Module 8: Error Handling and Debugging

Complete Presentation Materials

Course Information

- **Duration:** Week 14 (1 week intensive)
- **Prerequisites:** Modules 1-7 completed
- **Target:** Computer Science students ready for production-quality code
- **Focus:** Building robust, maintainable applications

Learning Objectives

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

1. Implement comprehensive error handling strategies
2. Create and use custom error types effectively
3. Use try-catch-finally blocks appropriately
4. Apply debugging techniques using browser developer tools
5. Implement logging strategies for production applications
6. Debug asynchronous code effectively
7. Detect and prevent memory leaks
8. Use performance profiling tools

Module Structure

8.1 Error Handling Strategies (Sessions 1-2)

- Error types and error objects
- Try-catch-finally blocks
- Custom error classes
- Error propagation patterns
- Graceful degradation techniques

8.2 Debugging Techniques (Sessions 3-4)

- Browser developer tools mastery
- Console debugging methods
- Breakpoint debugging strategies
- Performance profiling
- Memory leak detection

8.1 Error Handling Strategies

Understanding JavaScript Errors

Built-in Error Types

```javascript
// ReferenceError - variable not defined
console.log(unknownVariable); // ReferenceError: unknownVariable is not defined

// TypeError - wrong type operation
null.someMethod(); // TypeError: Cannot read property 'someMethod' of null

// SyntaxError - invalid syntax
eval('var a = }'); // SyntaxError: Unexpected token '}'

// RangeError - number out of range
new Array(-1); // RangeError: Invalid array length

// URIError - URI encoding/decoding error
decodeURI('%'); // URIError: URI malformed
```

Error Object Properties

```javascript
```

```
try {
  throw new Error("Something went wrong");
} catch (error) {
  console.log('Name:', error.name);           // "Error"
  console.log('Message:', error.message);     // "Something went wrong"
  console.log('Stack:', error.stack);         // Full stack trace
  console.log('File:', error.fileName);       // File where error occurred
  console.log('Line:', error.lineNumber);     // Line number
}
```

Custom Error Classes

Creating Specialized Errors

javascript

```
// Base custom error class
class AppError extends Error {
    constructor(message, statusCode = 500, isOperational = true) {
        super(message);
        this.name = this.constructor.name;
        this.statusCode = statusCode;
        this.isOperational = isOperational;

        // Maintain proper stack trace (V8 only)
        if (Error.captureStackTrace) {
            Error.captureStackTrace(this, this.constructor);
        }
    }
}

// Specific error types for university system
class ValidationError extends AppError {
    constructor(message, field = null) {
        super(message, 400);
        this.field = field;
        this.type = 'VALIDATION_ERROR';
    }
}

class AuthenticationError extends AppError {
    constructor(message = 'Authentication failed') {
        super(message, 401);
        this.type = 'AUTH_ERROR';
    }
}

class DatabaseError extends AppError {
    constructor(message, query = null) {
        super(message, 500);
        this.query = query;
        this.type = 'DATABASE_ERROR';
    }
}

class NetworkError extends AppError {
    constructor(message, url = null, statusCode = 503) {
        super(message, statusCode);
        this.url = url;
        this.type = 'NETWORK_ERROR';
    }
}

// Usage examples
function validateStudentData(data) {
    if (!data.name || data.name.trim().length < 2) {
        throw new ValidationError('Name must be at least 2 characters', 'name');
    }

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

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

Comprehensive Error Handling Patterns

The Robust Service Pattern

javascript

```
class StudentService {
  constructor() {
    this.students = new Map();
    this.errorLog = [];
    this.retryConfig = {
      maxAttempts: 3,
      baseDelay: 1000,
      backoffMultiplier: 2
    };
  }

  // Comprehensive error handling with logging
  async createStudent(studentData) {
    const operationId = this.generateOperationId();

    try {
      this.logOperation('createStudent', 'START', { operationId, data: studentData });

      // Input validation
      await this.validateStudentData(studentData);

      // Business logic validation
      await this.checkDuplicateStudent(studentData.email);

      // Create student with retry logic
      const student = await this.executeWithRetry(
        () => this.persistStudent(studentData),
        'persistStudent'
      );

      this.logOperation('createStudent', 'SUCCESS', { operationId, studentId: student.id });
      return { success: true, student };

    } catch (error) {
      this.handleError('createStudent', error, { operationId, studentData });
      return this.formatErrorResponse(error);
    }
  }

  // Retry mechanism with exponential backoff
  async executeWithRetry(operation, operationName) {
    const { maxAttempts, baseDelay, backoffMultiplier } = this.retryConfig;
    for (let attempt = 1; attempt <= maxAttempts; attempt++) {
      try {
        return await operation();
      } catch (error) {
        if (attempt === maxAttempts || !this.isRetryableError(error)) {
          throw error;
        }

        const delay = baseDelay * Math.pow(backoffMultiplier, attempt - 1);
        this.logOperation(operationName, 'RETRY', {
          attempt,
          nextDelay: delay,
          error: error.message
        });

        await this.sleep(delay);
      }
    }
  }

  // Error classification for retry logic
  isRetryableError(error) {
    const retryableTypes = ['NETWORK_ERROR', 'DATABASE_ERROR'];
    const retryableStatusCodes = [503, 504, 429];

    return retryableTypes.includes(error.type) ||
           retryableStatusCodes.includes(error.statusCode);
  }

  // Centralized error handling
  handleError(operation, error, context = {}) {
    const errorEntry = {
      timestamp: new Date().toISOString(),
      operation,
      error: {
        name: error.name,
        message: error.message,
        stack: error.stack,
        type: error.type || 'UNKNOWN',
        statusCode: error.statusCode || 500
      },
      context,
      severity: this.categorizeErrorSeverity(error)
    };
    this.errorLog.push(errorEntry);

    // Log to console in development
    if (process.env.NODE_ENV === 'development') {
      console.error('Operation failed:', errorEntry);
    }

    // Send to monitoring service in production
    if (errorEntry.severity === 'CRITICAL') {
      this.notifyErrorMonitoring(errorEntry);
    }
  }

  // Error severity classification
  categorizeErrorSeverity(error) {
    if (error instanceof ValidationError) return 'LOW';
    if (error instanceof AuthenticationError) return 'MEDIUM';
    if (error instanceof DatabaseError) return 'HIGH';
    if (error instanceof NetworkError) return 'MEDIUM';
    return 'CRITICAL';
  }

  // Structured error responses
  formatErrorResponse(error) {
    const baseResponse = { success: false };

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

    if (error instanceof AuthenticationError) {
      return {
        ...baseResponse,
        errorType: 'authentication',
        message: 'Authentication failed',
        userMessage: 'Please log in and try again.'
      };
    }

    if (error instanceof DatabaseError) {
        return {
            ...baseResponse,
            errorType: 'database',
            message: 'Data operation failed',
            userMessage: 'Unable to save data. Please try again later.'
        };
    }

    if (error instanceof NetworkError) {
        return {
            ...baseResponse,
            errorType: 'network',
            message: 'Network operation failed',
            statusCode: error.statusCode,
            userMessage: 'Connection problem. Please check your internet and try again.'
        };
    }

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

// Utility methods
generateOperationId() {
    return Date.now().toString(36) + Math.random().toString(36).substr(2);
}

sleep(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

logOperation(operation, status, details = {}) {
    console.log(`[${operation}] ${status}:`, details);
}

async validateStudentData(data) {
    // Simulate async validation
    if (!data.name || data.name.trim().length < 2) {
      throw new ValidationError('Name must be at least 2 characters', 'name');
    }

    if (!data.email || !this.isValidEmail(data.email)) {
      throw new ValidationError('Valid email is required', 'email');
    }
  }

  async checkDuplicateStudent(email) {
    if (this.students.has(email)) {
      throw new ValidationError('Student with this email already exists', 'email');
    }
  }

  async persistStudent(data) {
    // Simulate database operation with potential failure
    if (Math.random() > 0.7) {
      throw new DatabaseError('Failed to save student to database');
    }

    const student = {
      id: this.generateOperationId(),
      ...data,
      createdAt: new Date().toISOString()
    };

    this.students.set(data.email, student);
    return student;
  }

  isValidEmail(email) {
    return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);
  }

  notifyErrorMonitoring(errorEntry) {
    // In real application, send to monitoring service
    console.error('CRITICAL ERROR:', errorEntry);
  }
}
```

Error Boundary Patterns for Async Operations

Promise Chain Error Handling

javascript

```
class AsyncErrorHandler {
    static async handleAsyncOperation(operation, context = {}) {
        try {
            const result = await operation();
            return { success: true, data: result };
        } catch (error) {
            return this.handleAsyncError(error, context);
        }
    }

    static async handlePromiseChain(operations) {
        const results = [];

        for (const [index, operation] of operations.entries()) {
            try {
                const result = await operation();
                results.push({ success: true, data: result, index });
            } catch (error) {
                results.push({
                    success: false,
                    error: error.message,
                    index,
                    operation: operation.name || `Operation ${index}`
                });

                // Continue or break based on error type
                if (error instanceof ValidationError) {
                    break; // Stop on validation errors
                }
                // Continue on network errors
            }
        }

        return results;
    }

    static async handleConcurrentOperations(operations) {
        const results = await Promise.allSettled(operations);

        return results.map((result, index) => ({
            index,
            success: result.status === 'fulfilled',
            data: result.status === 'fulfilled' ? result.value : null,
      error: result.status === 'rejected' ? result.reason.message : null
    }));
  }

  static handleAsyncError(error, context) {
    const errorResponse = {
      success: false,
      error: {
        type: error.constructor.name,
        message: error.message,
        context
      }
    };

    // Add specific handling for different error types
    if (error instanceof NetworkError) {
      errorResponse.retryable = true;
      errorResponse.retryAfter = 5000;
    }

    return errorResponse;
  }
}

// Usage example
async function processStudentBatch(students) {
  const operations = students.map(student =>
    () => studentService.createStudent(student)
  );

  const results = await AsyncErrorHandler.handlePromiseChain(operations);

  const successful = results.filter(r => r.success);
  const failed = results.filter(r => !r.success);

  return {
    processed: successful.length,
    failed: failed.length,
    errors: failed.map(f => ({ index: f.index, error: f.error }))
  };
}
```

8.2 Debugging Techniques

Browser Developer Tools Mastery

Console Debugging Methods

javascript

```
class DebuggingUtilities {
    static setupConsoleHelpers() {
        // Enhanced logging with context
        window.debugLog = (level, message, data = null) => {
            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) {
                if (typeof data === 'object' && data !== null) {
                    console.table(data);
                } else {
                    console.log(data);
                }
            }
        };

        // Performance timing helper
        window.perfTimer = {
            timers: new Map(),

            start(label) {
                this.timers.set(label, performance.now());
                console.time(label);
            },

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

```
          return duration;
        }
      }
    };
  }

  // Function call tracing
  static trace(func, funcName) {
    return (...args) => {
      console.group(`🔍 Tracing: ${funcName}`);
      console.log('Arguments:', args);

      try {
        const result = func(...args);
        console.log('Return value:', result);
        console.groupEnd();
        return result;
      } catch (error) {
        console.error('Error thrown:', error);
        console.groupEnd();
        throw error;
      }
    };
  }

  // Async function debugging
  static async debugAsync(asyncFunc, funcName) {
    console.log(`🚀 Starting async operation: ${funcName}`);
    perfTimer.start(funcName);

    try {
      const result = await asyncFunc();
      console.log(`✅ Async operation completed: ${funcName}`);
      perfTimer.end(funcName);
      return result;
    } catch (error) {
      console.error(`❌ Async operation failed: ${funcName}`, error);
      perfTimer.end(funcName);
      throw error;
    }
  }

  // Memory usage tracking
  static checkMemoryUsage() {
```

```javascript
    if (performance.memory) {
      const memory = performance.memory;
      const used = Math.round(memory.usedJSHeapSize / 1048576);
      const total = Math.round(memory.totalJSHeapSize / 1048576);
      const limit = Math.round(memory.jsHeapSizeLimit / 1048576);

      console.log(`💾 Memory Usage: ${used}MB / ${total}MB (Limit: ${limit}MB)`);

      if (used / limit > 0.8) {
        console.warn('⚠️ High memory usage detected!');
      }

      return { used, total, limit };
    }
  }

  // DOM element inspector
  static inspectElement(selector) {
    const elements = document.querySelectorAll(selector);

    if (elements.length === 0) {
      console.warn(`No elements found for selector: ${selector}`);
      return;
    }

    console.group(`🔍 Inspecting elements: ${selector}`);
    elements.forEach((el, index) => {
      console.log(`Element ${index}:`, el);
      console.log('Computed styles:', getComputedStyle(el));
      console.log('Event listeners:', getEventListeners ? getEventListeners(el) : 'Use Chrome DevTools');
    });
    console.groupEnd();

    return elements;
  }
}

// Initialize debugging helpers
DebuggingUtilities.setupConsoleHelpers();
```

Advanced Breakpoint Strategies

javascript

```
class BreakpointDebugging {
  static conditionalDebugger(condition, context = {}) {
    if (condition) {
      console.log('Conditional breakpoint triggered:', context);
      debugger; // This will pause execution
    }
  }

  static logAndBreak(message, data = null) {
    console.log(`🛑 Debug point: ${message}`);
    if (data) console.log('Context:', data);
    debugger;
  }

  // Debug function calls with conditions
  static wrapWithDebug(func, condition, funcName = func.name) {
    return function(...args) {
      if (condition(...args)) {
        console.log(`🎯 Debug condition met for ${funcName}`);
        debugger;
      }
      return func.apply(this, args);
    };
  }

  // Async operation debugging
  static async debugAsyncFlow(asyncOperations, label = 'AsyncFlow') {
    console.group(`🔄 ${label} - Starting async operations`);

    for (const [index, operation] of asyncOperations.entries()) {
      try {
        console.log(`Step ${index + 1}: Starting ${operation.name || 'operation'}`);
        const result = await operation();
        console.log(`Step ${index + 1}: Completed successfully`);

        // Optional breakpoint after each step
        if (result?.debugBreak) {
          debugger;
        }
      } catch (error) {
        console.error(`Step ${index + 1}: Failed`, error);
        debugger; // Break on error
        throw error;
      }
    }

    console.groupEnd();
  }
}

// Usage examples
function debuggedStudentService() {
  const originalCreateStudent = StudentService.prototype.createStudent;

  // Debug when creating students with specific conditions
  StudentService.prototype.createStudent = BreakpointDebugging.wrapWithDebug(
    originalCreateStudent,
    (studentData) => studentData.email.includes('debug'),
    'createStudent'
  );
}
```

Performance Profiling and Optimization

Performance Monitoring Class

javascript

```
class PerformanceMonitor {
    constructor() {
        this.metrics = new Map();
        this.observers = [];
        this.setupObservers();
    }

    setupObservers() {
        // Performance Observer for monitoring various metrics
        if ('PerformanceObserver' in window) {
            // Monitor navigation timing
            const navObserver = new PerformanceObserver((list) => {
                for (const entry of list.getEntries()) {
                    this.recordMetric('navigation', {
                        type: entry.type,
                        loadEventEnd: entry.loadEventEnd,
                        domContentLoadedEventEnd: entry.domContentLoadedEventEnd,
                        responseEnd: entry.responseEnd
                    });
                }
            });
            navObserver.observe({ entryTypes: ['navigation'] });

            // Monitor resource loading
            const resourceObserver = new PerformanceObserver((list) => {
                for (const entry of list.getEntries()) {
                    if (entry.duration > 100) { // Only log slow resources
                        this.recordMetric('slow-resource', {
                            name: entry.name,
                            duration: entry.duration,
                            size: entry.transferSize
                        });
                    }
                }
            });
            resourceObserver.observe({ entryTypes: ['resource'] });

            // Monitor long tasks
            const longTaskObserver = new PerformanceObserver((list) => {
                for (const entry of list.getEntries()) {
                    this.recordMetric('long-task', {
                        duration: entry.duration,
                        startTime: entry.startTime
                });

                if (entry.duration > 50) {
                    console.warn(`⚠️ Long task detected: ${entry.duration}ms`);
                }
            }
        });
        longTaskObserver.observe({ entryTypes: ['longtask'] });
    }
}

// Measure function performance
measureFunction(func, funcName) {
    return (...args) => {
        const start = performance.now();
        const result = func(...args);
        const duration = performance.now() - start;

        this.recordMetric('function-performance', {
            name: funcName,
            duration,
            args: args.length
        });

        if (duration > 10) {
            console.warn(`🐌 Slow function: ${funcName} took ${duration.toFixed(2)}ms`);
        }

        return result;
    };
}

// Measure async function performance
measureAsyncFunction(asyncFunc, funcName) {
    return async (...args) => {
        const start = performance.now();
        try {
            const result = await asyncFunc(...args);
            const duration = performance.now() - start;

            this.recordMetric('async-function-performance', {
                name: funcName,
                duration,
                success: true
```

```
      });

      return result;
    } catch (error) {
      const duration = performance.now() - start;

      this.recordMetric('async-function-performance', {
        name: funcName,
        duration,
        success: false,
        error: error.message
      });

      throw error;
    }
  };
}

// Memory leak detection
detectMemoryLeaks() {
  if (!performance.memory) {
    console.warn('Memory monitoring not available in this browser');
    return;
  }

  const baseline = performance.memory.usedJSHeapSize;

  return {
    start: () => {
      this.memoryBaseline = baseline;
      console.log(`📊 Memory baseline: ${(baseline / 1048576).toFixed(2)}MB`);
    },

    check: (label = 'Memory Check') => {
      const current = performance.memory.usedJSHeapSize;
      const diff = current - this.memoryBaseline;
      const diffMB = (diff / 1048576).toFixed(2);

      console.log(`📊 ${label}: ${diffMB}MB change`);

      if (diff > 5242880) { // 5MB increase
        console.warn(`⚠️ Potential memory leak detected: +${diffMB}MB`);
      }
```

```
        return { current, baseline: this.memoryBaseline, diff };
      }
    };
  }

  // Record performance metrics
  recordMetric(type, data) {
    if (!this.metrics.has(type)) {
      this.metrics.set(type, []);
    }

    this.metrics.get(type).push({
      ...data,
      timestamp: Date.now()
    });

    // Keep only last 100 entries per type
    const entries = this.metrics.get(type);
    if (entries.length > 100) {
      entries.splice(0, entries.length - 100);
    }
  }

  // Generate performance report
  generateReport() {
    const report = {};

    for (const [type, entries] of this.metrics) {
      if (entries.length === 0) continue;

      const durations = entries
        .filter(e => e.duration !== undefined)
        .map(e => e.duration);

      if (durations.length > 0) {
        report[type] = {
          count: entries.length,
          avgDuration: durations.reduce((a, b) => a + b, 0) / durations.length,
          maxDuration: Math.max(...durations),
          minDuration: Math.min(...durations)
        };
      } else {
        report[type] = { count: entries.length };
      }
```

```
        }

        console.table(report);
        return report;
    }
}
```

Memory Leak Detection and Prevention

Common Memory Leak Patterns

javascript

```
class MemoryLeakDetector {
    constructor() {
        this.intervals = new Set();
        this.eventListeners = new WeakMap();
        this.domObservers = new Set();
    }

    // Safe interval management
    setInterval(callback, delay) {
        const id = setInterval(callback, delay);
        this.intervals.add(id);
        return id;
    }

    clearInterval(id) {
        clearInterval(id);
        this.intervals.delete(id);
    }

    clearAllIntervals() {
        for (const id of this.intervals) {
            clearInterval(id);
        }
        this.intervals.clear();
    }

    // Safe event listener management
    addEventListener(element, event, handler, options = {}) {
        element.addEventListener(event, handler, options);

        if (!this.eventListeners.has(element)) {
            this.eventListeners.set(element, new Map());
        }

        this.eventListeners.get(element).set(event, handler);
    }

    removeEventListener(element, event) {
        const handlers = this.eventListeners.get(element);
        if (handlers && handlers.has(event)) {
            const handler = handlers.get(event);
            element.removeEventListener(event, handler);
            handlers.delete(event);
    }
  }

  // Clean up all listeners for an element
  cleanupElement(element) {
    const handlers = this.eventListeners.get(element);
    if (handlers) {
      for (const [event, handler] of handlers) {
        element.removeEventListener(event, handler);
      }
      handlers.clear();
    }
  }

  // Detect potential memory leaks
  analyzeMemoryUsage() {
    if (!performance.memory) return null;

    const analysis = {
      intervals: this.intervals.size,
      domObservers: this.domObservers.size,
      memory: {
        used: Math.round(performance.memory.usedJSHeapSize / 1048576),
        total: Math.round(performance.memory.totalJSHeapSize / 1048576),
        limit: Math.round(performance.memory.jsHeapSizeLimit / 1048576)
      }
    };

    // Check for potential issues
    const warnings = [];

    if (this.intervals.size > 10) {
      warnings.push('High number of active intervals');
    }

    if (analysis.memory.used / analysis.memory.limit > 0.8) {
      warnings.push('High memory usage');
    }

    if (warnings.length > 0) {
      console.warn('Potential memory issues:', warnings);
    }

    return { analysis, warnings };
    }

    // Global cleanup
    cleanup() {
        this.clearAllIntervals();

        // Clear DOM observers
        for (const observer of this.domObservers) {
            observer.disconnect();
        }
        this.domObservers.clear();

        console.log('Memory leak detector cleanup completed');
    }
}

// Example of memory leak prevention in student management
class MemoryEfficientStudentService {
    constructor() {
        this.students = new Map();
        this.leakDetector = new MemoryLeakDetector();
        this.setupCleanupHandlers();
    }

    setupCleanupHandlers() {
        // Cleanup on page unload
        window.addEventListener('beforeunload', () => {
            this.cleanup();
        });

        // Periodic memory monitoring
        this.leakDetector.setInterval(() => {
            this.leakDetector.analyzeMemoryUsage();
        }, 30000); // Check every 30 seconds
    }

    addStudentWithCleanup(studentData) {
        const student = {
            ...studentData,
            id: this.generateId(),
            createdAt: Date.now()
        };

        this.students.set(student.id, student);
    // Set up automatic cleanup for old data
    this.leakDetector.setInterval(() => {
      if (Date.now() - student.createdAt > 3600000) { // 1 hour
        this.students.delete(student.id);
        console.log(`Auto-cleaned old student data: ${student.id}`);
      }
    }, 60000); // Check every minute

    return student;
  }

  cleanup() {
    this.students.clear();
    this.leakDetector.cleanup();
  }

  generateId() {
    return Date.now().toString(36) + Math.random().toString(36).substr(2);
  }
}
```

PowerPoint Slide Outlines

Slide Set 1: Error Handling Fundamentals (12 slides)

Slide 1: Module Overview

- Title: "Error Handling and Debugging Mastery"
- Learning objectives and importance
- Real-world impact of proper error handling

Slide 2: JavaScript Error Landscape

- Built-in error types with examples
- Error object anatomy
- Stack trace interpretation

Slide 3: The Cost of Poor Error Handling

- User experience impact

- Debugging time statistics
- Production failure examples

Slide 4: Custom Error Classes

- Extending the Error class
- Specialized error types for different scenarios
- Error categorization strategies

Slide 5: Try-Catch-Finally Mastery

- Proper exception handling flow
- When to use finally blocks
- Nested try-catch patterns

Slide 6: Error Propagation Strategies

- Fail fast vs graceful degradation
- Error boundaries in applications
- User-friendly error messages

Slide 7: Async Error Handling

- Promise rejection handling
- Async/await error patterns
- Error handling in concurrent operations

Slide 8: Logging and Monitoring

- Structured logging principles
- Error severity classification
- Production monitoring strategies

Slide 9: Retry and Recovery Patterns

- Exponential backoff algorithms
- Circuit breaker pattern
- Graceful service degradation

Slide 10: Error Handling Best Practices

- Input validation strategies
- Error message design principles
- Performance considerations

Slide 11: Real-World Error Scenarios

- Network failures
- Database errors
- User input validation
- Third-party service failures

Slide 12: Error Handling Summary

- Key patterns to remember
- Common antipatterns to avoid
- Next session preview

Slide Set 2: Debugging Techniques (15 slides)

Slide 13: Debugging Mindset

- Scientific approach to debugging
- Hypothesis-driven investigation
- Systematic vs random debugging

Slide 14: Browser DevTools Overview

- Console panel mastery
- Sources panel debugging
- Network panel analysis
- Performance panel profiling

Slide 15: Console Debugging Techniques

- Advanced console methods
- Conditional logging
- Performance timing
- Memory usage monitoring

Slide 16: Breakpoint Strategies

- Line breakpoints vs conditional breakpoints
- Exception breakpoints
- XHR/Fetch breakpoints
- Event listener breakpoints

Slide 17: Step-by-Step Debugging

- Step over vs step into vs step out
- Call stack navigation
- Variable inspection
- Watch expressions

Slide 18: Async Debugging Challenges

- Promise chain debugging
- Async/await debugging
- Event loop understanding
- Race condition detection

Slide 19: Performance Profiling

- CPU profiling techniques
- Memory heap snapshots
- Timeline analysis
- Bottleneck identification

Slide 20: Memory Leak Detection

- Common leak patterns
- Heap snapshot comparison
- Event listener cleanup
- DOM node retention

Slide 21: Network Debugging

- Request/response analysis
- CORS issue identification
- API error handling

- Performance optimization

Slide 22: Production Debugging

- Source map configuration
- Error tracking services
- Log aggregation
- Remote debugging techniques

Slide 23: Automated Debugging

- Unit test debugging
- Integration test debugging
- End-to-end test debugging
- CI/CD debugging strategies

Slide 24: Advanced Debugging Tools

- Browser extensions for debugging
- Node.js debugging
- Mobile debugging
- Cross-browser debugging

Slide 25: Debugging Workflow

- Issue reproduction steps
- Root cause analysis
- Fix validation
- Prevention strategies

Slide 26: Performance Optimization

- Code profiling results interpretation
- Memory optimization techniques
- Rendering performance
- Bundle size optimization

Slide 27: Module Summary and Next Steps

- Debugging skills assessment

- Continuous improvement strategies
- Advanced debugging resources

Live Coding Demonstrations

Demo 1: Building a Robust Error Handler (35 minutes)

Setup Phase (5 minutes)

```javascript
// Starting with basic error-prone code
async function fetchStudentData(studentId) {
    const response = await fetch(`/api/students/${studentId}`);
    const data = await response.json();
    return data;
}

function displayStudent(student) {
    document.getElementById('student-name').textContent = student.name;
    document.getElementById('student-email').textContent = student.email;
    document.getElementById('student-gpa').textContent = student.gpa;
}

// Basic usage - lots of potential failure points
fetchStudentData('123').then(displayStudent);
```

Error Identification Phase (10 minutes)

```javascript
```

```
// Identify potential failure points:
// 1. Network failures
// 2. Invalid student ID
// 3. Malformed JSON response
// 4. Missing DOM elements
// 5. Invalid student data structure

// Let's add basic error handling
async function fetchStudentDataV2(studentId) {
    try {
        const response = await fetch(`/api/students/${studentId}`);

        if (!response.ok) {
            throw new Error(`HTTP ${response.status}: ${response.statusText}`);
        }

        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Failed to fetch student data:', error);
        throw error; // Re-throw for upstream handling
    }
}

function displayStudentV2(student) {
    try {
        const nameEl = document.getElementById('student-name');
        const emailEl = document.getElementById('student-email');
        const gpaEl = document.getElementById('student-gpa');

        if (!nameEl || !emailEl || !gpaEl) {
            throw new Error('Required DOM elements not found');
        }

        nameEl.textContent = student.name || 'Unknown';
        emailEl.textContent = student.email || 'No email';
        gpaEl.textContent = student.gpa?.toFixed(2) || 'N/A';
    } catch (error) {
        console.error('Failed to display student:', error);
        // Show user-friendly error message
        showErrorMessage('Unable to display student information');
```

```
  }
}
```

Comprehensive Error Handling Implementation (15 minutes)

javascript

```
// Custom error classes for specific scenarios
class StudentAPIError extends Error {
    constructor(message, statusCode, studentId) {
        super(message);
        this.name = 'StudentAPIError';
        this.statusCode = statusCode;
        this.studentId = studentId;
    }
}

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

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

// Robust service with comprehensive error handling
class StudentService {
    constructor() {
        this.cache = new Map();
        this.retryAttempts = 3;
        this.retryDelay = 1000;
    }

    async fetchStudent(studentId, useCache = true) {
        // Input validation
        if (!studentId || typeof studentId !== 'string') {
            throw new ValidationError('Student ID must be a non-empty string', 'studentId');
        }

        // Check cache first
        if (useCache && this.cache.has(studentId)) {
            console.log(`Cache hit for student ${studentId}`);
```

```
      return this.cache.get(studentId);
    }

    // Fetch with retry logic
    let lastError;
    for (let attempt = 1; attempt <= this.retryAttempts; attempt++) {
      try {
        console.log(`Fetching student ${studentId}, attempt ${attempt}`);

        const controller = new AbortController();
        const timeoutId = setTimeout(() => controller.abort(), 10000); // 10 second timeout

        const response = await fetch(`/api/students/${studentId}`, {
          signal: controller.signal
        });

        clearTimeout(timeoutId);

        if (!response.ok) {
          if (response.status === 404) {
            throw new StudentAPIError(`Student ${studentId} not found`, 404, studentId);
          }

          if (response.status >= 500) {
            throw new StudentAPIError(`Server error: ${response.statusText}`, response.status, studentId);
          }

          throw new StudentAPIError(`Request failed: ${response.statusText}`, response.status, studentId);
        }

        const contentType = response.headers.get('content-type');
        if (!contentType?.includes('application/json')) {
          throw new StudentAPIError('Invalid response format', response.status, studentId);
        }

        const data = await response.json();

        // Validate response structure
        this.validateStudentData(data);

        // Cache successful response
        this.cache.set(studentId, data);

        return data;
        } catch (error) {
            lastError = error;

            // Don't retry for certain error types
            if (error instanceof ValidationError || error.statusCode === 404) {
                throw error;
            }

            // Don't retry on last attempt
            if (attempt === this.retryAttempts) {
                break;
            }

            // Exponential backoff
            const delay = this.retryDelay * Math.pow(2, attempt - 1);
            console.log(`Retrying in ${delay}ms...`);
            await this.sleep(delay);
        }
    }

    throw new StudentAPIError(
        `Failed to fetch student after ${this.retryAttempts} attempts: ${lastError.message}`,
        lastError.statusCode || 500,
        studentId
    );
}

validateStudentData(data) {
    const requiredFields = ['id', 'name', 'email'];

    for (const field of requiredFields) {
        if (!data[field]) {
            throw new ValidationError(`Missing required field: ${field}`, field);
        }
    }

    if (typeof data.gpa !== 'undefined') {
        const gpa = Number(data.gpa);
        if (isNaN(gpa) || gpa < 0 || gpa > 4.0) {
            throw new ValidationError('GPA must be a number between 0 and 4.0', 'gpa');
        }
        data.gpa = gpa; // Ensure numeric type
    }
    }

    sleep(ms) {
        return new Promise(resolve => setTimeout(resolve, ms));
    }
}

// UI Controller with error handling
class StudentUIController {
    constructor(studentService) {
        this.studentService = studentService;
        this.setupErrorDisplay();
    }

    async displayStudent(studentId) {
        try {
            this.showLoading(true);
            this.hideError();

            const student = await this.studentService.fetchStudent(studentId);
            this.renderStudent(student);

        } catch (error) {
            this.handleDisplayError(error);
        } finally {
            this.showLoading(false);
        }
    }

    renderStudent(student) {
        const container = document.getElementById('student-container');

        if (!container) {
            throw new UIError('Student container element not found', 'student-container');
        }

        try {
            container.innerHTML = `
                <div class="student-card">
                    <h3>${this.escapeHtml(student.name)}</h3>
                    <p><strong>ID:</strong> ${this.escapeHtml(student.id)}</p>
                    <p><strong>Email:</strong> ${this.escapeHtml(student.email)}</p>
                    <p><strong>GPA:</strong> ${student.gpa ? student.gpa.toFixed(2) : 'N/A'}</p>
                </div>
        `;
    } catch (error) {
        throw new UIError('Failed to render student data', 'student-container');
    }
}

handleDisplayError(error) {
    let userMessage = 'An unexpected error occurred';
    let shouldRetry = false;

    if (error instanceof StudentAPIError) {
        if (error.statusCode === 404) {
            userMessage = `Student ${error.studentId} not found`;
        } else if (error.statusCode >= 500) {
            userMessage = 'Server is temporarily unavailable. Please try again later.';
            shouldRetry = true;
        } else {
            userMessage = 'Unable to load student data';
            shouldRetry = true;
        }
    } else if (error instanceof ValidationError) {
        userMessage = `Invalid ${error.field}: ${error.message}`;
    } else if (error instanceof UIError) {
        userMessage = 'Display error occurred';
    }

    this.showError(userMessage, shouldRetry);

    // Log full error details for debugging
    console.error('Student display error:', {
        type: error.constructor.name,
        message: error.message,
        statusCode: error.statusCode,
        field: error.field,
        studentId: error.studentId,
        stack: error.stack
    });
}

setupErrorDisplay() {
    // Create error display elements if they don't exist
    if (!document.getElementById('error-container')) {
        const container = document.createElement('div');
        container.id = 'error-container';
```

```
            container.className = 'error-container hidden';
            document.body.appendChild(container);
        }

        if (!document.getElementById('loading-indicator')) {
            const loader = document.createElement('div');
            loader.id = 'loading-indicator';
            loader.className = 'loading-indicator hidden';
            loader.textContent = 'Loading...';
            document.body.appendChild(loader);
        }
    }

    showError(message, canRetry = false) {
        const container = document.getElementById('error-container');
        if (container) {
            container.innerHTML = `
                <div class="error-message">
                    <span class="error-text">${this.escapeHtml(message)}</span>
                    ${canRetry ? '<button class="retry-button">Retry</button>' : ''}
                    <button class="dismiss-error">×</button>
                </div>
            `;
            container.classList.remove('hidden');

            // Add event listeners
            const retryBtn = container.querySelector('.retry-button');
            const dismissBtn = container.querySelector('.dismiss-error');

            if (retryBtn) {
                retryBtn.addEventListener('click', () => {
                    // Trigger retry logic
                    this.hideError();
                });
            }

            if (dismissBtn) {
                dismissBtn.addEventListener('click', () => this.hideError());
            }
        }
    }

    hideError() {
        const container = document.getElementById('error-container');
        if (container) {
            container.classList.add('hidden');
        }
    }

    showLoading(show) {
        const loader = document.getElementById('loading-indicator');
        if (loader) {
            loader.classList.toggle('hidden', !show);
        }
    }

    escapeHtml(text) {
        const div = document.createElement('div');
        div.textContent = text;
        return div.innerHTML;
    }
}
```

Testing and Demonstration (5 minutes)

javascript

```javascript
// Initialize the robust system
const studentService = new StudentService();
const uiController = new StudentUIController(studentService);

// Test various scenarios
async function testErrorHandling() {
    console.log('Testing error handling scenarios...');

    // Valid student
    try {
        await uiController.displayStudent('valid-student-123');
    } catch (error) {
        console.log('Valid student test failed:', error);
    }

    // Invalid student ID
    try {
        await uiController.displayStudent('');
    } catch (error) {
        console.log('Empty ID test passed:', error.message);
    }

    // Non-existent student
    try {
        await uiController.displayStudent('non-existent-999');
    } catch (error) {
        console.log('Non-existent student test passed:', error.message);
    }
}

// Run tests
testErrorHandling();
```

Demo 2: Advanced Debugging Workshop (40 minutes)

Console Debugging Mastery (10 minutes)

javascript

```
// Set up advanced debugging utilities
class AdvancedDebugger {
    constructor() {
        this.logLevels = {
            ERROR: 0,
            WARN: 1,
            INFO: 2,
            DEBUG: 3
        };
        this.currentLevel = this.logLevels.DEBUG;
        this.performance = new Map();
    }

    log(level, message, data = null) {
        if (this.logLevels[level] <= this.currentLevel) {
            const timestamp = new Date().toISOString();
            const style = this.getLogStyle(level);

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

            if (data !== null) {
                if (Array.isArray(data) || (typeof data === 'object' && data !== null)) {
                    console.table(data);
                } else {
                    console.log(data);
                }
            }
        }
    }

    getLogStyle(level) {
        const styles = {
            ERROR: 'color: red; font-weight: bold; background: #ffebee;',
            WARN: 'color: orange; font-weight: bold; background: #fff3e0;',
            INFO: 'color: blue; background: #e3f2fd;',
            DEBUG: 'color: green; background: #e8f5e8;'
        };
        return styles[level] || '';
    }

    // Performance timing with nested operations
    startTimer(label) {
        const now = performance.now();
    this.performance.set(label, { start: now, children: [] });
    console.time(label);
    return label;
  }

  endTimer(label) {
    const timing = this.performance.get(label);
    if (timing) {
      const duration = performance.now() - timing.start;
      console.timeEnd(label);
      this.log('DEBUG', `Operation ${label} completed`, { duration: `${duration.toFixed(2)}ms` });
      return duration;
    }
  }

  // Conditional breakpoint helper
  breakIf(condition, context = {}) {
    if (condition) {
      console.log('Conditional breakpoint triggered:', context);
      debugger;
    }
  }

  // Function call tracing
  trace(obj, methodName) {
    const original = obj[methodName];
    const self = this;

    obj[methodName] = function(...args) {
      self.log('DEBUG', `Calling ${methodName}`, { args, thisContext: this });

      const result = original.apply(this, args);

      if (result instanceof Promise) {
        return result.then(value => {
          self.log('DEBUG', `${methodName} resolved`, { value });
          return value;
        }).catch(error => {
          self.log('ERROR', `${methodName} rejected`, { error: error.message });
          throw error;
        });
      } else {
        self.log('DEBUG', `${methodName} returned`, { result });
        return result;
```

```
            }
        };

        return obj;
    }
}

// Initialize debugger
window.debugger = new AdvancedDebugger();
```

Breakpoint Debugging Strategies (10 minutes)

javascript

```
// Problematic async function for debugging
async function processStudentBatch(students) {
    const results = [];

    for (const student of students) {
        try {
            // Conditional breakpoint: break when processing specific student
            debugger.breakIf(student.name.includes('Debug'), { student });

            const timer = debugger.startTimer(`process-${student.id}`);

            // Simulate complex processing
            const processed = await processIndividualStudent(student);

            debugger.endTimer(timer);

            results.push(processed);

            // Break if processing takes too long
            debugger.breakIf(debugger.endTimer(timer) > 1000, {
                student,
                duration: 'exceeded 1 second'
            });

        } catch (error) {
            debugger.log('ERROR', `Failed to process student ${student.id}`, error);
            // Break on errors for investigation
            debugger;
        }
    }

    return results;
}

async function processIndividualStudent(student) {
    // Random processing delay to simulate real work
    const delay = Math.random() * 2000;
    await new Promise(resolve => setTimeout(resolve, delay));

    // Simulate occasional failures
    if (Math.random() > 0.8) {
        throw new Error(`Processing failed for student ${student.id}`);
    }
    return {
        ...student,
        processed: true,
        processingTime: delay
    };
}

// Test data with debugging triggers
const testStudents = [
    { id: '001', name: 'Alice Johnson' },
    { id: '002', name: 'Debug Student' }, // This will trigger breakpoint
    { id: '003', name: 'Charlie Brown' },
    { id: '004', name: 'Diana Prince' }
];

// Enable tracing on the processing function
debugger.trace(window, 'processIndividualStudent');
```

Memory Debugging and Profiling (10 minutes)

javascript

```
// Memory leak detection utilities
class MemoryProfiler {
  constructor() {
    this.snapshots = [];
    this.leakDetectors = new Map();
  }

  takeSnapshot(label = `snapshot-${Date.now()}`) {
    if (!performance.memory) {
      console.warn('Memory profiling not available in this browser');
      return null;
    }

    const snapshot = {
      label,
      timestamp: Date.now(),
      used: performance.memory.usedJSHeapSize,
      total: performance.memory.totalJSHeapSize,
      limit: performance.memory.jsHeapSizeLimit
    };

    this.snapshots.push(snapshot);
    console.log(`Memory snapshot "${label}":`, this.formatMemory(snapshot));

    return snapshot;
  }

  compareSnapshots(label1, label2) {
    const snap1 = this.snapshots.find(s => s.label === label1);
    const snap2 = this.snapshots.find(s => s.label === label2);

    if (!snap1 || !snap2) {
      console.error('One or both snapshots not found');
      return null;
    }

    const diff = {
      timeDiff: snap2.timestamp - snap1.timestamp,
      memoryDiff: snap2.used - snap1.used,
      percentChange: ((snap2.used - snap1.used) / snap1.used * 100).toFixed(2)
    };

    console.log(`Memory comparison ${label1} → ${label2}:`, {
      ...diff,
      memoryDiffMB: (diff.memoryDiff / 1048576).toFixed(2) + 'MB'
    });

    if (Math.abs(diff.memoryDiff) > 5242880) { // 5MB
      console.warn('Significant memory change detected!');
    }

    return diff;
  }

  startLeakDetection(label) {
    const detector = {
      interval: setInterval(() => {
        this.takeSnapshot(`${label}-auto-${Date.now()}`);
      }, 10000), // Every 10 seconds
      startSnapshot: this.takeSnapshot(`${label}-start`)
    };

    this.leakDetectors.set(label, detector);
    console.log(`Leak detection started for: ${label}`);
  }

  stopLeakDetection(label) {
    const detector = this.leakDetectors.get(label);
    if (detector) {
      clearInterval(detector.interval);
      const endSnapshot = this.takeSnapshot(`${label}-end`);
      this.compareSnapshots(`${label}-start`, `${label}-end`);
      this.leakDetectors.delete(label);
    }
  }

  formatMemory(snapshot) {
    return {
      used: (snapshot.used / 1048576).toFixed(2) + 'MB',
      total: (snapshot.total / 1048576).toFixed(2) + 'MB',
      utilization: ((snapshot.used / snapshot.total) * 100).toFixed(1) + '%'
    };
  }

  generateReport() {
    const report = {
      totalSnapshots: this.snapshots.length,
            activeDetectors: this.leakDetectors.size,
            memoryTrend: this.calculateMemoryTrend()
        };

        console.table(this.snapshots.map(s => ({
            label: s.label,
            time: new Date(s.timestamp).toLocaleTimeString(),
            ...this.formatMemory(s)
        })));

        return report;
    }

    calculateMemoryTrend() {
        if (this.snapshots.length < 2) return 'insufficient data';

        const first = this.snapshots[0];
        const last = this.snapshots[this.snapshots.length - 1];
        const change = last.used - first.used;

        if (Math.abs(change) < 1048576) return 'stable'; // Less than 1MB
        return change > 0 ? 'increasing' : 'decreasing';
    }
}

// Memory leak test scenario
class MemoryLeakExample {
    constructor() {
        this.eventHandlers = new Map();
        this.intervals = new Set();
        this.domNodes = [];
    }

    // Potential memory leak: not cleaning up event listeners
    createLeakyComponent() {
        const div = document.createElement('div');
        div.textContent = 'Leaky Component';

        const handler = () => {
            console.log('Handler called');
        };

        // This creates a potential memory leak
        div.addEventListener('click', handler);
        this.eventHandlers.set(div, handler);

        document.body.appendChild(div);
        this.domNodes.push(div);

        return div;
    }

    // Proper cleanup
    cleanup() {
        // Clean up event listeners
        for (const [element, handler] of this.eventHandlers) {
            element.removeEventListener('click', handler);
        }
        this.eventHandlers.clear();

        // Remove DOM nodes
        this.domNodes.forEach(node => {
            if (node.parentNode) {
                node.parentNode.removeChild(node);
            }
        });
        this.domNodes = [];

        // Clear intervals
        this.intervals.forEach(id => clearInterval(id));
        this.intervals.clear();
    }
}

// Initialize memory profiler
const memoryProfiler = new MemoryProfiler();

// Test memory leak detection
async function testMemoryProfiling() {
    memoryProfiler.takeSnapshot('baseline');
    memoryProfiler.startLeakDetection('component-test');

    const leakyExample = new MemoryLeakExample();

    // Create components that might leak memory
    for (let i = 0; i < 100; i++) {
        leakyExample.createLeakyComponent();
```

```javascript
      if (i % 20 === 0) {
        memoryProfiler.takeSnapshot(`components-${i}`);
      }
    }

    // Wait a bit, then cleanup
    setTimeout(() => {
      memoryProfiler.takeSnapshot('before-cleanup');
      leakyExample.cleanup();

      setTimeout(() => {
        memoryProfiler.takeSnapshot('after-cleanup');
        memoryProfiler.stopLeakDetection('component-test');
        memoryProfiler.generateReport();
      }, 2000);
    }, 5000);
  }
```

Performance Profiling Demonstration (10 minutes)

javascript

```
// Performance bottleneck examples
class PerformanceTestSuite {
  constructor() {
    this.results = new Map();
  }

  // Inefficient DOM manipulation
  inefficientDOMUpdate(count = 1000) {
    const container = document.getElementById('test-container') || this.createTestContainer();

    console.time('Inefficient DOM Update');

    for (let i = 0; i < count; i++) {
      const div = document.createElement('div');
      div.textContent = `Item ${i}`;
      div.style.background = `hsl(${i % 360}, 50%, 50%)`;
      container.appendChild(div); // This causes reflow on each iteration
    }

    console.timeEnd('Inefficient DOM Update');
  }

  // Efficient DOM manipulation
  efficientDOMUpdate(count = 1000) {
    const container = document.getElementById('test-container') || this.createTestContainer();

    console.time('Efficient DOM Update');

    const fragment = document.createDocumentFragment();

    for (let i = 0; i < count; i++) {
      const div = document.createElement('div');
      div.textContent = `Item ${i}`;
      div.style.background = `hsl(${i % 360}, 50%, 50%)`;
      fragment.appendChild(div); // No reflow until final append
    }

    container.appendChild(fragment); // Single reflow

    console.timeEnd('Efficient DOM Update');
  }

  // Memory-intensive operation
```

```
memoryIntensiveOperation(size = 1000000) {
    console.time('Memory Intensive Operation');
    memoryProfiler.takeSnapshot('before-memory-op');

    const largeArray = new Array(size);
    for (let i = 0; i < size; i++) {
        largeArray[i] = {
            id: i,
            data: `item-${i}`,
            timestamp: Date.now(),
            random: Math.random()
        };
    }

    memoryProfiler.takeSnapshot('after-memory-op');

    // Process the array
    const processed = largeArray
        .filter(item => item.random > 0.5)
        .map(item => ({ ...item, processed: true }))
        .sort((a, b) => a.random - b.random);

    memoryProfiler.takeSnapshot('after-processing');
    console.timeEnd('Memory Intensive Operation');

    return processed;
}

// CPU-intensive operation
cpuIntensiveOperation(iterations = 100000) {
    console.time('CPU Intensive Operation');

    let result = 0;
    for (let i = 0; i < iterations; i++) {
        result += Math.sin(i) * Math.cos(i) * Math.tan(i);
    }

    console.timeEnd('CPU Intensive Operation');
    return result;
}

// Compare performance of different approaches
async comparePerformance() {
    const tests = [
      { name: 'Inefficient DOM', fn: () => this.inefficientDOMUpdate(100) },
      { name: 'Efficient DOM', fn: () => this.efficientDOMUpdate(100) },
      { name: 'Memory Intensive', fn: () => this.memoryIntensiveOperation(10000) },
      { name: 'CPU Intensive', fn: () => this.cpuIntensiveOperation(10000) }
    ];

    for (const test of tests) {
      console.group(`Performance Test: ${test
```