Intermediate JavaScript Programming

LESSON 7: Writing Maintainable & Efficient Code

Learning Objectives:

By the end of this lesson, participants will be able to:

- Identify and refactor inefficient or repetitive code patterns.
- Recognize and avoid common JavaScript pitfalls.
- Make decisions balancing code readability with modern syntax features.
- Understand when browser compatibility should influence design choices.

Lesson Outline:

I. Identifying Repetition and Inefficiency (10 min)

Code repetition leads to more bugs and slower updates. Look for similar code blocks that can be abstracted into a function.

Example – Repetitive:

```
console.log("User: " + user.name);
console.log("Email: " + user.email);
console.log("Phone: " + user.phone);
```

Refactored:

```
function logUserField(label, value) {
  console.log(label + ": " + value);
}

logUserField("User", user.name);
logUserField("Email", user.email);
logUserField("Phone", user.phone);
```

Commentary:

Even small abstractions improve maintainability and reduce errors from copy/paste edits.

II. Avoiding Common JavaScript Pitfalls (15 min)

1. Implicit type coercion:

```
console.log("5" + 1);  // "51"
console.log("5" - 1);  // 4
```

Be cautious when combining different types. Use `===` instead of `==` to avoid coercion:

```
console.log(5 === "5"); // false
```

2. Accidental global variables:

```
function setFlag() {
  flag = true; // creates a global if 'let' or 'const' is missing
}
```

Always declare variables with `let` or `const`.

3. Using `var` instead of `let` or `const`:

```
for (var i = 0; i < 3; i++) {
  setTimeout(function() {
    console.log(i); // logs 3, 3, 3
  }, 100);
}
```

Better:

```
for (let i = 0; i < 3; i++) {
  setTimeout(function() {
    console.log(i); // logs 0, 1, 2
  }, 100);
}
```

Commentary:

Understanding scoping rules and avoiding implicit behavior can eliminate entire classes of bugs.

III. Balancing Modern Features with Readability (10 min)

Modern JavaScript includes concise features like arrow functions, destructuring, and optional chaining. Use them where they add clarity, not just to appear modern.

Example:

```
const user = data && data.profile && data.profile.email;
// vs.
const user = data?.profile?.email;
```

Commentary:

Optional chaining (`?.`) avoids runtime errors on missing properties, but if overused, it can obscure the fact that your data might be broken. Be intentional with use.

Avoid destructuring and shorthand when teaching or maintaining beginner-friendly codebases:

```
const { x, y } = point; // harder to read than:
const x = point.x;
const y = point.y;
```

IV. Writing for Future You (10 min)

You may not remember why you wrote something the way you did. Leave clues:

- Use descriptive names:

```
let d = new Date(); // unclear
let currentTimestamp = new Date(); // better
```

- Add comments where necessary — especially for edge cases, algorithms, or hacks.
- Follow conventions and use consistent formatting.

Commentary:

Write code for someone else — even if that someone else is just you in 6 months.

V. When Browser Compatibility Matters (10 min)

Not all JavaScript features work in older browsers. Optional chaining, `let`, `const`, and some array methods may fail in older versions of Internet Explorer or embedded web views.

Use a tool like Can I use to check compatibility. When targeting older systems:

- Avoid unsupported features.
- Use transpilers like Babel if needed.

Commentary:

Modern features should be used where appropriate, but only when you understand the runtime environment.

VI. Recap & Q&A (5 min)

- Refactoring repeated code.
- Avoiding subtle JavaScript bugs.
- Weighing readability vs. syntactic convenience.
- Considering environment constraints when writing code.

Final Multiple-Choice Question:

Which of the following helps prevent accidental global variables? A. Using `var` instead of `let` B. Declaring variables with `let` or `const` C. Omitting `use strict` D. Declaring variables in global scope

(Answer: B. Declaring variables with `let` or `const`)