions , this , Call-Apply-Bind

🧠 Question:

What is the practical difference between the `.call()` and `.apply()` methods?

✅ Answer:

Both `.call()` and `.apply()` invoke a function immediately with a specified `this` context. The only difference is how they accept arguments for the function being called:

- `.call(thisContext, arg1, arg2, ...)` : Accepts arguments as a comma-separated list.
- `.apply(thisContext, [arg1, arg2, ...])` : Accepts arguments as a single array.

`.apply()` is particularly useful when the arguments are already in an array or when the number of arguments is dynamic.

? Q #38 (Advanced)

Tags:  MAANG-level , `this` , Constructors , Call-Apply-Bind

 EXTREMELY IMPORTANT

 Question:

Explain the binding priority: What happens to `this` when a function created with `.bind()` is called with the `new` operator?

 Answer:

The `new` operator has a higher binding priority than `.bind()`. When you use `new` on a function that was previously hard-bound using `.bind()`, the `this` context provided to `.bind()` is ignored. Instead, `this` will be a newly created object, as is standard for constructor calls. This allows a bound function to still be used as a constructor, ensuring `new` always results in a new instance.


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

const boundPerson = Person.bind({ name: 'IGNORED' });

// 'new' overrides the bound 'this' context.
const newPerson = new boundPerson('Charlie');

// The new instance's name is 'Charlie', not 'IGNORED'.
console.log(newPerson.name); // 'Charlie'
```

? Q #39 (Intermediate)

Tags:  General Interview , Scope , Closures

 Question:

What is lexical scoping and how does it relate to closures?

✓ Answer:

Lexical scoping (or static scoping) means that a variable's scope is determined by its position in the source code at the time it was written, not by where the function is called. An inner function has access to the variables of its outer functions simply because of its physical placement in the code.

This principle is the foundation of closures. A closure is created when an inner function "remembers" its lexical scope (the variables from its parent functions) even after the parent function has finished executing. The inner function maintains a reference to this remembered scope.

? Q #40 (🟡 Intermediate)

Tags: 🌱 General Interview , Functions , Patterns , IIFE

🧠 Question:

What is an IIFE (Immediately Invoked Function Expression), and what are its primary use cases?

✓ Answer:

An IIFE is a JavaScript function that is defined and executed immediately. It is typically anonymous and is not assigned to a variable, so it runs only once.

The primary use cases are:

1. **Creating a Private Scope:** To avoid polluting the global scope with variables. Any `var`, `let`, or `const` declared inside an IIFE is not accessible from the outside.
2. **Solving Closure Issues in Loops:** Before `let` was introduced, an IIFE was used to capture the value of a loop variable for each iteration in asynchronous operations.

```
jsx // Example of creating a private scope
(function() {
  var privateVar = "I am private";
  console.log(privateVar); // Works
})();

// console.log(privateVar); // ReferenceError: privateVar is not defined
```

? Q #41 (🟡 Intermediate)

Tags:  General Interview ,  Functions ,  this

 **Question:**





When should you choose a traditional function over an arrow function, and vice versa?

 **Answer:**

The choice depends primarily on how `this` should be handled:

- **Use a traditional `function` when:**
 1. You need a method within an object that should refer to the object itself (`this`).
 2. You are creating a constructor function that will be called with `new`.
 3. You need the `arguments` object.
- **Use an `⇒` arrow function when:**
 1. You need a short, concise function, especially for callbacks in methods like `.map()` or `.filter()`.
 2. You need to preserve the `this` from the surrounding (lexical) context, such as inside a `setTimeout` or event listener within an object method.

? Q #42 (Advanced)

Tags:  MAANG-level ,  Objects ,  Iteration ,  Prototypes

 **Question:**

Explain how a `for...in` loop works and why it can be risky to use without safeguards.

 **Answer:**

A `for...in` loop iterates over the keys (property names) of an object. The risk is that it traverses not only the object's *own* properties but also any *enumerable* properties it inherits from its prototype chain. This can lead to unexpected behavior if you unintentionally access or modify inherited properties.

To use it safely, you should always include a check with `Object.hasOwnProperty()` inside the loop to ensure you are only processing the object's own properties.

```
jsxObject.prototype.inheritedProp = 'inherited';  
const myObj = { ownProp: 'own' };
```

```
for (const key in myObj) {  
  if (myObj.hasOwnProperty(key)) {  
    console.log(`Own property: ${key}`); // Logs "Own property: ownProp"  
  } else {  
    console.log(`Inherited property: ${key}`); // Logs "Inherited property: inheritedProp"  
  }  
}
```

? Q #43 (● Basic)

Tags: ✖ General Interview , Prototypes

 Question:

What is at the end of every prototype chain in JavaScript?

✓ Answer:

The end of every prototype chain is `null`. When the JavaScript engine walks up the chain looking for a property and reaches an object whose prototype is `null`, the search stops. `Object.prototype` is the final object in most chains, and its own prototype is `null`.

? Q #44 (● Advanced)

Tags: ✖ MAANG-level , Objects , Prototypes

 Question:

What is the difference between creating an object with `{}` versus `Object.create(null)` ?

✓ Answer:

- `{}` (Object Literal Syntax): This creates a new object that inherits from `Object.prototype`. This means it automatically has access to standard object methods like `.toString()`, `.hasOwnProperty()`, etc.
 - `Object.create(null)`: This creates a "pure" or "dictionary" object that does **not** inherit from anything. Its prototype is `null`. This object is completely empty and will not have any of the standard built-in object methods. This can be useful for creating map-like objects where you don't want to worry about potential key collisions with properties from `Object.prototype`.
-

? Q #45 (● Intermediate)

Tags: ✖ MAANG-level , this , Async , Call-ApPLY-Bind

🧠 Question:

An object method uses `setTimeout`. How do you ensure `this` inside the callback correctly refers to the object instance? Show a solution using `.bind()`.

✅ Answer:

When a regular function is used as a `setTimeout` callback, its `this` context is lost and defaults to the global object (`window`) or `undefined`. To fix this, you can use `.bind(this)` on the callback function. `.bind(this)` creates a new function where `this` is permanently bound to the context of the outer method (the object instance).

```
jsxconst pet = {
  name: "Fluffy",
  bark: function() {
    // 'this' here refers to the 'pet' object
    const callback = function() {
      // Without .bind(), `this.name` would be undefined here
      console.log(this.name + " says woof!");
    };
    // We bind the callback to the current `this` context (the pet object)
    setTimeout(callback.bind(this), 1000);
  }
};

pet.bark(); // After 1 second, correctly logs: "Fluffy says woof!"
```

? Q #46 (🟡 Expert)

Tags: 🚀 MAANG-level , Objects , Types , Prototypes

🧠 Question:

How would you write a function to robustly check if a value is a "plain" JavaScript object (created by `{}` or `new Object()`) and not an array, date, or instance of a custom class?

✅ Answer:

A robust check involves several steps:

1. Rule out primitives and `null` by checking `typeof value === 'object' && value !== null`.
2. Get the object's direct prototype using `Object.getPrototypeOf(value)`.
3. If the prototype is `null`, it's a plain object (created with `Object.create(null)`).
4. If it has a prototype, check if that prototype is the standard `Object.prototype`. This can be done by inspecting the prototype's `constructor` property. The constructor should be the `Object` function itself. Comparing the string

representation of the constructor (`Function.prototype.toString.call(Ctor)`) with that of the global `Object` constructor confirms it's not a subclass.

This multi-step validation ensures you correctly identify only plain objects and exclude specialized object types.

? Q #47 (● Basic)

Tags: ✖ General Interview , Objects , ES6

 Question:

What is object destructuring and how does it simplify accessing object properties?

✔ Answer:

Object destructuring is an ES6 feature that provides a concise way to extract properties from an object and assign them to variables. Instead of accessing each property individually using dot or bracket notation, you can declare the variables you want to create using a syntax that mirrors the object literal. This makes the code cleaner and easier to read.

```
jsxconst user = {
  id: 42,
  username: "alice_dev",
  email: "alice@example.com"
};

// Without destructuring// const username = user.username;// const email = user.email;// With destructuring
const { username, email } = user;

console.log(username); // "alice_dev"
console.log(email);    // "alice@example.com"
```

? Q #48 (● Advanced)

Tags: 🚀 MAANG-level , JS Fundamentals , Coercion , Equality

 Question:

Explain why `[] == ![]` evaluates to `true` in JavaScript.

✔ Answer:

This is a classic example of how JavaScript's type coercion and equality rules interact. The evaluation happens in these steps:

1. **![] (Logical NOT):** The `!` operator coerces its operand to a boolean and then negates it. An array `[]` is a "truthy" value. So, `![]` becomes `!true`, which evaluates to `false`.
 2. `[] == false`: The expression is now `[] == false`. The loose equality (`==`) algorithm sees an object (`[]`) and a boolean (`false`). It converts both to numbers.
 3. **ToNumber Conversion:** The boolean `false` is converted to the number `0`. The empty array `[]` is first converted to an empty string `""` (`[]`.toString()), and then the empty string `""` is converted to the number `0`.
 4. `0 == 0`: The expression becomes `0 == 0`, which is `true`.
-

? Q #49 (🟡 Intermediate)

Tags: 🟢 General Interview , Objects , Property Access

🧠 **Question:**

What is the best practice for accessing an object's prototype today, `__proto__` or `Object.getPrototypeOf()` ? Explain why.

✅ **Answer:**

The best practice is to use `Object.getPrototypeOf()`.

The `__proto__` property was a non-standard, de-facto way to access an object's prototype, introduced by some browsers. While it is now standardized in modern JavaScript for compatibility, it's considered legacy.

`Object.getPrototypeOf(obj)` is the official, standard ES5 method. It is safer because it's a read-only way to get the prototype, preventing accidental modification of an object's inheritance chain, which can be a source of complex bugs.

? Q #50 (🟡 Intermediate)

Tags: 🟢 General Interview , JS Fundamentals , Objects , Destructuring

🧠 **Question:**

What is the difference between `Object.keys()`, `Object.values()`, and `Object.entries()` ?

✅ **Answer:**

These methods provide different ways to extract an object's own enumerable properties into an array:

- `Object.keys(obj)` : Returns an array of the object's property **keys** (names) as strings.
- `Object.values(obj)` : Returns an array of the object's property **values**.
- `Object.entries(obj)` : Returns an array of `[key, value]` pairs. Each element in the returned array is itself a two-element array. This is very useful for iterating over both the key and value simultaneously, for example with a `for...of` loop.

```
jsxconst user = { name: 'Bob', age: 40 };
```

```
console.log(Object.keys(user)); // ['name', 'age']
console.log(Object.values(user)); // ['Bob', 40]
console.log(Object.entries(user)); // [['name', 'Bob'], ['age', 40]]
```

? Q #51 (● Advanced)

Tags: 🚀 MAANG-level , Objects , Iteration , Prototypes

🧠 Question:

Contrast the behavior of `Object.getOwnPropertyNames()` with a `for...in` loop.

✅ Answer:

The two methods serve different purposes for property inspection:

- `Object.getOwnPropertyNames(obj)` : Returns an array of strings that correspond to *all* of the object's **own** properties, regardless of whether they are enumerable or not. It does not look up the prototype chain.
- `for...in` loop: Iterates over the keys of all **enumerable** properties of an object. This includes both the object's own properties and any enumerable properties it inherits from its prototype chain

? Q #52 (● Intermediate)

Tags: ✂️ General Interview , Objects , Properties

🧠 Question:

What is the purpose of `Object.assign()` and what does it return?

✅ Answer:

`Object.assign()` is used to copy the values of all **enumerable own properties** from one or more source objects to a target object. It modifies the target object in

place and also returns the modified target object. It's a common way to merge objects or create a shallow copy.

? Q #53 (Intermediate)

Tags:  MAANG-level , this , Async , Call-ApPLY-Bind

 EXTREMELY IMPORTANT

 Question:

When a regular function is used as a callback in `setTimeout` inside an object method, `this` loses its context. Why does this happen and how can you fix it?

 Answer:

This happens because the `setTimeout` callback is executed by the JavaScript runtime in a separate context, not by the object itself. Therefore, `this` defaults to the global object (`window`) in non-strict mode or `undefined` in strict mode.

There are two common ways to fix this:

1. `.bind(this)`: Create a new function with `this` permanently bound to the correct object context.
2. **Arrow Function**: Use an arrow function for the callback, as it lexically inherits `this` from its surrounding scope (the object method).

```
jsxconst pet = {
  name: "Fluffy",
  speakLater: function() {
    // Fix using .bind()
    setTimeout(function() {
      console.log(this.name);
    }.bind(this), 500);

    // Modern fix using an arrow function
    setTimeout(() => {
      console.log(this.name); // `this` is lexically scoped
    }, 1000);
  }
};

pet.speakLater(); // Logs "Fluffy" twice
```

? Q #54 (Basic)

Tags:  General Interview , Functions , JS Fundamentals

 Question:

What are the three main ways to define a function in JavaScript?

✅ Answer:

The three main ways are:

1. **Function Declaration:** The standard `function name() {}` syntax. These are fully hoisted.
2. **Function Expression:** Assigning a function to a variable, like `const myFunction = function() {};`. These are not fully hoisted.
3. **Arrow Function:** The ES6 `() => {}` syntax, which provides a more concise way to write functions and has lexical `this` binding.

? Q #55 (🟡 Intermediate)

Tags: 🚀 MAANG-level , Functions , Hoisting

🧠 Question:

How does hoisting differ for a function declaration versus an arrow function assigned to a `let` variable?

✅ Answer:

- **Function Declaration:** The entire function (both name and body) is hoisted. You can call it before it appears in the code.
- **Arrow Function (with `let` or `const`):** Only the variable's declaration is hoisted, but it remains uninitialized in the Temporal Dead Zone (TDZ). The function assignment doesn't happen until the line is executed. Trying to call it before its definition results in a `ReferenceError`.

? Q #56 (🟡 Intermediate)

Tags: 🌱 General Interview , Scope , Closures

🧠 Question:

The LEGB rule is often discussed with Python. What does it stand for, and does it apply to JavaScript?

✅ Answer:

LEGB stands for **L**ocal, **E**nclosing, **G**lobal, **B**uilt-in. It describes the order in which a variable is looked up in nested scopes.

While the term "LEGB" is specific to Python, JavaScript's scope chain mechanism is very similar. When resolving a variable, JavaScript looks in:

1. The current **Local** scope.
2. The scope of any **Enclosing** (outer) functions (which enables closures).
3. The **Global** scope (`window` in browsers).

JavaScript does not have a distinct "Built-in" scope in the same way Python does; built-in APIs like `console` or `Math` exist on the global object.

? Q #57 (● Advanced)

Tags: 🚀 MAANG-level , `Objects` , `Prototypes`

🧠 Question:

How can you create an object that has no prototype, and why might you do this?

✅ Answer:

You can create an object with no prototype by using `Object.create(null)` . The resulting object does not inherit from `Object.prototype` , so it is completely "pure" or "empty"

This is useful for creating hashmaps or "dictionary" objects. By doing this, you prevent potential property name collisions with standard object methods like `toString` or `hasOwnProperty` , making your data structure safer and more predictable.

? Q #58 (● Intermediate)

Tags: ✂ General Interview , `Prototypes` , `Objects`

🧠 Question:

What is the modern, standard way to get an object's prototype, and why is it preferred over the older `__proto__` property?

✅ Answer:

The modern, standard method is `Object.getPrototypeOf(obj)` .

It is preferred because:

1. **It's the Official Standard:** It was introduced in ES5 as the canonical way to access a prototype.

2. **It's Read-Only:** It's a getter, which prevents accidental modification of an object's prototype chain. Directly assigning to `__proto__` is a slow operation and can lead to hard-to-debug issues in performance-critical code.
-

? Q #59 (● Advanced)

Tags: 🚀 MAANG-level , Prototypes , Inheritance

🔥 EXTREMELY IMPORTANT

🧠 Question:

In the context of prototypal inheritance, what is the crucial difference between a constructor function's `.prototype` property and an object instance's `__proto__` link?

✅ Answer:

- `Constructor.prototype` : This is a property on a **constructor function**. It is an object that serves as the **blueprint** for all instances created with that constructor. When you want to add a shared method for all instances, you add it to this `prototype` object.
- `instance.__proto__` : This is an internal property on an **object instance**. It is a **direct link** that points to the `prototype` object of the constructor that created it. This link is what the JavaScript engine uses to travel up the prototype chain during property lookups.

In summary: `prototype` is the template; `__proto__` is the instance's link to that template.

? Q #60 (● Basic)

Tags: ✖ General Interview , ES6 , Objects , Destructuring

🧠 Question:

What is object destructuring and how does it simplify code?

✅ Answer:

Object destructuring is an ES6 feature allowing you to unpack properties from objects into distinct variables. It provides a more concise and readable way to access an object's properties compared to repeatedly using dot or bracket notation.

```
jsxconst user = { name: "Alex", role: "Admin" };

// Before:// const name = user.name;// const role = user.role;// With destructuring:
const { name, role } = user;
console.log(name, role); // "Alex", "Admin"
```

? Q #61 (Intermediate)

Tags:  MAANG-level , Equality , NaN

 Question:

What are the two key ways that `Object.is()` is more precise than the strict equality operator (`===`)?

✓ Answer:

`Object.is()` improves upon `===` in two specific edge cases:

1. **NaN Equality:** `Object.is(NaN, NaN)` returns `true`, while `NaN === NaN` returns `false`. This makes `Object.is()` a reliable way to check for `NaN`.
2. **Signed Zeros:** It correctly distinguishes between `+0` and `0`. `Object.is(+0, -0)` returns `false`, while `+0 === -0` returns `true`.

? Q #62 (Basic)

Tags:  General Interview , Variables , Scope , Best Practices

 Question:

Why is using `var` generally discouraged in modern JavaScript in favor of `let` and `const`?

✓ Answer:

`var` is discouraged because its scoping and hoisting behavior can lead to bugs:

- **Scope:** `var` is function-scoped, not block-scoped. It can "leak" out of `if` blocks or `for` loops, causing unexpected behavior.
- **Hoisting:** `var` declarations are hoisted and initialized with `undefined`, allowing you to access the variable before its declaration without an error, which can hide bugs.
- **Re-declaration:** You can re-declare the same `var` variable in the same scope without an error, which can lead to accidental overwrites.

`let` and `const` have block scope and a Temporal Dead Zone (TDZ), making them safer and more predictable.

? Q #63 (Intermediate)

Tags:  MAANG-level , Closures , Functions , State

 Question:

Write a simple counter function that uses a closure to maintain its state across multiple calls.

 Answer:

A closure can be used to create a private state that persists between function calls. The outer function creates the state (the `count` variable), and the returned inner function has access to and modifies this state.

```
function createCounter() {  
  let count = 0; // This variable is in the closure's "backpack"  
  
  return function() {  
    count++;  
    console.log(count);  
  };  
}
```

```
const counter1 = createCounter();  
counter1(); // 1  
counter1(); // 2
```

```
const counter2 = createCounter(); // Creates a new, independent closure  
counter2(); // 1
```

Each call to `createCounter` creates a new, independent closure with its own `count` variable.

? Q #64 (Advanced)

Tags:  MAANG-level , this , Scope , Strict Mode

 Question:

What is "this substitution" and in which JavaScript mode does it occur?

 Answer:

"This substitution" is a behavior specific to **non-strict mode**. It occurs when a function is called without a context (e.g., as a standalone `myFunc()` call). If the

`this` value inside the function would otherwise be `null` or `undefined`, JavaScript automatically substitutes it with the global object (`window` in browsers).

In **strict mode**, this substitution does not happen, and `this` remains `undefined`, preventing accidental modifications to the global object.

? Q #65 (🟡 Intermediate)

Tags: 🌱 General Interview , Functions , Function Expression

🧠 Question:

Can a function expression be named? If so, what is a primary benefit of naming it?

✅ Answer:

Yes, a function expression can be named (e.g., `const myFunc = function myNamedFunc() { ... }`).

A primary benefit of naming the function expression is for **debugging**. The name `myNamedFunc` will appear in call stacks and error messages, making it much easier to identify the source of a problem compared to an anonymous function. It can also be used for recursion.

? Q #66 (🟡 Intermediate)

Tags: 🌱 General Interview , apply , Functions , Arrays

🧠 Question:

How can you use the `.apply()` method to find the maximum value in an array of numbers?

✅ Answer:

You can use `.apply()` on the `Math.max` function. `Math.max` expects a list of numbers as arguments, not a single array. `.apply()` is perfect for this because its second argument is an array of values that it unpacks and passes as individual arguments to the function.

```
jsxconst numbers = [5, 3, 8, 2, 9];  
// The elements of 'numbers' are passed as individual arguments to Math.max  
const max = Math.max.apply(null, numbers);  
console.log(max); // 9
```

The first argument is `null` because `Math.max` does not use a `this` context.

? Q #67 (🟡 Intermediate)

Tags: 🚀 MAANG-level , this , Arrow Functions , Callbacks

🔥 EXTREMELY IMPORTANT

🧠 Question:

What is the main advantage of using an arrow function for a callback inside an object method?

✅ Answer:

The main advantage is that arrow functions **lexically bind** `this`. This means they don't have their own `this` context; they automatically inherit `this` from their parent (enclosing) scope. When used as a callback inside an object method, the arrow function's `this` will correctly refer to the object instance, solving the common "lost `this`" problem without needing workarounds like `.bind(this)` or `const self = this`.

? Q #68 (🟡 Intermediate)

Tags: 🌱 General Interview , Objects , Arrays , Prototypes

🧠 Question:

What is the fundamental difference between an Array and a "plain" object in JavaScript?

✅ Answer:

While both are objects, their key difference lies in their **prototype**.

- A **plain object** (created with `{}` or `new Object()`) inherits from `Object.prototype`, giving it generic object methods. `task1.js+1`
 - An **Array** is a specialized object that inherits from `Array.prototype`. This gives it special properties and methods tailored for ordered collections, such as `.length`, `.push()`, `.slice()`, and `.forEach()`.
-

? Q #69 (🟡 Intermediate)

Tags: 🌱 General Interview , Classes , Prototypes , ES6

🔥 EXTREMELY IMPORTANT

🧠 Question:

Why is the `class` keyword in JavaScript often described as "syntactic sugar"?

✅ Answer:

The `class` keyword is called syntactic sugar because it does **not** introduce a new object-oriented inheritance model to JavaScript. Instead, it provides a cleaner, more familiar syntax that sits on top of the existing **prototypal inheritance** mechanism. Under the hood, a `class` definition still creates a constructor function and manages the `.prototype` property for methods, just as was done manually before ES6.

? Q #70 (🟢 Basic)

Tags: 🌱 General Interview , Objects , Property Access

🧠 Question:

When is it necessary to use bracket notation instead of dot notation to access an object's property?

✅ Answer:

You must use bracket notation (`obj['prop']`) in two main scenarios:

1. **Invalid Identifiers:** When the property key is not a valid JavaScript identifier, such as when it contains spaces, hyphens, or starts with a number (e.g., `obj['first-name']`).
2. **Dynamic Keys:** When the property key is stored in a variable. Dot notation can only access literal keys, while bracket notation evaluates the expression inside the brackets to get the key name.

```
jsxconst key = 'city';
const person = { 'home-town': 'New York', city: 'Boston' };
console.log(person['home-town']); // Works
console.log(person[key]); // Works, evaluates to person['city']
```

? Q #71 (🟡 Intermediate)

Tags: 🌱 General Interview , Objects , Iteration , ES6

🧠 Question:

What does `Object.entries()` return, and why is it particularly useful for iteration?

✅ Answer:

`Object.entries(obj)` returns an array containing an object's own enumerable `[key, value]` pairs. Each element of the returned array is itself a two-element array `([key, value])`.

It is especially useful for iteration because it allows you to easily loop over both the key and the value of an object at the same time using a `for...of` loop with destructuring.

```
jsxconst user = { name: "Sam", id: 123 };

for (const [key, value] of Object.entries(user)) {
  console.log(`${key}: ${value}`);
}
// Output:// name: Sam// id: 123
```

? Q #72 (🟡 Intermediate)

Tags: 🌱 General Interview , this , DOM

🧠 Question:

If you have an HTML button with an `onclick` attribute, what does `this` refer to inside that attribute's JavaScript code?

✅ Answer:

Inside an inline event handler attribute like `onclick`, the `this` keyword refers to the **HTML element** on which the handler is placed. This provides a direct reference to the element that triggered the event, allowing for easy manipulation.

```
xml <!-- `this` here refers to the button element →
<button onclick="this.textContent='Clicked!'">Click Me</button>
```

? Q #73 (🟡 Intermediate)

Tags: 🚀 MAANG-level , JS Fundamentals , Equality , Coercion

🧠 Question:

Explain step-by-step why `[] == ![]` evaluates to `true` in JavaScript.

✅ Answer:

This is a classic coercion puzzle that unfolds as follows:

1. **![] (Right side):** The logical NOT `!` operator coerces its operand to a boolean. An empty array `[]` is a truthy value. `!true` is `false`. The expression becomes `[] == false`.

2. `[] == false` (**Comparison**): The loose equality `==` operator sees an object on the left and a boolean on the right. It converts both operands to numbers.
 3. `ToNumber([])`: The array `[]` is first converted to a string via its `toString()` method, resulting in `""`. The empty string `""` is then converted to the number `0`.
 4. `ToNumber(false)`: The boolean `false` is converted to the number `0`.
 5. `0 == 0` (**Final Check**): The expression is now `0 == 0`, which evaluates to `true`.
-

? Q #74 (● Advanced)

Tags: 🚀 MAANG-level , Prototypes , Inheritance , Bugs

🧠 Question:

What is "Shared Mutable State" in the context of prototypes, and what kind of bug can it cause?

✅ Answer:

Shared Mutable State occurs when multiple object instances inherit a property from their prototype that is a mutable type, such as an array or an object. Because all instances share a reference to the *same* mutable property on the prototype, a modification made by one instance will be reflected in all other instances.

This can cause bugs where state changes in one part of an application unintentionally affect another. For example, if two objects inherit a `shoppingList` array from their prototype, adding an item to the list from one object will also add it to the list for the second object.

? Q #75 (● Advanced)

Tags: 🚀 MAANG-level , Functions , Constructors , Call-Applied-Bind

🔥 EXTREMELY IMPORTANT

🧠 Question:


Explain the binding priority of `new` versus `.bind()`. If a bound function is called with `new`, what will `this` refer to?

✅ Answer:

The `new` keyword has a **higher binding priority** than `.bind()`.

If you use the `new` operator on a function that was created with `.bind()`, the hard-bound `this` from `.bind()` is **ignored**. Instead, `this` will refer to the **newly created object instance**, as is standard for any constructor call. This ensures that even a bound function can still be used as a constructor to create new, distinct objects.

? Q #76 (Intermediate)

Tags:  General Interview , Functions , ES6

 Question:

Compare and contrast the `arguments` object with ES6 rest parameters (`...args`).

 Answer:

- **Availability:** The `arguments` object is available in traditional `function` declarations but **not** in arrow functions. Rest parameters (`...args`) can be used in all function types.
- **Type:** `arguments` is an array-like object; it lacks array methods like `.map()` or `.forEach()` unless you convert it first. Rest parameters collect arguments into a **true Array instance**, with full access to all array methods.
- **Binding:** `arguments` contains all arguments passed to the function. Rest parameters collect only the arguments that haven't been assigned to a named parameter, making it more flexible for handling a variable number of trailing arguments.

```
jsx // Rest parameters create a true array
function logArgs(...args) {
  console.log(Array.isArray(args)); // true
  args.forEach(arg => console.log(arg));
}
logArgs(1, 2, 3);
```

? Q #77 (Basic)

Tags:  General Interview , Functions , Debugging

 Question:

What is a primary benefit of giving a name to a function expression (a named function expression)?

 Answer:

The primary benefit is for **debugging**. When an error occurs within a named function expression, the function's name will appear in the call stack trace. This makes it significantly easier to identify the source of the error compared to an anonymous function, which might just show up as `(anonymous function)`.

```
jsx // Harder to debug if an error occurs inside
const anonymousFunc = function() { /* ... */ };

// Easier to debug, call stack will show 'myNamedFunc'
const namedFunc = function myNamedFunc() { /* ... */ };
```

? Q #78 (🟡 Intermediate)

Tags: 🟢 General Interview, Variables, Objects, const

🧠 Question:

Why is it possible to modify the properties of an object declared with `const`, while a primitive value declared with `const` cannot be changed?

✅ Answer:

The `const` declaration creates a read-only **reference** to a value.

- For **primitives** (like numbers or strings), the value itself is held in the variable. Since the reference is constant, the value cannot be changed.
- For **objects** (including arrays), the variable holds a constant reference to the object's location in memory, not the object's content. This means you cannot reassign the variable to a new object, but you can freely change the properties inside the existing object because that does not change the memory reference.

```
jsxconst user = { name: "Alice" };
user.name = "Bob"; // This is allowed, we are mutating the object.
// user = { name: "Charlie" }; // TypeError:
// Assignment to constant variable.
```

? Q #79 (🔴 Advanced)

Tags: 🚀 MAANG-level, JS Fundamentals, Objects, Coercion

🧠 Question:

Why is `new Boolean(false)` considered a "truthy" value in a boolean context?

✅ Answer:




This is a key example of how object wrappers and type coercion interact. `new Boolean(false)` creates an **object**, not a primitive boolean value. In JavaScript, all

objects (except `null`) are "truthy," regardless of their content. When this object is evaluated in a boolean context like an `if` statement, it coerces to `true`. This is why using object wrappers like `new Boolean()` is highly discouraged.

```
jsxconst value = new Boolean(false);

if (value) {
  // This code runs because `value` is an object, which is truthy.
  console.log("This is truthy!");
}
```

? Q #80 (🟡 Intermediate)

Tags:  General Interview ,  Objects ,  Copying

 Question:

Does `Object.assign()` perform a deep or shallow copy? Explain.

✅ Answer:

`Object.assign()` performs a **shallow copy**. It copies the values of enumerable own properties from source objects to a target object. If a property's value is a primitive (like a number or string), the value is copied. However, if a property's value is an object (like an array or another object), only the **reference** to that object is copied, not the object itself. Modifying a nested object in the copy will also affect the original.

? Q #81 (🟡 Intermediate)

Tags:  MAANG-level ,  Functions ,  this ,  Arrow Functions

 Question:

What happens if you attempt to change the `this` context of an arrow function using methods like `.call()`, `.apply()`, or `.bind()`?

✅ Answer:

It has no effect. The attempt will be ignored. Arrow functions have a lexically bound `this`, meaning their `this` value is permanently fixed to the `this` of their enclosing scope at the time of their creation. This lexical binding cannot be overridden by any method.

? Q #82 (🟡 Intermediate)

Tags: MAANG-level , Closures , Scope , ES6

EXTREMELY IMPORTANT

Question:

What is the modern ES6 solution to the classic `for` loop closure problem, which previously required an IIFE?

Answer:

The modern solution is to use the `let` keyword for the loop variable declaration instead of `var`. The `let` keyword is **block-scoped**, meaning a new, separate binding for the loop variable (`i`) is created for each iteration of the loop. Each `setTimeout` callback then closes over its own unique, iteration-specific `i`, preserving the correct value.

```
jsxfor (let i = 0; i < 3; i++) {  
  // 'let' creates a new `i` for each loop iteration.  
  setTimeout(function() {  
    console.log(i);  
  }, 1000);  
}  
// Output: 0, 1, 2
```

? Q #83 (Basic)

Tags: General Interview , Objects , Destructuring

Question:

Explain how to use aliasing (renaming) when destructuring an object property.

Answer:

When destructuring, you can assign an object property to a variable with a different name using a colon (`:`). The syntax is `{ originalKey: newName }`. This is useful for avoiding naming conflicts or for giving a more descriptive name to a variable.

```
jsxconst user = {  
  id: 101,  
  name: "Charlie"  
};  
  
// Extract the 'name' property into a variable called 'userName'  
const { name: userName } = user;  
  
console.log(userName); // "Charlie"// console.log(name); // ReferenceError: name is not defined
```

? Q #84 (🟡 Intermediate)

Tags: 🌱 General Interview , Prototypes , Best Practices

🧠 Question:

Your notes mention that `__proto__` is "old fashioned." What are the primary reasons `Object.getPrototypeOf()` is the preferred modern approach?

✅ Answer:

`Object.getPrototypeOf()` is preferred for several reasons:

1. **Standardization:** It is the official, standard ECMAScript 5 method for accessing an object's prototype. `__proto__` was a non-standard, browser-specific implementation that was only standardized later for compatibility.
 2. **Immutability:** `Object.getPrototypeOf()` is a read-only getter. It prevents accidental modification of the prototype chain, which can cause severe, hard-to-debug issues. Directly setting `__proto__` is a slow operation and considered bad practice.
 3. **Consistency:** It's a standard static method on the `Object` constructor, fitting a more consistent and predictable programming model.
-

? Q #85 (🟡 Intermediate)

Tags: 🌱 General Interview , Classes , Prototypes

🧠 Question:

What does the JavaScript engine create behind the scenes when you use the `class` keyword?

✅ Answer:

The `class` keyword is syntactic sugar over JavaScript's prototype-based system. When you define a `class`, JavaScript creates:

1. A **constructor function** with the same name as the class.
 2. All methods defined within the class (like `greet()`) are placed on the constructor function's `.prototype` **object**. This makes them shared and accessible to all instances of the class via the prototype chain.
-

? Q #86 (🔴 Advanced)

Tags: MAANG-level , Prototypes , Objects

Question:

What is the practical difference between using `Object.create(someProto)` and `new SomeConstructor()` for creating an object?

Answer:

Both methods create an object that inherits from a prototype, but they are used differently:

- `new SomeConstructor()` : This is used with **constructor functions**. It's a multi-step process that creates a new object, sets its prototype to `SomeConstructor.prototype`, binds `this` to the new object, and executes the constructor's code, which often initializes instance-specific properties.
- `Object.create(someProto)` : This is a more direct way to set up inheritance. It creates a new, empty object and immediately sets its internal `[[Prototype]]` to the object you pass as the first argument (`someProto`). It doesn't involve running a constructor function. It is ideal for creating objects that inherit from other plain objects.

? Q #87 (Intermediate)

Tags: General Interview , Objects

Question:

If you use `Object.assign()` with multiple source objects that have conflicting property keys, how is the final target object determined?

Answer:

Properties from later source objects in the argument list will overwrite properties from earlier ones. `Object.assign()` processes the source objects in order from left to right, copying their properties onto the target. If a key already exists on the target, its value will be updated by the last source object that has that same key.

```
jsxconst target = { a: 1, b: 1 };  
const source1 = { b: 2, c: 2 };  
const source2 = { c: 3 };
```

```
Object.assign(target, source1, source2);
```

```
console.log(target); // { a: 1, b: 2, c: 3 } // b was overwritten by source1, c was overwritten by source2
```

? Q #88 (🟡 Intermediate)

Tags: 🌱 General Interview , Hoisting , Functions

🧠 Question:

What is the specific error you get when you try to invoke a function expression assigned to a `var` before its definition, and why?

✅ Answer:

You will get a `TypeError`, typically stating that the variable "is not a function."

This happens because of hoisting. The declaration `var myFunc` is hoisted to the top of its scope and initialized with `undefined`. When you try to invoke `myFunc()`, you are essentially trying to execute `undefined()`, which is not a function, hence the `TypeError`. The actual function assignment only happens when the execution reaches that line.

? Q #89 (🟡 Intermediate)

Tags: 🌱 General Interview , this , DOM , Callbacks

🧠 Question:

Your notes mention using `.bind()` for `setTimeout` callbacks. How would this apply to a browser event listener, and why is it necessary?

✅ Answer:

It applies in the exact same way. When an object's method is used as an event listener (e.g., `button.addEventListener('click', myObj.myMethod)`), the `this` context inside `myMethod` when it's called will refer to the **DOM element that triggered the event** (the button), not `myObj`.

Using `.bind(myObj)` is necessary to create a new function where `this` is permanently bound to the `myObj` instance, ensuring that you can correctly access the object's properties and methods from within the event handler.

? Q #90 (🟡 Intermediate)

Tags: 🌱 General Interview , Prototypes

🧠 Question:

What does `Object.getPrototypeOf(Object.prototype)` return, and what does this signify?

✅ Answer:

It returns `null`.

This signifies that `Object.prototype` is at the very top of the prototype chain for most standard objects. When the JavaScript engine is looking for a property and reaches an object whose prototype is `null`, the search stops. This is the end of the line for prototypal inheritance.

? Q #91 (● Basic)

Tags: ✖ General Interview , Scope

 Question:

How does JavaScript's scope chain lookup relate to the LEGB (Local, Enclosing, Global, Built-in) rule from other languages like Python?

✔ Answer:

The concepts are almost identical. LEGB stands for **L**ocal, **E**nclosing, **G**lobal, **B**uilt-in, which is the order a variable is looked for. JavaScript's scope chain follows the same pattern:

1. **Local Scope:** The current function's scope.
2. **Enclosing Scope:** The scope of any outer functions (this is the basis for closures).
3. **Global Scope:** The outermost scope (`window` in browsers).

The main difference is that JavaScript doesn't have a distinct "Built-in" scope; built-in APIs are part of the global object.

? Q #92 (● Advanced)

Tags: ✖ MAANG-level , Iteration , Arrays , Bugs

 Question:

Why is using a `for...in` loop to iterate over an Array generally considered a bad practice?

✔ Answer:

It's bad practice for two main reasons:

1. **It iterates over inherited properties:** A `for...in` loop will traverse up the prototype chain and may include properties from `Array.prototype` or any

custom properties added to it. This can lead to unexpected items in your loop.

2. **It iterates over property keys, not indices:** The keys are strings (`"0"` , `"1"` , `"2"`), not numbers, and the order of iteration is not guaranteed to be sequential, which is usually required when processing arrays.

For arrays, you should always use standard loops like `for...of` , a traditional `for` loop, or array methods like `.forEach()` .

? Q #93 (🟡 Intermediate)

Tags: 🟢 General Interview , Variables , `const`

🧠 Question:

What is the core difference between `let` and `const` if you can still modify the properties of an object declared with `const` ?

✅ Answer:

The core difference lies in **reassignment**.

- `let` allows a variable to be reassigned to a completely new value or reference.
- `const` **prevents reassignment**. Once a variable is assigned a value (or a reference to an object), that variable cannot be pointed to anything else.

For `const` objects, the lock is on the variable's reference to the object, not on the object's contents.

```
let user = { name: "A" };  
user = { name: "B" }; // Allowed
```

```
const admin = { name: "X" };  
// admin = { name: "Y" }; // TypeError: Assignment to constant variable.
```

? Q #94 (🟡 Intermediate)

Tags: 🟢 General Interview , Functions , `apply`

🧠 Question:

How can you use the `.apply()` method with `Math.max` to find the largest number in an array?

✅ Answer:

The `Math.max()` function expects a list of numbers as individual arguments, not a single array. The `.apply()` method is perfect for this situation because it takes an array of arguments and passes them to the function as if they were listed out individually.

```
jsxconst numbers = [5, 10, 2, 8];  
// `null` is used for the `this` context because Math.max doesn't need one.// The `numbers` array is unpacked into  
arguments for `Math.max`.  
const max = Math.max.apply(null, numbers);  
console.log(max); // 10
```

? Q #95 (🟡 Intermediate)

Tags: 🚀 MAANG-level , this , Strict Mode

🧠 Question:

Your notes mention "this substitution." What exactly happens to `this` inside a regular function in non-strict mode if the function is called with a `this` context of `null` or `undefined` ?

✅ Answer:

In non-strict mode, if a function is called in a way that its `this` context would be `null` or `undefined` (like a simple standalone function call), JavaScript automatically substitutes `this` with the **global object** (`window` in a browser). This behavior is known as "this substitution." In strict mode, this does not happen, and `this` remains `undefined` .

? Q #96 (🟣 Expert)

Tags: 🚀 MAANG-level , Objects , Prototypes

🧠 Question:

The `isPlainObject` function from your notes uses the check


`Function.prototype.toString.call(Ctor) === Function.prototype.toString.call(Object)` . What is the purpose of this specific check?

✅ Answer:

This check is a robust way to verify if an object's constructor (`Ctor`) is the original, built-in `Object` constructor and not a custom class or subclass. It works because `Function.prototype.toString.call()` returns the source code of a function as a string. For built-in functions, this returns a standardized, native code representation (e.g., `"function Object() { [native code] }"`). By comparing the string

representation of the value's constructor to that of the global `Object` constructor, it ensures the object was created from a plain `{}` or `new Object()` and is not an instance of a different class (`Array`, `Date`, custom `class Person`, etc.).

? Q #97 (Intermediate)

Tags:  General Interview , `Functions` , `bind`

 Question:


Explain how `.bind()` can be used to create a partially applied function (a function with preset arguments).

 Answer:

Besides setting the `this` context, `.bind()` can also accept initial arguments that will be permanently "bound" to the new function it returns. When the new function is called later, any additional arguments it receives are appended to the list of preset arguments. This is known as partial application or currying.

```
jsxfunction multiply(a, b) {  
  return a * b;  
}  
  
// Create a new function 'double' where the first argument `a` is always 2.  
const double = multiply.bind(null, 2);  
  
console.log(double(5)); // 10 (calls multiply(2, 5))  
console.log(double(7)); // 14 (calls multiply(2, 7))
```

? Q #98 (Basic)

Tags:  General Interview , `Objects`

 Question:

What is the return value of `Object.assign()`, and how can this be useful?

 Answer:

`Object.assign()` returns the **modified target object**. The target object is the first argument passed to the method. This is useful for method chaining or for creating a new merged object in a single line without needing a separate variable declaration first.

```
jsxconst obj1 = { a: 1 };  
const obj2 = { b: 2 };  
  
// Create a new object by passing an empty object as the target
```



```
const merged = Object.assign({}, obj1, obj2);
console.log(merged); // { a: 1, b: 2 }
```

? Q #99 (Advanced)

Tags:  MAANG-level , Objects , Prototypes

 Question:

Can you create a "pure" object that doesn't inherit any methods from `Object.prototype` ? If so, how and why would this be useful?

 Answer:

Yes, you can create a pure object using `Object.create(null)` . This creates an object whose internal `[[Prototype]]` is `null` , so it does not sit on any prototype chain.

This is extremely useful for creating hashmaps or dictionaries. Since the object has no inherited properties, you can use any string as a key without worrying about accidentally colliding with built-in object property names like `toString` or `constructor` .

? Q #100 (Expert)

Tags:  MAANG-level , this , Scope , Binding

 EXTREMELY IMPORTANT

 Question:

Combining binding rules: If an arrow function is defined inside a regular function that was created with `.bind()` , and this bound function is then invoked with `new` , what will `this` refer to inside the arrow function?

 Answer:

The `this` inside the arrow function will refer to the **newly created object instance**.

Here's the priority breakdown:

1. The `new` operator has higher priority than `.bind()` . When `new` is used, the `this` context supplied to `.bind()` is ignored, and `this` inside the regular function becomes the new object instance.
2. The arrow function is defined within that scope, so it lexically captures the `this` of its parent.

3. Therefore, the arrow function's `this` is the new object created by the `new` operator.

? Q #101 (🟡 Intermediate)

Tags: 🌱 General Interview , Classes , Prototypes , Inheritance

🧠 Question:

How does the `extends` keyword in a JavaScript `class` relate to the underlying prototypal inheritance model?

✅ Answer:

The `extends` keyword is syntactic sugar that directly sets up prototypal inheritance between two constructor functions. When you write `class Child extends Parent`, JavaScript sets the `Child.prototype`'s internal `[[Prototype]]` link to point to `Parent.prototype`. This creates a prototype chain where instances created from `Child` can access methods defined on `Parent.prototype`, effectively inheriting from the `Parent` class in a clean, readable way.

```
jsxclass Person {  
  greet() { console.log("Hello!"); }  
}  
  
// `extends` sets up the prototype chain: Friend.prototype → Person.prototype  
class Friend extends Person { }  
  
const bob = new Friend();  
bob.greet(); // Works! The method is found on Person.prototype.
```

? Q #102 (🔴 Advanced)

Tags: 🚀 MAANG-level , Objects , Prototypes

🧠 Question:

What is the functional difference between creating an object using `const obj = {}` versus `const obj = Object.create(Object.prototype)` ?

✅ Answer:

While both methods create a new object that inherits from `Object.prototype`, the object literal `{}` is the idiomatic and preferred syntax for its simplicity and readability. `Object.create(Object.prototype)` achieves the same inheritance link explicitly but is more verbose. The main functional difference is that the literal syntax can

also initialize properties directly (e.g., `{ a: 1 }`), whereas `Object.create()` only sets the prototype; properties must be added in a separate step or via a second argument (a property descriptors map).

? Q #103 (● Advanced)

Tags: 🚀 MAANG-level, this, Scope, Coercion

🧠 Question:

The notes mention a behavior called "this substitution." What is this behavior, and in which JavaScript mode is it active?

✅ Answer:

"This substitution" is a behavior that occurs in **non-strict mode**. When a regular function is called without a specific `this` context (e.g., as a standalone `myFunc()` call), its `this` value would normally be `undefined`. However, in non-strict mode, JavaScript automatically "substitutes" this `undefined` value with the **global object** (`window` in browsers). This can lead to accidental modifications of the global scope. In **strict mode**, this substitution is disabled, and `this` correctly remains `undefined`.

? Q #104 (● Expert)

Tags: 🚀 MAANG-level, Objects, Types, Bugs

🧠 Question:

In the provided `isPlainObject` function, the first check is `value === null || typeof value !== "object"`. Why is it necessary to check for `null` explicitly, given that `typeof null` returns `"object"`?

✅ Answer:

This explicit check for `null` is crucial because of a long-standing bug in JavaScript where `typeof null` incorrectly returns `"object"`. Without first checking `value === null`, a `null` value would pass the `typeof value !== "object"` condition (since `"object" !== "object"` is false) and be incorrectly processed as a potential plain object by the rest of the function. This would lead to a `TypeError` on the next line when the code attempts to call `Object.getPrototypeOf(null)`. The `value === null` check acts as a safeguard to handle this specific historical edge case.

? Q #105 (Intermediate)

Tags:  General Interview , Closures , Memory Management

 Question:

How can closures inadvertently cause memory leaks in a JavaScript application?

 Answer:

A memory leak can occur when a closure maintains a reference to a variable from its parent scope, and that variable in turn holds a large object (like a DOM element, a large array, or a data structure). Even if the large object is no longer needed by the main application, the JavaScript garbage collector cannot reclaim its memory as long as the closure, which is still accessible (e.g., as an event listener or a `setTimeout` callback), holds a reference to its scope. The memory is "leaked" because it cannot be freed, leading to unnecessary memory consumption.