Module 8: Error Handling and Debugging

JavaScript for Computer Science Students

Week 14

Module Overview

Learning Objectives

By the end of this module, students will be able to:

- Implement comprehensive error handling strategies
- Create custom error types and exceptions
- Use try-catch-finally blocks effectively
- Apply advanced debugging techniques
- Utilize browser developer tools for debugging
- Implement logging strategies for error tracking

Prerequisites

- Understanding of JavaScript functions and objects
- Familiarity with asynchronous programming (Promises, async/await)
- Basic knowledge of browser developer tools

Section 8.1: Error Handling Strategies

Types of Errors in JavaScript

1. Syntax Errors

- Occur during code parsing
- Prevent code execution
- Usually caught during development

```javascript
```

```javascript
// Syntax Error Example
function greetUser() {
  console.log("Hello World"  // Missing closing parenthesis
}
```

2. Runtime Errors

- Occur during code execution
- Can crash the application if not handled

```javascript
// Runtime Error Example
let user = null;
console.log(user.name); // TypeError: Cannot read property 'name' of null
```

3. Logic Errors

- Code runs but produces incorrect results
- Hardest to detect and debug

```javascript
// Logic Error Example
function calculateAge(birthYear) {
  return birthYear - new Date().getFullYear(); // Should be reversed
}
```

The Error Object

JavaScript's built-in Error object provides information about errors:

```javascript
try {
  throw new Error("Something went wrong!");
} catch (error) {
  console.log(error.name);    // "Error"
  console.log(error.message); // "Something went wrong!"
  console.log(error.stack);   // Stack trace
}
```

Built-in Error Types

Error Type	Description	Example
`Error`	Generic error	`new Error("Generic error")`
`SyntaxError`	Invalid syntax	`eval("invalid syntax")`
`TypeError`	Wrong data type	`null.someMethod()`
`ReferenceError`	Undefined variable	`console.log(undefinedVar)`
`RangeError`	Number out of range	`new Array(-1)`
`URIError`	Invalid URI	`decodeURI("%")`

Custom Error Classes

Creating Custom Errors

javascript

```javascript
class ValidationError extends Error {
  constructor(message, field) {
    super(message);
    this.name = 'ValidationError';
    this.field = field;
  }
}

class DatabaseError extends Error {
  constructor(message, query) {
    super(message);
    this.name = 'DatabaseError';
    this.query = query;
  }
}

class NetworkError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.name = 'NetworkError';
    this.statusCode = statusCode;
  }
}
```

Using Custom Errors

javascript

```
function validateUser(userData) {
  if (!userData.name || userData.name.trim().length < 2) {
    throw new ValidationError(
      'Name must be at least 2 characters',
      'name'
    );
  }

  if (!userData.email || !userData.email.includes('@')) {
    throw new ValidationError(
      'Valid email is required',
      'email'
    );
  }

  return true;
}
```

Try-Catch-Finally Blocks

Basic Structure

javascript

```
try {
  // Code that might throw an error
  let result = riskyOperation();
  console.log(result);
} catch (error) {
  // Handle the error
  console.error('An error occurred:', error.message);
} finally {
  // Always executes (cleanup code)
  console.log('Cleanup operations');
}
```

Handling Multiple Error Types

javascript

```javascript
async function processUserData(userData) {
  try {
    validateUser(userData);
    const user = await saveUserToDatabase(userData);
    return { success: true, user };
  } catch (error) {
    // Handle different error types differently
    switch (error.name) {
      case 'ValidationError':
        return {
          success: false,
          errorType: 'validation',
          message: error.message,
          field: error.field
        };
      case 'DatabaseError':
        return {
          success: false,
          errorType: 'database',
          message: 'Unable to save user data'
        };
      case 'NetworkError':
        return {
          success: false,
          errorType: 'network',
          message: 'Connection problem occurred'
        };
      default:
        return {
          success: false,
          errorType: 'unknown',
          message: 'An unexpected error occurred'
        };
    }
  }
}
```

Error Propagation

javascript

```javascript
function deepFunction() {
  throw new Error("Deep error");
}

function middleFunction() {
  deepFunction(); // Error propagates up
}

function topFunction() {
  try {
    middleFunction();
  } catch (error) {
    console.log("Caught error from deep function:", error.message);
  }
}
```

Comprehensive Error Handling Example

javascript

```
class StudentService {
  constructor() {
    this.students = [];
    this.errorLog = [];
  }

  validateStudentData(studentData) {
    const errors = [];

    if (!studentData.name || studentData.name.trim().length < 2) {
      errors.push(new ValidationError(
        'Name must be at least 2 characters',
        'name'
      ));
    }

    if (!studentData.email || !this.isValidEmail(studentData.email)) {
      errors.push(new ValidationError(
        'Valid email is required',
        'email'
      ));
    }

    if (studentData.gpa !== undefined &&
        (studentData.gpa < 0 || studentData.gpa > 4.0)) {
      errors.push(new ValidationError(
        'GPA must be between 0 and 4.0',
        'gpa'
      ));
    }

    if (errors.length > 0) {
      const error = new Error('Validation failed');
      error.name = 'MultipleValidationError';
      error.errors = errors;
      throw error;
    }

    return true;
  }

  async createStudent(studentData) {
    const operationId = Date.now();
```

```
  try {
    console.log(`Starting student creation (ID: ${operationId})`);

    // Validation
    this.validateStudentData(studentData);

    // Check for duplicates
    const existingStudent = this.students.find(
      s => s.email === studentData.email
    );
    if (existingStudent) {
      throw new ValidationError(
        'Student with this email already exists',
        'email'
      );
    }

    // Simulate database operation
    const student = await this.simulateDatabaseSave(studentData);
    this.students.push(student);

    console.log(`Student created successfully (ID: ${operationId})`);
    return { success: true, student };

  } catch (error) {
    // Comprehensive error logging
    this.logError(error, {
      operation: 'createStudent',
      operationId,
      studentData
    });

    // Return structured error response
    return this.handleError(error);
  } finally {
    console.log(`Student creation operation completed (ID: ${operationId})`);
  }
}

logError(error, context = {}) {
  const errorEntry = {
    timestamp: new Date().toISOString(),
    error: {
```

```
      name: error.name,
      message: error.message,
      stack: error.stack
    },
    context
  };

  // Add specific error properties
  if (error instanceof ValidationError) {
    errorEntry.error.field = error.field;
  } else if (error instanceof DatabaseError) {
    errorEntry.error.query = error.query;
  } else if (error instanceof NetworkError) {
    errorEntry.error.statusCode = error.statusCode;
  }

  this.errorLog.push(errorEntry);
  console.error('Error logged:', errorEntry);
}

handleError(error) {
  const baseResponse = { success: false };

  switch (error.name) {
    case 'ValidationError':
      return {
        ...baseResponse,
        errorType: 'validation',
        message: error.message,
        field: error.field,
        userMessage: 'Please check your input and try again.'
      };

    case 'MultipleValidationError':
      return {
        ...baseResponse,
        errorType: 'validation',
        message: 'Multiple validation errors occurred',
        errors: error.errors.map(e => ({
          field: e.field,
          message: e.message
        })),
        userMessage: 'Please correct the highlighted fields.'
      };
```

```
      case 'DatabaseError':
        return {
          ...baseResponse,
          errorType: 'database',
          message: 'Database operation failed',
          userMessage: 'Unable to save data. Please try again later.'
        };

      default:
        return {
          ...baseResponse,
          errorType: 'unknown',
          message: 'An unexpected error occurred',
          userMessage: 'Something went wrong. Please try again.'
        };
    }
  }
}
```

Section 8.2: Debugging Techniques

Console Debugging Methods

Basic Console Methods

javascript

```
console.log('Basic information');
console.info('Information message');
console.warn('Warning message');
console.error('Error message');
```

Advanced Console Methods

javascript

```javascript
// Grouping logs
console.group('User Operations');
console.log('Creating user...');
console.log('Validating data...');
console.groupEnd();

// Conditional logging
console.assert(2 + 2 === 4, 'Math still works!');
console.assert(2 + 2 === 5, 'This assertion will fail');

// Counting function calls
function processData() {
  console.count('processData called');
}

// Timing operations
console.time('Database Query');
// ... some operation
console.timeEnd('Database Query');

// Stack trace
console.trace('Current call stack');

// Table display
const users = [
  { name: 'Alice', age: 25, city: 'New York' },
  { name: 'Bob', age: 30, city: 'London' }
];
console.table(users);
```

Debugging Utilities Class

javascript

```
class DebuggingUtilities {
  constructor() {
    this.isDebugMode = true; // Toggle for production
    this.performanceMarks = new Map();
  }

  // Enhanced console logging
  log(level, message, data = null) {
    if (!this.isDebugMode && level === 'debug') return;

    const timestamp = new Date().toISOString();
    const styles = {
      error: 'color: red; font-weight: bold',
      warn: 'color: orange; font-weight: bold',
      info: 'color: blue',
      debug: 'color: green',
      success: 'color: green; font-weight: bold'
    };

    console.log(
      `%c[${timestamp}] ${level.toUpperCase()}: ${message}`,
      styles[level]
    );

    if (data) {
      console.table ? console.table(data) : console.log(data);
    }
  }

  // Performance measurement
  startPerformanceMeasure(label) {
    this.performanceMarks.set(label, performance.now());
    console.time(label);
  }

  endPerformanceMeasure(label) {
    const startTime = this.performanceMarks.get(label);
    if (startTime) {
      const duration = performance.now() - startTime;
      console.timeEnd(label);
      this.log('debug', `Performance: ${label} took ${duration.toFixed(2)}ms`);
      this.performanceMarks.delete(label);
      return duration;
```

```
    }
  }

  // Memory usage tracking
  checkMemoryUsage() {
    if (performance.memory) {
      const memory = performance.memory;
      this.log('debug', 'Memory Usage:', {
        used: `${(memory.usedJSHeapSize / 1048576).toFixed(2)} MB`,
        total: `${(memory.totalJSHeapSize / 1048576).toFixed(2)} MB`,
        limit: `${(memory.jsHeapSizeLimit / 1048576).toFixed(2)} MB`
      });
    }
  }

  // Function call tracing
  trace(func, funcName) {
    return (...args) => {
      this.log('debug', `Calling ${funcName}`, { arguments: args });
      try {
        const result = func(...args);
        this.log('debug', `${funcName} returned`, { result });
        return result;
      } catch (error) {
        this.log('error', `${funcName} threw error`, { error: error.message });
        throw error;
      }
    };
  }
}
```

Browser Developer Tools

Breakpoints

- **Line Breakpoints**: Click on line numbers
- **Conditional Breakpoints**: Right-click on line number
- **Exception Breakpoints**: Pause on caught/uncaught exceptions

Sources Panel Features

- **Call Stack**: Shows function call hierarchy
- **Scope Variables**: Inspect local and global variables

- **Watch Expressions**: Monitor specific variables or expressions

Network Tab

- Monitor API calls and responses
- Check for failed requests
- Analyze request/response headers

Performance Tab

- Profile JavaScript execution
- Identify performance bottlenecks
- Memory leak detection

Debugging Asynchronous Code

```javascript
```

```
class AsyncDebuggingExample {
  constructor() {
    this.debug = new DebuggingUtilities();
  }

  // Debugging Promise chains
  async debugPromiseChain() {
    console.group('Promise Chain Debugging');
    try {
      const step1 = await this.simulateAsyncStep('Step 1: Fetch user data')
        .then(data => {
          console.log('✓ Step 1 completed:', data);
          return data;
        })
        .catch(error => {
          console.error('✗ Step 1 failed:', error);
          throw error;
        });

      const step2 = await this.simulateAsyncStep('Step 2: Process user data')
        .then(data => {
          console.log('✓ Step 2 completed:', data);
          return { ...step1, processed: data };
        })
        .catch(error => {
          console.error('✗ Step 2 failed:', error);
          throw error;
        });

      console.log('🎉 All steps completed successfully:', step2);
      return step2;
    } catch (error) {
      console.error('💥 Promise chain failed:', error.message);
      throw error;
    } finally {
      console.groupEnd();
    }
  }

  // Debugging concurrent operations
  async debugConcurrentOperations() {
    console.group('Concurrent Operations Debugging');
    const operations = [
```

```
    { name: 'Operation A', delay: 1000 },
    { name: 'Operation B', delay: 1500 },
    { name: 'Operation C', delay: 800 }
  ];

  try {
    const startTime = Date.now();
    console.log('🚀 Starting concurrent operations...');

    const promises = operations.map(async (op, index) => {
      console.log(`📋 Starting ${op.name}`);
      const result = await this.simulateAsyncStep(op.name, op.delay);
      console.log(`✅ ${op.name} completed in ${Date.now() - startTime}ms`);
      return result;
    });

    const results = await Promise.allSettled(promises);

    const successful = results.filter(r => r.status === 'fulfilled');
    const failed = results.filter(r => r.status === 'rejected');

    console.log(`📊 Results: ${successful.length} successful, ${failed.length} failed`);

    if (failed.length > 0) {
      console.group('Failed Operations');
      failed.forEach((result, index) => {
        console.error(`❌ ${operations[index].name}:`, result.reason.message);
      });
      console.groupEnd();
    }

    return { successful: successful.length, failed: failed.length };
  } catch (error) {
    console.error('💥 Concurrent operations failed:', error);
    throw error;
  } finally {
    console.groupEnd();
  }
}

simulateAsyncStep(stepName, delay = 1000) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // 20% chance of failure for demonstration
```

```
        if (Math.random() > 0.8) {
          reject(new Error(`${stepName} failed randomly`));
        } else {
          resolve(`${stepName} result: ${Math.random().toString(36).substr(2, 5)}`);
        }
      }, delay);
    });
  }
}
```

Best Practices for Error Handling

1. Error Prevention

- **Input Validation**: Validate all user inputs
- **Type Checking**: Use TypeScript or runtime type checking
- **Defensive Programming**: Check for null/undefined values

2. Graceful Degradation

- Provide fallback functionality when errors occur
- Display user-friendly error messages
- Maintain application state consistency

3. Logging Strategy

- Log all errors with context information
- Use different log levels (error, warn, info, debug)
- Include timestamps and user session information
- Avoid logging sensitive information

4. Error Recovery

- Implement retry mechanisms for transient errors
- Use circuit breakers for external service calls
- Provide manual recovery options for users

5. Testing Error Scenarios

- Write unit tests for error conditions

- Test with invalid inputs and edge cases
- Simulate network failures and timeouts

Performance Debugging

Memory Leaks Detection

```javascript
```

```
class MemoryLeakDetector {
  constructor() {
    this.intervalId = null;
    this.memorySnapshots = [];
  }

  startMonitoring(interval = 5000) {
    this.intervalId = setInterval(() => {
      if (performance.memory) {
        const snapshot = {
          timestamp: Date.now(),
          usedJSHeapSize: performance.memory.usedJSHeapSize,
          totalJSHeapSize: performance.memory.totalJSHeapSize
        };
        this.memorySnapshots.push(snapshot);

        // Keep only last 20 snapshots
        if (this.memorySnapshots.length > 20) {
          this.memorySnapshots.shift();
        }

        this.analyzeMemoryTrend();
      }
    }, interval);
  }

  stopMonitoring() {
    if (this.intervalId) {
      clearInterval(this.intervalId);
      this.intervalId = null;
    }
  }

  analyzeMemoryTrend() {
    if (this.memorySnapshots.length < 5) return;

    const recent = this.memorySnapshots.slice(-5);
    const trend = recent.map(s => s.usedJSHeapSize);

    // Check if memory is consistently increasing
    let increasing = true;
    for (let i = 1; i < trend.length; i++) {
      if (trend[i] <= trend[i - 1]) {
```

```
        increasing = false;
        break;
      }
    }

    if (increasing) {
      console.warn('⚠️ Potential memory leak detected - memory consistently increasing');
      console.table(recent);
    }
  }
}
```

Performance Profiling

javascript

```
class PerformanceProfiler {
  constructor() {
    this.profiles = new Map();
  }

  profile(name, fn) {
    return async (...args) => {
      const startTime = performance.now();
      const startMemory = performance.memory?.usedJSHeapSize || 0;

      try {
        const result = await fn(...args);

        const endTime = performance.now();
        const endMemory = performance.memory?.usedJSHeapSize || 0;

        const profile = {
          name,
          duration: endTime - startTime,
          memoryDelta: endMemory - startMemory,
          timestamp: new Date().toISOString()
        };

        this.profiles.set(name, profile);

        console.log(`🔍 Profile [${name}]:`, {
          duration: `${profile.duration.toFixed(2)}ms`,
          memory: `${(profile.memoryDelta / 1024).toFixed(2)}KB`
        });

        return result;
      } catch (error) {
        console.error(`🔍 Profile [${name}] failed:`, error.message);
        throw error;
      }
    };
  }

  getProfiles() {
    return Array.from(this.profiles.values());
  }

  clearProfiles() {
```

```
    this.profiles.clear();
  }
}
```

Practical Exercise: Debug a Broken Application

Buggy Code Example

javascript

```
class BuggyStudentManager {
  constructor() {
    this.students = [];
  }

  // Bug 1: No input validation
  addStudent(name, email, gpa) {
    const student = {
      id: this.students.length,  // Bug 2: ID collision possible
      name: name.trim(),         // Bug 3: No null check
      email: email.toLowerCase(),
      gpa: parseFloat(gpa)
    };

    this.students.push(student);
    return student;
  }

  // Bug 4: Synchronous method name suggests async
  async findStudentByEmail(email) {
    for (let student of this.students) {
      if (student.email = email) {  // Bug 5: Assignment instead of comparison
        return student;
      }
    }
    return null;
  }

  // Bug 6: No error handling for division by zero
  calculateAverageGPA() {
    let total = 0;
    for (let student of this.students) {
      total += student.gpa;
    }
    return total / this.students.length;
  }

  // Bug 7: Modifies original array
  getTopStudents(count) {
    return this.students.sort((a, b) => b.gpa - a.gpa).slice(0, count);
  }
}
```

Debugging Process

1. **Identify the bugs** in the code above
2. **Add proper error handling**
3. **Implement input validation**
4. **Add debugging logs**
5. **Test with edge cases**

Assignment: Error Handling Implementation

Task

Create a robust error handling system for your student management application that includes:

1. **Custom Error Classes**
   - ValidationError
   - DatabaseError
   - NetworkError
2. **Comprehensive Error Handling**
   - Try-catch-finally blocks
   - Error propagation
   - Graceful degradation
3. **Debugging Utilities**
   - Logging system with different levels
   - Performance monitoring
   - Memory usage tracking
4. **Testing Error Scenarios**
   - Unit tests for error conditions
   - Integration tests with error simulation
   - Edge case handling

Deliverables

- Implementation of error handling classes
- Test cases demonstrating error scenarios

- Documentation of debugging procedures
- Performance analysis report

Key Takeaways

1. **Error handling is not optional** - It's essential for robust applications
2. **Custom error types** provide better error categorization and handling
3. **Proper logging** is crucial for debugging production issues
4. **Browser developer tools** are powerful debugging aids
5. **Performance monitoring** helps identify bottlenecks and memory leaks
6. **Testing error scenarios** is as important as testing happy paths
7. **User-friendly error messages** improve user experience
8. **Graceful degradation** keeps applications functional even when errors occur

---

Next Module Preview

Module 9: Testing JavaScript Code

- Unit testing frameworks
- Integration testing
- Test-driven development (TDD)
- Mocking and stubbing
- Code coverage analysis

Questions and Discussion

Common Questions

1. **When should I use custom error types vs. built-in Error?**
2. **How do I handle errors in Promise chains effectively?**
3. **What's the difference between development and production error handling?**
4. **How can I implement global error handling in web applications?**
5. **What are the best practices for error logging and monitoring?**

Discussion Topics

- Error handling strategies in different application architectures
- Performance impact of extensive error handling
- Security considerations in error messages
- Error handling in team development environments