JavaScript Interview Notes - Complete Guide

JavaScript Core Fundamentals

Data Types and Variables

Primitive Data Types

Interview One-liner: "JavaScript has 7 primitive data types: string, number, boolean, null, undefined, symbol, and bigint - all stored by value, not reference."

JavaScript has 7 primitive data types:

```javascript
// String
let name = "John";
let message = 'Hello World';

// Number
let age = 25;
let price = 99.99;

// Boolean
let isActive = true;
let isComplete = false;

// Null (intentionally empty)
let data = null;

// Undefined (declared but not assigned)
let x;
console.log(x); // undefined

// Symbol (unique identifier)
let id = Symbol('id');

// BigInt (large integers)
let bigNumber = 1234567890123456789012345678901234567890n;
```

var vs let vs const

Interview One-liner: "var is function-scoped and hoisted with undefined, let/const are block-scoped with temporal dead zone, and const cannot be reassigned."

Key Differences:

Feature	var	let	const
Scope	Function/Global	Block	Block
Hoisting	Yes (undefined)	Yes (TDZ)	Yes (TDZ)
Re-declaration	Yes	No	No
Re-assignment	Yes	Yes	No

javascript

```javascript
// VAR - Function scoped
function example() {
  if (true) {
    var x = 1;
  }
  console.log(x); // 1 (accessible outside block)
}

// LET - Block scoped
function example() {
  if (true) {
    let y = 1;
  }
  console.log(y); // ReferenceError: y is not defined
}

// CONST - Block scoped, cannot be reassigned
const PI = 3.14;
// PI = 3.15; // TypeError: Assignment to constant variable

// But objects/arrays can be modified
const user = { name: "John" };
user.age = 25; // This works
```

Temporal Dead Zone (TDZ):

javascript

```
console.log(a); // undefined (hoisted)
console.log(b); // ReferenceError (TDZ)
console.log(c); // ReferenceError (TDZ)

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

Type Coercion and Comparison

Interview One-liner: "Double equals (==) performs type coercion before comparison, triple equals (===) compares both value and type without coercion."

== vs ===:

javascript

```
// == (loose equality - allows type coercion)
5 == "5"   // true
null == undefined  // true
0 == false  // true

// === (strict equality - no type coercion)
5 === "5"   // false
null === undefined  // false
0 === false // false
```

Truthy and Falsy Values

Interview One-liner: "Only 8 values are falsy in JavaScript: false, 0, -0, 0n, empty string, null, undefined, and NaN - everything else is truthy."

Falsy values (only 8):

javascript

```
false
0
-0
0n
""
null
undefined
NaN
```

Everything else is truthy:

```javascript
if ("0") console.log("truthy");   // runs
if ([]) console.log("truthy");    // runs
if ({}) console.log("truthy");    // runs
if (-1) console.log("truthy");    // runs
```

Functions and Scope

Function Declarations vs Expressions

Interview One-liner: "Function declarations are hoisted completely and can be called before definition, function expressions are not hoisted and create functions at runtime."

```javascript
// Function Declaration (hoisted)
console.log(add(2, 3)); // 5 (works before declaration)

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

// Function Expression (not hoisted)
console.log(subtract(5, 2)); // TypeError: subtract is not a function

var subtract = function(a, b) {
  return a - b;
};
```

Arrow Functions

Interview One-liner: "Arrow functions don't have their own 'this', arguments object, or prototype property, and cannot be used as constructors."

javascript

```javascript
// Regular function
function regular(x) {
  return x * 2;
}

// Arrow function
const arrow = (x) => x * 2;

// Key differences:
// 1. No 'this' binding
const obj = {
  name: "John",
  regular: function() {
    console.log(this.name); // "John"
  },
  arrow: () => {
    console.log(this.name); // undefined (inherits from parent scope)
  }
};

// 2. Cannot be used as constructors
const Person = (name) => {
  this.name = name;
};
// new Person("John"); // TypeError

// 3. No arguments object
function regular() {
  console.log(arguments); // [1, 2, 3]
}
const arrow = () => {
  console.log(arguments); // ReferenceError
};
regular(1, 2, 3);
```

Closures

Interview One-liner: "A closure is when an inner function has access to outer function's variables even after the outer function has finished executing."

```javascript
// Closure: inner function has access to outer function's variables
function outerFunction(x) {
  // This is the outer function's scope

  function innerFunction(y) {
    console.log(x + y); // Can access x from outer scope
  }

  return innerFunction;
}

const addFive = outerFunction(5);
addFive(3); // 8

// Practical example: Creating private variables
function createCounter() {
  let count = 0;

  return {
    increment: () => ++count,
    decrement: () => --count,
    getCount: () => count
  };
}

const counter = createCounter();
console.log(counter.getCount()); // 0
counter.increment();
console.log(counter.getCount()); // 1
// count is not accessible from outside
```

Higher-Order Functions

Interview One-liner: "Higher-order functions either take functions as arguments or return functions as results - like map, filter, reduce."

```javascript
```

```javascript
// Function that takes another function as argument
function higherOrder(callback) {
  callback();
}

// Function that returns another function
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

const double = multiplier(2);
console.log(double(5)); // 10

// Common higher-order functions
const numbers = [1, 2, 3, 4, 5];

// map
const doubled = numbers.map(n => n * 2);

// filter
const evens = numbers.filter(n => n % 2 === 0);

// reduce
const sum = numbers.reduce((acc, n) => acc + n, 0);
```

The 'this' Keyword

Interview One-liner: "The value of 'this' depends on how a function is called - it's the object before the dot in method calls, or can be set with call/apply/bind."

```javascript
```

```
// Global context
console.log(this); // Window object (browser) or global (Node.js)

// Object method
const person = {
  name: "John",
  greet: function() {
    console.log(this.name); // "John"
  }
};

// Constructor function
function Person(name) {
  this.name = name;
  this.greet = function() {
    console.log(this.name);
  };
}

// Arrow functions inherit 'this'
const obj = {
  name: "John",
  regularMethod: function() {
    const arrowFunction = () => {
      console.log(this.name); // "John" (inherits from regularMethod)
    };
    arrowFunction();
  }
};

// Call, Apply, Bind
const person1 = { name: "John" };
const person2 = { name: "Jane" };

function introduce(age) {
  console.log(`Hi, I'm ${this.name}, ${age} years old`);
}

introduce.call(person1, 25);   // Hi, I'm John, 25 years old
introduce.apply(person2, [30]); // Hi, I'm Jane, 30 years old
```

```javascript
const boundFunction = introduce.bind(person1);
boundFunction(25); // Hi, I'm John, 25 years old
```

Asynchronous JavaScript

Event Loop, Call Stack, and Callback Queue

Interview One-liner: "JavaScript is single-threaded with an event loop that processes the call stack first, then microtasks (Promises), then macrotasks (setTimeout)."

javascript

```javascript
console.log("1");

setTimeout(() => {
  console.log("2");
}, 0);

Promise.resolve().then(() => {
  console.log("3");
});

console.log("4");

// Output: 1, 4, 3, 2
// Explanation:
// 1. Synchronous code runs first (1, 4)
// 2. Microtasks (Promises) run before macrotasks (setTimeout)
// 3. Then macrotasks run (2)
```

Promises

Interview One-liner: "Promises represent eventual completion of asynchronous operations with three states: pending, fulfilled, or rejected."

javascript

```javascript
// Creating a Promise
const myPromise = new Promise((resolve, reject) => {
  const success = true;

  if (success) {
    resolve("Operation successful!");
  } else {
    reject("Operation failed!");
  }
});

// Using Promise
myPromise
  .then(result => console.log(result))
  .catch(error => console.log(error));

// Promise chaining
fetch('/api/user/1')
  .then(response => response.json())
  .then(user => fetch(`/api/posts/${user.id}`))
  .then(response => response.json())
  .then(posts => console.log(posts))
  .catch(error => console.log('Error:', error));

// Promise.all (wait for all)
Promise.all([
  fetch('/api/users'),
  fetch('/api/posts'),
  fetch('/api/comments')
])
.then(responses => {
  // All requests completed
})
.catch(error => {
  // If any request fails
});

// Promise.race (first to complete)
Promise.race([
  fetch('/api/fast'),
  fetch('/api/slow')
])
.then(result => {
```

```
  // Result from whichever completes first
});
```

Async/Await

Interview One-liner: "Async/await is syntactic sugar over Promises that makes asynchronous code look and behave like synchronous code."

javascript

```javascript
// Converting Promise to async/await
async function fetchUserPosts(userId) {
  try {
    const userResponse = await fetch(`/api/user/${userId}`);
    const user = await userResponse.json();

    const postsResponse = await fetch(`/api/posts/${user.id}`);
    const posts = await postsResponse.json();

    return posts;
  } catch (error) {
    console.log('Error:', error);
    throw error; // Re-throw if needed
  }
}

// Using the async function
fetchUserPosts(1)
  .then(posts => console.log(posts))
  .catch(error => console.log('Failed:', error));

// Async/await with Promise.all
async function fetchAllData() {
  try {
    const [users, posts, comments] = await Promise.all([
      fetch('/api/users').then(r => r.json()),
      fetch('/api/posts').then(r => r.json()),
      fetch('/api/comments').then(r => r.json())
    ]);

    return { users, posts, comments };
  } catch (error) {
    console.log('Error fetching data:', error);
  }
}
```

setTimeout and setInterval

Interview One-liner: "setTimeout executes code once after a delay, setInterval executes repeatedly at intervals - both return IDs for cancellation."

```javascript
```

```javascript
// setTimeout - runs once after delay
const timeoutId = setTimeout(() => {
  console.log("This runs after 2 seconds");
}, 2000);

// Cancel setTimeout
clearTimeout(timeoutId);

// setInterval - runs repeatedly
const intervalId = setInterval(() => {
  console.log("This runs every 1 second");
}, 1000);

// Cancel setInterval
clearInterval(intervalId);

// Common pattern: cleanup intervals
function startTimer() {
  let count = 0;
  const interval = setInterval(() => {
    count++;
    console.log(count);

    if (count >= 5) {
      clearInterval(interval);
    }
  }, 1000);
}
```

ES6+ Modern JavaScript Features

Template Literals

Interview One-liner: "Template literals use backticks for string interpolation with ${} syntax and support multi-line strings."

```javascript
```

```javascript
const name = "John";
const age = 25;

// Old way
const message1 = "Hello, my name is " + name + " and I'm " + age + " years old.";

// Template literal
const message2 = `Hello, my name is ${name} and I'm ${age} years old.`;

// Multi-line strings
const html = `
  <div>
    <h1>${name}</h1>
    <p>Age: ${age}</p>
  </div>
`;

// Expressions in template literals
const price = 100;
const tax = 0.1;
console.log(`Total: $${(price * (1 + tax)).toFixed(2)}`);
```

Destructuring Assignment

Interview One-liner: "Destructuring extracts values from arrays or properties from objects into distinct variables using pattern matching syntax."

```javascript
```

```javascript
// Array destructuring
const colors = ["red", "green", "blue"];
const [first, second, third] = colors;
console.log(first); // "red"

// Skipping elements
const [primary, , tertiary] = colors;

// Default values
const [a, b, c, d = "yellow"] = colors;

// Object destructuring
const person = {
  name: "John",
  age: 25,
  city: "New York"
};

const { name, age, city } = person;

// Renaming variables
const { name: fullName, age: years } = person;

// Default values
const { name, age, country = "USA" } = person;

// Nested destructuring
const user = {
  id: 1,
  profile: {
    name: "John",
    settings: {
      theme: "dark"
    }
  }
};

const { profile: { name, settings: { theme } } } = user;

// Function parameter destructuring
function greet({ name, age }) {
  console.log(`Hello ${name}, you are ${age} years old`);
}
```

```
greet({ name: "John", age: 25 });
```

Default and Rest Parameters

Interview One-liner: "Default parameters provide fallback values when arguments are undefined, rest parameters collect remaining arguments into an array."

javascript

```
// Default parameters
function greet(name = "Guest", message = "Hello") {
  console.log(`${message}, ${name}!`);
}

greet(); // "Hello, Guest!"
greet("John"); // "Hello, John!"
greet("John", "Hi"); // "Hi, John!"

// Rest parameters
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3, 4, 5)); // 15

// Combining regular and rest parameters
function introduce(firstName, lastName, ...hobbies) {
  console.log(`I'm ${firstName} ${lastName}`);
  console.log(`My hobbies are: ${hobbies.join(', ')}`);
}

introduce("John", "Doe", "reading", "coding", "gaming");
```

Spread Operator

Interview One-liner: "Spread operator (...) expands iterables into individual elements for copying arrays/objects or passing multiple arguments."

javascript

```javascript
// Array spread
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]

// Copy array
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];

// Object spread
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };
const combined = { ...obj1, ...obj2 }; // { a: 1, b: 2, c: 3, d: 4 }

// Copy object
const original = { name: "John", age: 25 };
const copy = { ...original };

// Override properties
const updated = { ...original, age: 26 };

// Function calls
function add(a, b, c) {
  return a + b + c;
}

const numbers = [1, 2, 3];
console.log(add(...numbers)); // 6
```

Classes and Inheritance

Interview One-liner: "ES6 classes are syntactic sugar over prototypal inheritance with constructor, methods, static methods, and extends/super for inheritance."

```javascript
```

```
// Basic class
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hello, I'm ${this.name}`);
  }

  // Static method
  static createAdult(name) {
    return new Person(name, 18);
  }
}

// Inheritance
class Student extends Person {
  constructor(name, age, grade) {
    super(name, age); // Call parent constructor
    this.grade = grade;
  }

  study() {
    console.log(`${this.name} is studying`);
  }

  // Override parent method
  greet() {
    super.greet(); // Call parent method
    console.log(`I'm in grade ${this.grade}`);
  }
}

const student = new Student("Alice", 16, "10th");
student.greet();
student.study();
```

Modules (Import/Export)

Interview One-liner: "ES6 modules use import/export for code organization with named exports, default exports, and static analysis benefits."

```javascript
// math.js (exporting)
export const PI = 3.14159;

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

export function subtract(a, b) {
  return a - b;
}

// Default export
export default function multiply(a, b) {
  return a * b;
}

// main.js (importing)
import multiply from './math.js'; // Default import
import { add, subtract, PI } from './math.js'; // Named imports
import * as math from './math.js'; // Import all

console.log(add(2, 3)); // 5
console.log(multiply(4, 5)); // 20
console.log(math.PI); // 3.14159
```

Maps, Sets, WeakMaps, and WeakSets

Interview One-liner: "Map/Set store unique values with any data types as keys/values, WeakMap/WeakSet use weak references and allow garbage collection."

```javascript
```

```javascript
// Map - key-value pairs with any type of keys
const map = new Map();
map.set('name', 'John');
map.set(1, 'number key');
map.set(true, 'boolean key');

console.log(map.get('name')); // "John"
console.log(map.has('name')); // true
console.log(map.size); // 3

// Iterating Map
for (let [key, value] of map) {
  console.log(key, value);
}

// Set - unique values
const set = new Set([1, 2, 3, 3, 4]);
console.log(set); // Set {1, 2, 3, 4}

set.add(5);
set.delete(1);
console.log(set.has(2)); // true

// WeakMap - weak references, only objects as keys
const weakMap = new WeakMap();
let obj = { name: "John" };
weakMap.set(obj, "some value");

// WeakSet - weak references, only objects as values
const weakSet = new WeakSet();
weakSet.add(obj);
```

Symbols

Interview One-liner: "Symbols are unique primitive values used as object property keys to avoid naming collisions and create private properties."

javascript

```javascript
// Creating symbols
const id = Symbol('id');
const anotherId = Symbol('id');

console.log(id === anotherId); // false (always unique)

// Using symbols as object keys
const user = {
  name: "John",
  [id]: 123
};

console.log(user[id]); // 123

// Well-known symbols
const obj = {
  [Symbol.iterator]: function* () {
    yield 1;
    yield 2;
    yield 3;
  }
};

for (let value of obj) {
  console.log(value); // 1, 2, 3
}
```

Generators

Interview One-liner: "Generators are functions that can pause and resume execution using yield, returning an iterator object with next() method."

```javascript
```

```javascript
// Generator function
function* numberGenerator() {
  yield 1;
  yield 2;
  yield 3;
}

const gen = numberGenerator();
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: 3, done: false }
console.log(gen.next()); // { value: undefined, done: true }

// Infinite generator
function* infiniteNumbers() {
  let num = 0;
  while (true) {
    yield num++;
  }
}

const infinite = infiniteNumbers();
console.log(infinite.next().value); // 0
console.log(infinite.next().value); // 1
```

Recent Features (ES2020+)

Optional Chaining (?.)

Interview One-liner: "Optional chaining safely accesses nested object properties without throwing errors if intermediate values are null or undefined."

```javascript
```

```javascript
const user = {
  name: "John",
  address: {
    street: "123 Main St",
    city: "New York"
  }
};

// Without optional chaining (old way)
const zipCode = user && user.address && user.address.zipCode;

// With optional chaining
const zipCode2 = user?.address?.zipCode; // undefined (no error)

// With arrays
const firstFriend = user?.friends?.[0]?.name;

// With methods
const result = user?.someMethod?.();
```

Nullish Coalescing (??)

Interview One-liner: "Nullish coalescing returns the right operand only when the left operand is null or undefined, unlike || which triggers on all falsy values."

javascript

```javascript
// || operator treats falsy values as fallback triggers
const value1 = 0 || "default"; // "default"
const value2 = "" || "default"; // "default"
const value3 = false || "default"; // "default"

// ?? operator only treats null/undefined as fallback triggers
const value4 = 0 ?? "default"; // 0
const value5 = "" ?? "default"; // ""
const value6 = false ?? "default"; // false
const value7 = null ?? "default"; // "default"
const value8 = undefined ?? "default"; // "default"

// Practical example
function processUser(user) {
  const name = user.name ?? "Anonymous";
  const age = user.age ?? 0;
  const isActive = user.isActive ?? true;
}
```

Private Class Fields

Interview One-liner: "Private class fields use # prefix and are only accessible within the class, providing true encapsulation in JavaScript classes."

```javascript
```

```javascript
class BankAccount {
  // Private fields (start with #)
  #balance = 0;
  #accountNumber;

  constructor(accountNumber) {
    this.#accountNumber = accountNumber;
  }

  // Private method
  #validateAmount(amount) {
    return amount > 0 && typeof amount === 'number';
  }

  deposit(amount) {
    if (this.#validateAmount(amount)) {
      this.#balance += amount;
    }
  }

  getBalance() {
    return this.#balance;
  }
}

const account = new BankAccount("12345");
account.deposit(100);
console.log(account.getBalance()); // 100

// These would throw errors:
// console.log(account.#balance); // SyntaxError
// account.#validateAmount(50); // SyntaxError
```

Top-Level Await

Interview One-liner: "Top-level await allows using await outside async functions at the module level, simplifying async module initialization."

```javascript
```

```
// Before: had to wrap in async function
(async () => {
  const response = await fetch('/api/data');
  const data = await response.json();
  console.log(data);
})();

// Now: can use await at module level
const response = await fetch('/api/data');
const data = await response.json();
console.log(data);

// Conditional imports
const theme = await import(
  isDarkMode ? './dark-theme.js' : './light-theme.js'
);
```

Quick Reference Summary

Common Patterns

- **Checking for undefined**: `value ?? defaultValue`
- **Safe property access**: `obj?.prop?.nestedProp`
- **Array operations**: `map()`, `filter()`, `reduce()`
- **Async operations**: `async/await` with try/catch
- **Object copying**: `{...original}` or `[...original]`
- **Destructuring**: `const {prop} = obj` or `const [item] = array`

Performance Tips

- Use `const` by default, `let` when reassigning, avoid `var`
- Prefer arrow functions for callbacks
- Use template literals instead of string concatenation
- Utilize destructuring for cleaner code
- Use async/await over Promise chains for readability

Good luck with your interview! Remember to practice these concepts with actual code examples.