Intermediate JavaScript Programming

LESSON 6: Error Handling & Debugging Best Practices

Learning Objectives:

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

- Understand JavaScript's built-in error types.
- Use `try`, `catch`, and `finally` to handle exceptions.
- Create and throw custom errors.
- Use console tools to inspect and debug code effectively.

Lesson Outline:

I. Common JavaScript Errors (10 min)

JavaScript has several built-in error types:

- **Error**: Base type for all errors.
- **TypeError**: A value is not the expected type (e.g. calling something that's not a function).
- **ReferenceError**: Refers to an undeclared or unavailable variable.
- **SyntaxError**: Code can't be parsed.

Example:

```
let result = null;
console.log(result.length); // TypeError: Cannot read property 'length' of
null
```

```
undeclaredVar + 5; // ReferenceError
```

Commentary:

These errors usually provide a message and a stack trace. Knowing what they mean can significantly reduce debugging time.

II. try-catch-finally (10 min)

Use `try` to run code that might fail, `catch` to handle the error, and optionally `finally` to run cleanup code.

```
try {
  let data = JSON.parse("{ invalid json }");
```

```
} catch (error) {
  console.error("Caught error:", error.message);
} finally {
  console.log("Done attempting to parse.");
}
```

Commentary:

Use `try/catch` around operations that might throw (e.g., parsing, file access, external API calls). Do not overuse — only catch errors you expect to handle meaningfully.

III. Throwing Custom Errors (10 min)

You can create your own error messages using `throw`. For structured control, define your own error classes.

```
function divide(a, b) {
  if (b === 0) {
    throw new Error("Cannot divide by zero.");
  }
  return a / b;
}
```

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

function checkName(name) {
  if (name.length < 2) {
    throw new ValidationError("Name is too short.");
  }
}
```

Commentary:

Throwing custom errors with specific names makes it easier to distinguish between expected and unexpected failures during debugging.

IV. Console Debugging Techniques (10 min)

The `console` object provides useful tools for debugging:

- `console.log()` – general output
- `console.error()` – highlight an error
- `console.warn()` – highlight a caution
- `console.table()` – display arrays/objects in tabular form
- `console.trace()` – print a stack trace from the current location

Example:

```
const people = [
  { name: "Greg", age: 65 },
  { name: "Sara", age: 29 }
];

console.table(people);
```

Commentary:

Don't leave `console.log` statements in production code. Use them liberally during development, but remove or guard them before deployment.

V. Tips for Diagnosing Problems (10 min)

- Reproduce the error consistently.
- Use stack traces to find where it broke.
- Isolate small parts of the program.
- Check types and undefined/null values.
- Comment out large blocks to narrow the issue.
- Use the browser or Node debugger (`debugger` keyword).

Example:

```
function process(value) {
  debugger; // triggers dev tools pause
  return value.toUpperCase();
}
```

Commentary:

Reading error messages carefully and stepping through execution with breakpoints saves hours of guesswork.

VI. Recap & Q&A (5 min)

- Types of common errors.

- `try/catch`, throwing errors, and custom error classes.
- Console and debugger tools for effective inspection.

Final Multiple-Choice Question:

Which of the following is a good use of `try/catch`? A. Wrapping every line of a script B. Handling expected parsing errors C. Preventing all runtime failures silently D. Avoiding writing tests

(Answer: B. Handling expected parsing errors)