Complete JavaScript Practice Guide

25 Programs — Basic to Advanced (with detailed explanations)

Prepared for: Rakshith HK

Contents

1. Print “Hello, World!”

```
console.log("Hello, World!");
```

Expected output: Hello, World!

Explanation: This is the simplest JS program to verify your environment. console.log writes text to the console, which is visible in browser DevTools or Node.js terminal. It demonstrates executing statements and string literals. Use this to confirm your runtime is working before moving to more complex code.

2. Sum of Two Numbers

```
let a = 5, b = 10;
console.log("Sum =", a + b);
```

Expected output: Sum = 15

Explanation: Shows basic variable declaration with let and arithmetic operation. The + operator adds numeric values; when used with strings it concatenates. console.log accepts multiple arguments separated by commas which get printed with spaces. Useful to learn expression evaluation and output.

3. Check if Number is Even or Odd

```
let num = 7;
if (num % 2 === 0)
  console.log("Even");
else
  console.log("Odd");
```

Expected output: Odd

Explanation: Demonstrates conditional statements and the modulus operator (%). num % 2 === 0 checks divisibility by 2. Use === for strict equality (no type coercion). If/else branches allow different execution paths. This pattern is common for basic numeric checks and branching logic.

4. Find the Largest of Three Numbers

```
let a = 10, b = 25, c = 5;
let largest = Math.max(a, b, c);
console.log("Largest:", largest);
```

Expected output: Largest: 25

Explanation: Uses Math.max to find the largest among arguments—an inbuilt utility. Alternatively, you could compare values with if statements. This teaches use of standard library functions. Passing multiple arguments to Math.max is concise and avoids manual comparisons.

5. Factorial of a Number

```
let num = 5, fact = 1;
for (let i = 1; i <= num; i++) {
  fact *= i;
}
console.log(`Factorial of ${num} is ${fact}`);
```

Expected output: Factorial of 5 is 120

Explanation: Illustrates loops (for) and compound assignment (*=). The loop multiplies consecutive integers up to num. Template literals (backticks) allow embedding expressions easily. Factorial problems reinforce loop control and arithmetic accumulation.

6. Check if String is Palindrome

```
let str = "madam";
let reversed = str.split("").reverse().join("");
console.log(str === reversed ? "Palindrome" : "Not Palindrome");
```

Expected output: Palindrome

Explanation: Uses string and array methods: split turns the string into an array of characters, reverse reverses the array, join reconstructs the string. The ternary operator provides concise conditional output. This demonstrates transforming data and comparing results for equality checks.

7. Reverse a Number

```
let num = 12345;
let rev = parseInt(num.toString().split("").reverse().join(""));
console.log("Reversed Number:", rev);
```

Expected output: Reversed Number: 54321

Explanation: Converts number to string to reuse string reversal technique, then parses the reversed string back to number with parseInt. Shows type conversion between number and string and composition of methods for solving problems.

8. Find Maximum in an Array

```
let arr = [5, 8, 2, 9, 1];
console.log("Max:", Math.max(...arr));
```

Expected output: Max: 9

Explanation: Uses the spread operator (...) to pass array elements as individual arguments to Math.max. This is more concise than looping; it demonstrates modern ES6 features and how to interact with built-in functions using spreads.

9. Sum of Array Elements

```
let arr = [1, 2, 3, 4];
let sum = arr.reduce((a, b) => a + b, 0);
console.log("Sum:", sum);
```

Expected output: Sum: 10

Explanation: Introduces Array.prototype.reduce which accumulates array values into a single result. The arrow function (a, b) => a + b is concise syntax for callbacks. reduce is powerful for aggregation, mapping, and folding tasks.

10. Count Vowels in a String

```
let str = "javascript";
let count = (str.match(/[aeiou]/gi) || []).length;
console.log("Vowel count:", count);
```

Expected output: Vowel count: 3

Explanation: Uses a regular expression to match vowels globally and case-insensitively. match returns null if no matches, so || [] avoids errors. Regex is efficient for pattern-based string processing; handling null results is important for robust code.

11. Check for Prime Number

```
function isPrime(num) {
  if (num < 2) return false;
  for (let i = 2; i <= Math.sqrt(num); i++) {
    if (num % i === 0) return false;
  }
  return true;
}
console.log(isPrime(11));
```

Expected output: true

Explanation: Uses trial division up to sqrt(num) for efficiency: if a number has a factor greater than sqrt it pairs with one smaller than sqrt. Early return improves performance. This is a classic algorithmic optimization illustrating complexity awareness.

12. Find Second Largest Number in Array

```
let arr = [10, 40, 20, 30];
arr.sort((a, b) => b - a);
console.log("Second Largest:", arr[1]);
```

Expected output: Second Largest: 30

Explanation: Sorts the array in descending order and picks the second element. Sorting has O(n log n) complexity; for huge arrays a single-pass scan tracking top two values is faster. This teaches tradeoffs between simplicity and performance.

13. Remove Duplicates from Array

```
let arr = [1, 2, 2, 3, 4, 4, 5];
let unique = [...new Set(arr)];
console.log(unique);
```

Expected output: [1, 2, 3, 4, 5]

Explanation: Uses the Set object which stores unique values and the spread operator to convert it back to an array. Sets are ideal for de-duplication and this is a concise ES6 approach compared to manual checks with loops.

14. Generate Fibonacci Series

```
let n = 7, a = 0, b = 1;
console.log(a, b);
for (let i = 2; i < n; i++) {
  let c = a + b;
  console.log(c);
  a = b;
  b = c;
}
```

Expected output: 0 1 1 2 3 5 8

Explanation: Generates the Fibonacci sequence iteratively using constant memory by keeping only the last two numbers. This demonstrates loop control, sequence generation, and updating multiple variables per iteration.

15. Find Sum of Digits of a Number

```
let num = 1234;
let sum = 0;
while (num > 0) {
  sum += num % 10;
  num = Math.floor(num / 10);
}
console.log("Sum of digits:", sum);
```

Expected output: Sum of digits: 10

Explanation: Extracts digits using modulus and integer division (Math.floor). This is a common pattern for digit- level numeric processing. Loop continues until the number is reduced to zero; integer division

removes the last digit each step.

16. Sort Objects by Property

```
let users = [
  { name: "John", age: 25 },
  { name: "Alice", age: 22 },
  { name: "Bob", age: 30 }
];
users.sort((a, b) => a.age - b.age);
console.log(users);
```

Expected output: [{name: 'Alice', age: 22}, {name: 'John', age: 25}, {name: 'Bob', age: 30}]

Explanation: Sorts an array of objects by a numeric property using a compare function. The compare returns negative/positive to determine order. This pattern is essential when working with structured data like records or API responses.

17. Async/Await Example (Fetch API)

```
async function getData() {
  let response = await fetch('https://jsonplaceholder.typicode.com/posts/1');
  let data = await response.json();
  console.log(data);
}
getData();
```

Expected output: (Displays a JSON post object in console)

Explanation: Shows asynchronous code using async/await for readable promise handling. await pauses until the Promise resolves without blocking the event loop. Useful for network calls or any IO. In browsers, fetch returns a Promise for the HTTP response.

18. Promise Example

```
let promise = new Promise((resolve, reject) => {
  let success = true;
  success ? resolve("Task Completed!") : reject("Error!");
});
promise.then(msg => console.log(msg))
       .catch(err => console.error(err));
```

Expected output: Task Completed!

Explanation: Demonstrates manual Promise creation and handling with then/catch. resolve and reject represent completion and failure. Promises are the foundation of modern async JS and understanding them is critical before using async/await.

19. Closure Example

```
function counter() {
  let count = 0;
```

```
  return function() {
    count++;
    return count;
  }
}
let add = counter();
console.log(add());
console.log(add());
```

Expected output: 1 2

Explanation: Closures allow an inner function to retain access to variables from its outer scope even after the outer function returns. This enables private state and factory functions. It's a powerful concept used for encapsulation and function factories.

20. DOM Manipulation (Simple)

```
<!-- HTML -->
<button onclick="changeText()">Click Me</button>
<p id="demo">Old Text</p>

<script>
  function changeText() {
    document.getElementById("demo").innerText = "New Text!";
  }
</script>
```

Expected output: (Paragraph text changes from 'Old Text' to 'New Text!' when button clicked)

Explanation: Demonstrates accessing and modifying DOM elements with getElementById and innerText. This shows how JS interacts with the page to create dynamic behavior. Event handlers (onclick) trigger code when users interact.

21. Local Storage Example

```
localStorage.setItem("username", "Rakshith");
console.log(localStorage.getItem("username"));
```

Expected output: Rakshith

Explanation: Uses the Web Storage API to persist simple key/value data in the browser. Data remains across page reloads but is origin-scoped. localStorage stores strings; use JSON.stringify/parse for objects. It's useful for caching user preferences.

22. Fetch and Display User Data

```
fetch('https://jsonplaceholder.typicode.com/users')
  .then(res => res.json())
  .then(data => data.forEach(u => console.log(u.name)));
```

Expected output: (Prints names of users from the API)

Explanation: Demonstrates fetching JSON from an API and handling the Promise chain. forEach iterates over the returned array of user objects. This mirrors real-world tasks like consuming REST APIs and

rendering received data.

23. Debounce Function

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

const log = debounce(() => console.log("Typing..."), 1000);
document.addEventListener("keyup", log);
```

Expected output: (Logs 'Typing...' 1s after user stops typing)

Explanation: Debouncing prevents a function from running too frequently by delaying execution until activity stops. It's essential for performance-sensitive handlers (resize, scroll, keyup). The returned function captures timeout via closure.

24. Deep Copy of Object

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

Expected output: { a: 1, b: { c: 2 } }

Explanation: Uses JSON serialization to create a deep copy for JSON-compatible objects. This breaks references to nested objects. Limitations: functions, undefined, Date, Map/Set won't serialize correctly. For complex objects use structuredClone or custom cloning.

25. Module Example (ES6 Import/Export)

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

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

Expected output: 5

Explanation: Shows ES6 modules with explicit exports and imports. Modules enable modular code organization and scope isolation. In browser use type="module" in script tag or use bundlers (Webpack, Vite) / Node.js ESM support.