# Module 7: Modern JavaScript Features

# **Complete Presentation Materials**

#### **Course Information**

• **Duration**: Week 12-13 (2 weeks)

Prerequisites: Modules 1-6 completed

Target: Computer Science students with foundational JavaScript knowledge

# **Learning Objectives**

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

- 1. Use modern JavaScript syntax and features (ES6+)
- 2. Understand and implement destructuring and spread operators
- 3. Work effectively with template literals and modules
- 4. Apply advanced JavaScript concepts including symbols, iterators, and generators
- 5. Understand prototypal inheritance and modern class syntax
- 6. Implement WeakMap, WeakSet, and other advanced data structures

#### Module Structure

#### Week 12: ES6+ Features (7.1)

- Let, const, and block scope
- Arrow functions and template literals
- Destructuring assignment
- Spread and rest operators
- Default parameters and classes

# Week 13: Advanced Concepts (7.2)

- Prototypal inheritance
- Symbols and iterators
- Generators and async generators

- Proxy and Reflect
- WeakMap/WeakSet and RegExp

### 7.1 ES6+ Features

# Block Scope with let and const

The Problem with (var)

```
javascript

// Hoisting and function scope issues
console.log(x); // undefined (not error!)

var x = 5;

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

# Modern Solutions: (let) and (const)

```
javascript

// Block scoped variables
{
    let blockScoped = "I'm trapped in this block";
    const alsoBlockScoped = "Me too!";
}

// console.log(blockScoped); // ReferenceError

// Temporal Dead Zone
    console.log(y); // ReferenceError: Cannot access 'y' before initialization
let y = 10;

// Loop fix with let
for (let i = 0; i < 3; i++) {
    setTimeout(() => console.log(i), 100); // Prints 0, 1, 2
}
```

#### **Best Practices**

• Use (const) by default

- Use let when you need to reassign
- Avoid (var) in modern JavaScript

#### **Arrow Functions**

### **Syntax Evolution**

```
javascript
// Traditional function
function multiply(a, b) {
  return a * b;
// Function expression
const multiply = function(a, b) {
  return a * b;
};
// Arrow function - concise
const multiply = (a, b) => a * b;
// Arrow function - with body
const multiply = (a, b) => {
  console.log('Multiplying', a, 'and', b);
  return a * b;
};
// Single parameter (no parentheses needed)
const square = x = > x * x;
// No parameters
const getRandom = () => Math.random();
```

# (this) Binding Differences

```
// Traditional function - dynamic this
const obj = {
  name: 'Alice',
  greet: function() {
     setTimeout(function() {
        console.log('Hello, ' + this.name); // undefined
     }, 1000);
  }
};
// Arrow function - lexical this
const obj = {
  name: 'Alice',
  greet: function() {
     setTimeout(() => {
        console.log('Hello, ' + this.name); // Alice
     }, 1000);
};
```

# **Template Literals**

### **String Interpolation**

```
javascript

const name = 'Alice';

const age = 25;

// Old way - concatenation

const message = 'Hello, my name is ' + name + ' and I am ' + age + ' years old.';

// New way - template literals

const message = `Hello, my name is ${name} and I am ${age} years old.`;

// Expressions in templates

const price = 19.99;

const tax = 0.08;

const total = `Total: $${(price * (1 + tax)).toFixed(2)}`;
```

# **Multi-line Strings**

### **Tagged Template Literals**

```
javascript

function highlight(strings, ...values) {
   return strings.reduce((result, string, i) => {
      const value = values[i] ? `<mark>${values[i]}</mark>`: ";
      return result + string + value;
      }, '');
}

const name = 'JavaScript';
const year = 2024;
const message = highlight`Learning ${name} in ${year} is exciting!`;
// Result: "Learning <mark>JavaScript</mark> in <mark>2024</mark> is exciting!"
```

# **Destructuring Assignment**

# **Array Destructuring**

javascript			

```
// Basic array destructuring
const colors = ['red', 'green', 'blue'];
const [primary, secondary, tertiary] = colors;

// Skipping elements
const [first, , third] = colors;

// Default values
const [a, b, c, d = 'yellow'] = colors;

// Rest elements
const [head, ...tail] = colors;

// Swapping variables
let x = 1, y = 2;
[x, y] = [y, x];
```

# **Object Destructuring**



```
const student = {
  name: 'Alice',
  age: 20,
  major: 'Computer Science',
  gpa: 3.8
// Basic destructuring
const { name, age } = student;
// Renaming variables
const { name: studentName, age: studentAge } = student;
// Default values
const { name, age, year = 'Sophomore' } = student;
// Nested destructuring
const course = {
  title: 'JavaScript',
  instructor: {
     name: 'Dr. Smith',
     email: 'smith@university.edu'
};
const { instructor: { name: instructorName } } = course;
```

### **Function Parameter Destructuring**

```
javascript
```

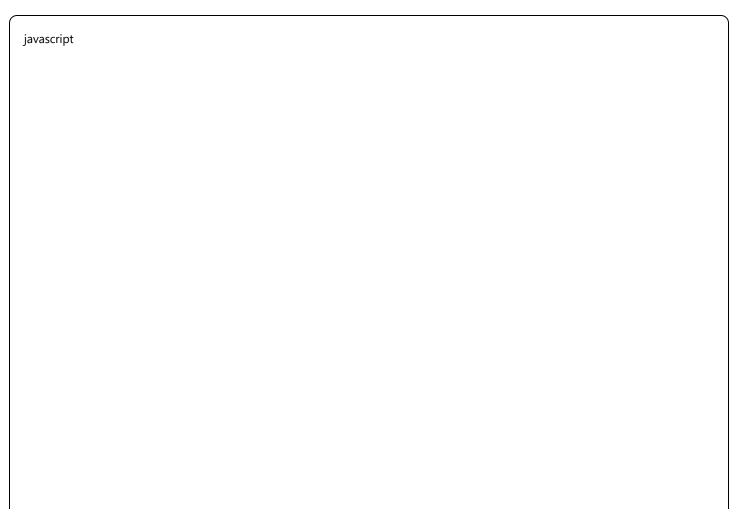
```
// Instead of this
function createStudent(options) {
   const name = options.name;
   const age = options.major || 'Undeclared';
   // ...
}

// Use this
function createStudent({ name, age, major = 'Undeclared' }) {
   console.log('Creating student: ${name}, ${age}, ${major}');
}

// Array parameter destructuring
function processCoordinates([x, y, z = 0]) {
   return { x, y, z };
}
```

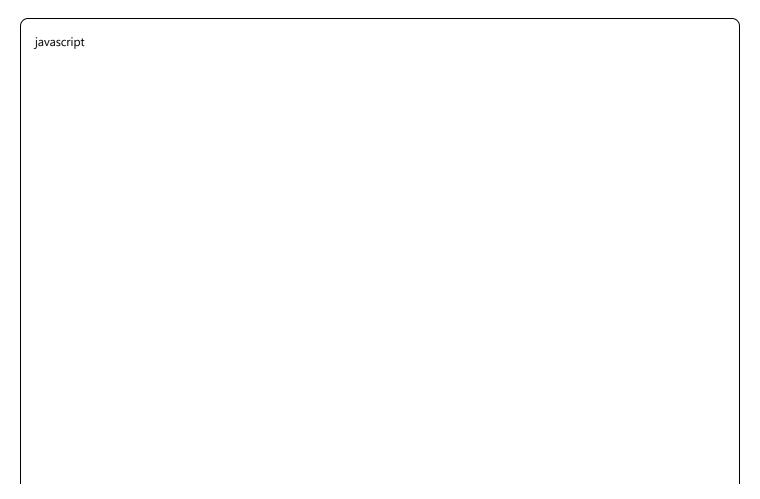
# **Spread and Rest Operators**

# Spread Operator (...)



```
// Array spreading
const fruits = ['apple', 'banana'];
const vegetables = ['carrot', 'broccoli'];
const food = [...fruits, ...vegetables, 'cheese'];
// Object spreading
const baseConfig = { debug: true, version: '1.0' };
const userConfig = { theme: 'dark', debug: false };
const finalConfig = { ...baseConfig, ...userConfig };
// Function calls
const numbers = [1, 2, 3, 4, 5];
const max = Math.max(...numbers);
// Copying arrays/objects
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];
const originalObject = { a: 1, b: 2 };
const copiedObject = { ...originalObject };
```

#### **Rest Parameters**



```
// Collecting multiple arguments
function sum(...numbers) {
    return numbers.reduce((total, num) => total + num, 0);
}

sum(1, 2, 3, 4, 5); // 15

// Mixed parameters
function greet(greeting, ...names) {
    return '${greeting} ${names.join(', ')}!';
}

greet('Hello', 'Alice', 'Bob', 'Charlie');
// "Hello Alice, Bob, Charlie!"

// Rest in destructuring
const [first, second, ...rest] = [1, 2, 3, 4, 5];
const { name, ...otherProps } = { name: 'Alice', age: 25, major: 'CS' };
```

# **Enhanced Object Literals**

### **Property Shorthand**

```
javascript

const name = 'Alice';
const age = 25;

// Old way
const student = {
    name: name,
    age: age
};

// New way
const student = { name, age };
```

#### **Method Shorthand**

```
javascript
```

```
// Old way
const calculator = {
   add: function(a, b) { return a + b; },
   multiply: function(a, b) { return a * b; }
};

// New way
const calculator = {
   add(a, b) { return a + b; },
   multiply(a, b) { return a * b; }
};
```

# **Computed Property Names**

```
javascript

const propertyName = 'score';

const student = {
    name: 'Alice',
    [propertyName]: 95,
    ['is' + 'Active']: true
};
```

### Classes

### **Class Declaration**

```
class Student {
  constructor(name, major) {
     this.name = name;
     this.major = major;
    this.courses = [];
  // Method
  enroll(course) {
     this.courses.push(course);
     return this;
  // Getter
  get courseCount() {
     return this.courses.length;
  // Setter
  set gpa(value) {
    if (value < 0 || value > 4.0) {
       throw new Error('GPA must be between 0 and 4.0');
     this._gpa = value;
  // Static method
  static compareGPA(student1, student2) {
     return student2._gpa - student1._gpa;
const alice = new Student('Alice', 'Computer Science');
alice.enroll('JavaScript').enroll('Data Structures');
console.log(alice.courseCount); // 2
```

#### **Class Inheritance**

```
class Person {
  constructor(name, age) {
     this.name = name;
     this.age = age;
  introduce() {
     return `Hi, I'm ${this.name}`;
class Student extends Person {
  constructor(name, age, major) {
     super(name, age);
     this.major = major;
  introduce() {
     return `$(super.introduce()), majoring in $(this.major)`;
const alice = new Student('Alice', 20, 'Computer Science');
console.log(alice.introduce()); // "Hi, I'm Alice, majoring in Computer Science"
```

# 7.2 Advanced JavaScript Concepts

# **Prototypal Inheritance**

**Understanding the Prototype Chain** 

```
// Every object has a prototype
const obj = {};
console.log(Object.getPrototypeOf(obj) === Object.prototype); // true

// Functions have prototypes too
function Student(name) {
    this.name = name;
}

Student.prototype.introduce = function() {
    return `Hi, I'm ${this.name}`;
};

const alice = new Student('Alice');
console.log(alice.introduce()); // "Hi, I'm Alice"

// Prototype chain lookup
console.log(alice._proto_ === Student.prototype); // true
console.log(Student.prototype._proto_ === Object.prototype); // true
```

### Creating Objects with Object.create()

```
javascript

const studentPrototype = {
  introduce() {
    return `Hi, I'm ${this.name}`;
  },

enroll(course) {
    this.courses = this.courses || [];
    this.courses.push(course);
  }
};

const alice = Object.create(studentPrototype);
alice.name = 'Alice';
alice.major = 'Computer Science';
```

# **Symbols**

# **Creating and Using Symbols**

```
javascript
// Creating symbols
const id = Symbol('id');
const id2 = Symbol('id');
console.log(id === id2); // false - symbols are unique

// Using symbols as property keys
const user = {
    name: 'Alice',
    [id]: 123
};

console.log(user[id]); // 123
console.log(user.id); // undefined

// Symbols are hidden from normal enumeration
console.log(Object.keys(user)); // ['name']
console.log(Object.getOwnPropertySymbols(user)); // [Symbol(id)]
```

### Well-known Symbols

```
javascript
// Symbol.iterator
const numbers = [1, 2, 3];
const iterator = numbers[Symbol.iterator]();
console.log(iterator.next()); // { value: 1, done: false }
// Symbol.toPrimitive
const student = {
  name: 'Alice',
  gpa: 3.8,
  [Symbol.toPrimitive](hint) {
     if (hint === 'number') return this.gpa;
     if (hint === 'string') return this.name;
     return this.name:
};
console.log(+student); // 3.8
console.log(`${student}`); // "Alice"
```

### **Iterators and Generators**

### **Custom Iterators**

```
javascript
// Creating an iterable object
const range = {
  start: 1,
  end: 5,
  [Symbol.iterator]() {
     let current = this.start;
     const end = this.end;
     return {
        next() {
          if (current <= end) {</pre>
             return { value: current++, done: false };
          } else {
             return { done: true };
     };
};
for (const num of range) {
  console.log(num); // 1, 2, 3, 4, 5
```

#### **Generator Functions**

```
javascript
```

```
// Simple generator
function* numberGenerator() {
  yield 1;
  yield 2;
  yield 3;
const gen = numberGenerator();
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
// Generator with parameters and logic
function* fibonacci(n) {
  let a = 0, b = 1;
  for (let i = 0; i < n; i++) {
     yield a;
     [a, b] = [b, a + b];
const fib = [...fibonacci(10)];
console.log(fib); // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
// Infinite generators
function* idGenerator() {
  let id = 1;
  while (true) {
     yield id++;
```

# **Async Generators**

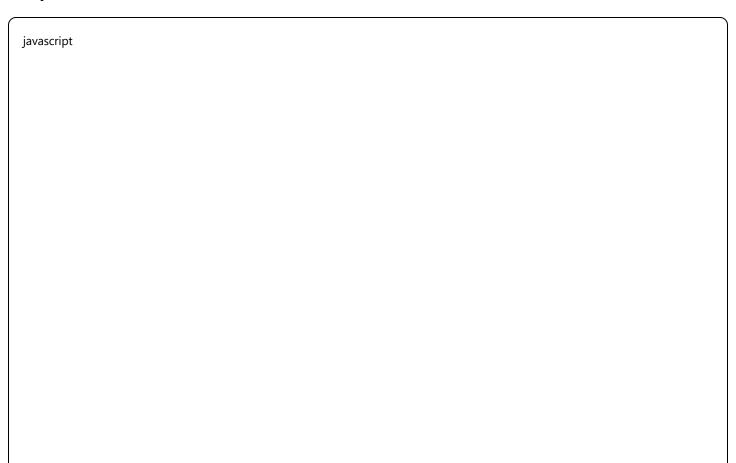
```
async function* dataFetcher(urls) {
  for (const url of urls) {
    try {
      const response = await fetch(url);
      const data = await response.json();
      yield data;
    } catch (error) {
      yield { error: error.message };
    }
}

// Usage
const urls = ['api/students', 'api/courses', 'api/grades'];
const fetcher = dataFetcher(urls);

for await (const data of fetcher) {
      console.log(data);
    }
}
```

# **Proxy and Reflect**

# **Proxy Basics**



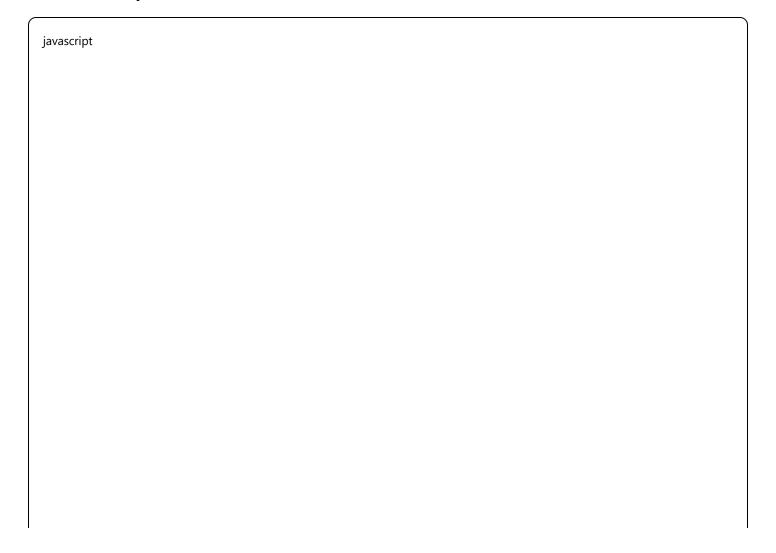
```
// Creating a proxy for property access logging
const student = { name: 'Alice', age: 20 };

const proxy = new Proxy(student, {
    get(target, prop) {
        console.log(`Getting property: ${prop}`);
        return target[prop];
    },

set(target, prop, value) {
        console.log(`Setting property: ${prop} = ${value}`);
        target[prop] = value;
        return true;
    }
});

proxy.name; // "Getting property: name"
proxy.gpa = 3.8; // "Setting property: gpa = 3.8"
```

# **Advanced Proxy Use Cases**



```
// Validation proxy
function createValidatedUser(validations) {
  return new Proxy({}), {
     set(target, prop, value) {
        const validator = validations[prop];
       if (validator && !validator(value)) {
          throw new Error(`Invalid value for ${prop}`);
       target[prop] = value;
       return true;
  });
const user = createValidatedUser({
  age: value => typeof value === 'number' && value > 0,
  email: value => typeof value === 'string' && value.includes('@')
});
// Array-like object proxy
function createArrayLike() {
  return new Proxy({}, {
     get(target, prop) {
       if (prop === 'length') {
          return Object.keys(target).length;
        return target[prop];
     },
     set(target, prop, value) {
       target[prop] = value;
        return true;
  });
```

# WeakMap and WeakSet

# WeakMap Usage

```
// Private data with WeakMap
const privateData = new WeakMap();
class Student {
  constructor(name, ssn) {
     this.name = name:
     // Store sensitive data privately
     privateData.set(this, { ssn });
  getSSN() {
     return privateData.get(this).ssn;
const alice = new Student('Alice', '123-45-6789');
console.log(alice.getSSN()); // "123-45-6789"
// alice.ssn is undefined - no direct access
// Automatic garbage collection
let obj = {};
const weakMap = new WeakMap();
weakMap.set(obj, 'some data');
obj = null; // Object can be garbage collected
```

#### WeakSet Usage

```
javascript

// Tracking objects without preventing garbage collection
const processedStudents = new WeakSet();

function processStudent(student) {
    if (processedStudents.has(student)) {
        console.log('Student already processed');
        return;
    }

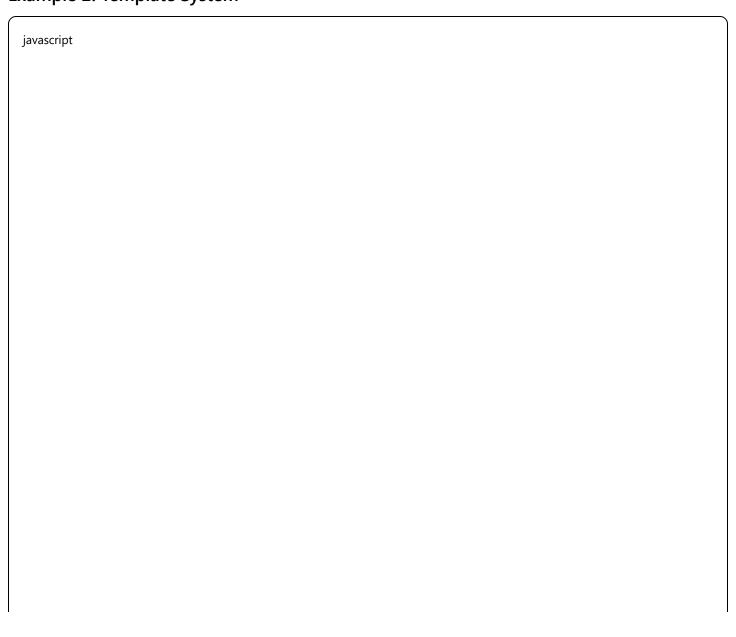
    // Process the student
    console.log('Processing ${student.name}');
    processedStudents.add(student);
}
```

# **Live Coding Examples**

javascript			

```
class CourseManager {
  constructor() {
     this.courses = new Map();
     this.students = new Map();
    // Use Symbol for private methods
     this[Symbol.for('privateData')] = new WeakMap();
  }
  // Using modern syntax throughout
  addCourse({ name, code, credits, instructor, capacity = 30 }) {
     if (this.courses.has(code)) {
       throw new Error('Course already exists');
     const course = {
       id: `${code}-${Date.now()}`,
       name.
       code,
       credits.
       instructor,
       capacity,
       enrolled: [],
       createdAt: new Date().tolSOString()
     }:
     this.courses.set(code, course);
     return course;
  // Generator for batch operations
  *processBatchEnrollments(enrollmentData) {
     for (const enrollment of enrollmentData) {
       try {
          const result = this.enrollStudent(enrollment);
          yield { success: true, enrollment: result };
       } catch (error) {
          yield { success: false, enrollment, error: error.message };
  // Using destructuring and spread
  getCourseStatistics() {
```

# **Example 2: Template System**



```
// Advanced template processing
class TemplateEngine {
  constructor() {
     this.templates = new Map();
     this.helpers = new Map();
  }
  // Tagged template for HTML
  html(strings, ...values) {
     return strings.reduce((result, string, i) => {
       const value = values[i] || ";
       const escaped = this.escapeHtml(value);
       return result + string + escaped;
    }, ");
  // Template compilation with destructuring
  compile(template, { escapeHtml = true, helpers = {} } = {}) {
     return (data) => {
       // Use destructuring in template processing
       return template.replace((\{(\w+)\})\, (match, key) => {
          const value = data[key];
          return escapeHtml ? this.escapeHtml(value) : value;
       });
     };
  escapeHtml(text) {
     const map = {
       '&': '&',
       '<': '&lt;',
       '>': '>',
       "": '"',
       "": '''
     };
     return text.replace(/[&<>"']/g, m => map[m]);
  }
```

### **Interactive Exercises**

# **Exercise 1: Destructuring Challenge**

```
javascript
// Given this complex data structure:
const university = {
  name: 'Tech University',
  location: { city: 'San Francisco', state: 'CA' },
  departments: [
        name: 'Computer Science',
       head: { name: 'Dr. Smith', email: 'smith@tech.edu' },
        courses: ['CS101', 'CS201', 'CS301']
        name: 'Mathematics',
        head: { name: 'Prof. Johnson', email: 'johnson@tech.edu' },
        courses: ['MATH101', 'MATH201']
// Extract the following using destructuring:
// 1. University name and city
// 2. Computer Science department head's name and email
// 3. All course codes from all departments
// 4. First course from each department
// Solution:
const {
  name: universityName,
  location: { city },
  departments: [
     { head: { name: csHeadName, email: csHeadEmail }, courses: csCourses },
     { courses: mathCourses }
} = university;
const allCourses = [...csCourses, ...mathCourses];
const firstCourses = [csCourses[0], mathCourses[0]];
```

### **Exercise 2: Generator Challenge**

```
javascript

// Create a generator that produces student registration numbers

// Format: YYYY-DEPT-NNN (e.g., 2024-CS-001)

function* registrationGenerator(department, year = new Date().getFullYear()) {

    let counter = 1;

    while (true) {

        const paddedCounter = counter.toString().padStart(3, '0');

        yield `${year}-${department}-${paddedCounter}`;

        counter++;

    }

}

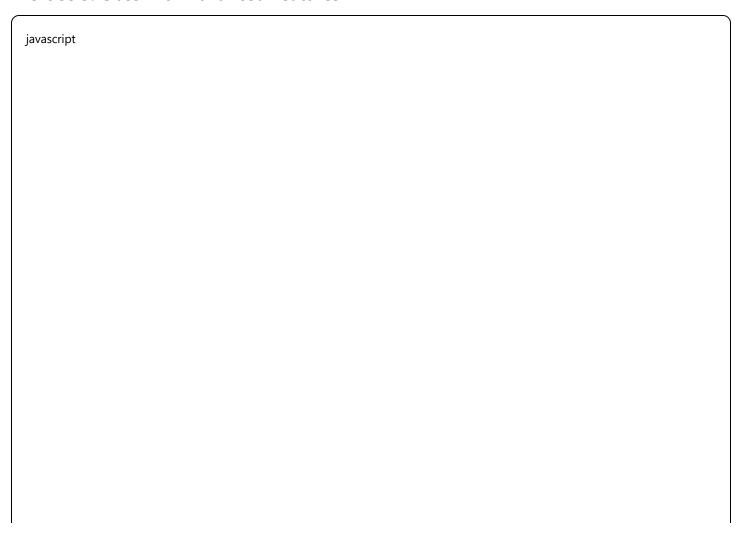
// Usage

const csRegistrations = registrationGenerator('CS');

console.log(csRegistrations.next().value); // "2024-CS-001"

console.log(csRegistrations.next().value); // "2024-CS-002"
```

### **Exercise 3: Class with Advanced Features**



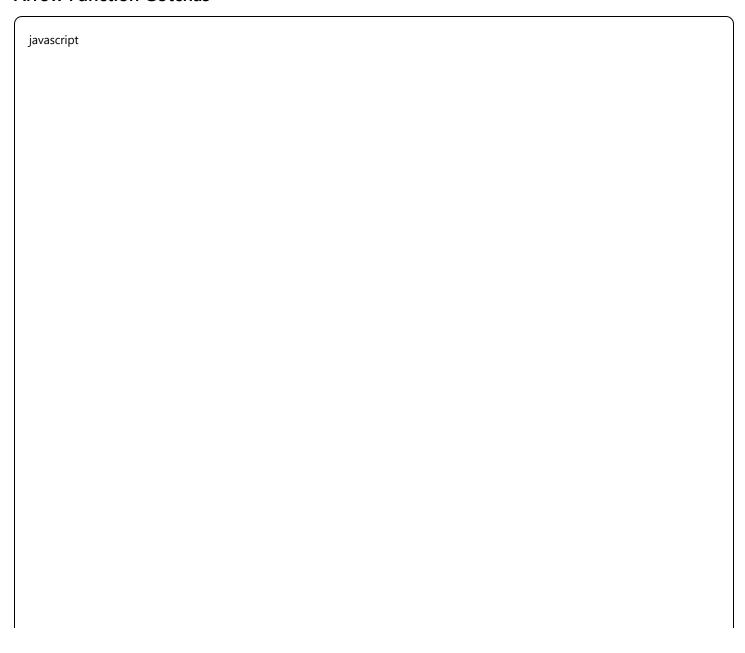
```
// Create a Student class with:
// - Private fields for sensitive data
// - Static methods for comparison
// - Getters/setters with validation
// - Symbol-keyed methods
const PRIVATE_DATA = Symbol('privateData');
class Student {
  // Private fields (if supported)
  #ssn:
  #grades = [];
  constructor(name, email, ssn) {
     this.name = name:
     this.email = email;
     this.#ssn = ssn:
     // Alternative private storage using Symbol
     this[PRIVATE_DATA] = {
       enrollmentDate: new Date(),
       advisorNotes: []
    };
  // Getter with computation
  get gpa() {
     if (this.#grades.length ===0) return 0;
     const sum = this.#grades.reduce((total, grade) => total + grade, 0);
     return (sum / this.#grades.length).toFixed(2);
  // Setter with validation
  set gpa(value) {
     throw new Error('GPA is computed automatically');
  addGrade(grade) {
     if (grade < 0 || grade > 4.0) {
       throw new Error('Grade must be between 0 and 4.0');
     this.#grades.push(grade);
```

```
// Static method for comparison
static compareByGPA(student1, student2) {
    return student2.gpa - student1.gpa;
}

// Symbol-keyed method (somewhat private)
[Symbol.for('getPrivateData')]() {
    return this[PRIVATE_DATA];
}
```

# **Common Pitfalls and Best Practices**

### **Arrow Function Gotchas**



```
// X Wrong: Arrow function for object methods
const obj = {
  name: 'Alice',
  greet: () => {
     console.log(this.name); // undefined - no 'this' binding
};
// Correct: Use regular function for methods
const obj = {
  name: 'Alice',
  greet() {
     console.log(this.name); // 'Alice'
};
// X Wrong: Arrow function as constructor
const Student = (name) => {
  this.name = name; // Error: Arrow functions can't be constructors
};
// 
// Correct: Use class or function declaration
class Student {
  constructor(name) {
     this.name = name;
```

# **Destructuring Pitfalls**

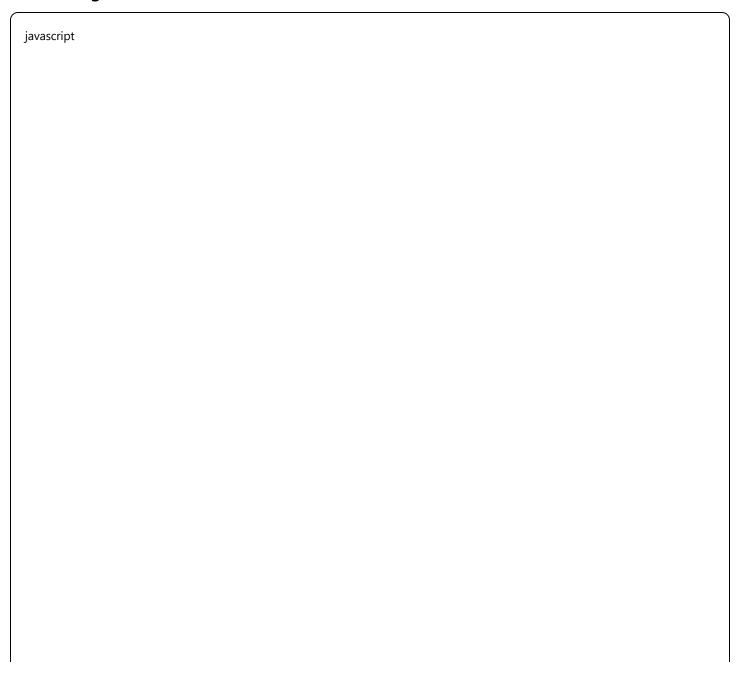
```
// X Wrong: Destructuring undefined
const { name } = undefined; // TypeError

// Correct: Provide defaults
const { name } = student || {};
const { name = 'Unknown' } = student || {};

// X Wrong: Confusing nested destructuring
const { student: { courses: { 0: firstCourse } } } = data; // Hard to read

// Correct: Step by step or use optional chaining
const firstCourse = data?.student?.courses?.[0];
```

# **Class Design Best Practices**



```
// Good: Use private fields when available
class BankAccount {
  \#balance = 0;
  deposit(amount) {
     if (amount > 0) {
        this.#balance += amount;
  get balance() {
     return this.#balance;
// Good: Use static methods for utility functions
class MathUtils {
  static factorial(n) {
     return n \le 1 ? 1 : n * this.factorial(n - 1);
  static isPrime(n) {
     if (n < 2) return false;
     for (let i = 2; i \le Math.sqrt(n); i++) {
        if (n \% i === 0) return false;
     return true;
```

# **Assignment: University Management System Enhancement**

Objective: Enhance the existing University Management System using modern JavaScript features.

### Requirements:

- 1. Convert all function declarations to ES6 classes.
- 2. Implement destructuring for all data operations
- 3. Use template literals for all string generation
- 4. Add generators for data pagination

- 5. Implement proxy for data validation
- 6. Use symbols for private methods
- 7. Add async generators for batch operations

#### **Deliverables:**

- Refactored codebase using modern JavaScript
- Documentation explaining each modern feature used
- Unit tests demonstrating functionality
- Performance comparison with old implementation

#### **Grading Criteria:**

- Correct implementation of ES6+ features (40%)
- Code organization and readability (30%)
- Testing and documentation (20%)
- Creative use of advanced features (10%)

### **Assessment Questions**

# **Knowledge Check (Multiple Choice)**

- 1. Which of the following creates a block-scoped variable?
  - a) (var x = 5)
  - b) (let x = 5;
  - c)  $\left(\text{const } x = 5;\right)$
  - d) Both b and c
- 2. What does the following code output?

```
javascript

const arr = [1, 2, 3];
const [a, ...rest] = arr;
console.log(rest.length);
```

- a) 1
- b) 2

- c) 3
- d) Error

### **Practical Application**

Write a function that:

- Uses destructuring for parameters
- Returns an object with computed properties
- Implements default parameters
- Uses template literals for output

### **Code Review**

Identify and fix the issues in this code:

```
javascript

const Student = (name, courses) => {
    this.name = name;
    this.courses = courses;

    this.addCourse = (course) => {
        this.courses.push(course);
    };

    this.getCourseList = () => {
        return 'Student ' + this.name + ' is enrolled in: ' + this.courses.join(', ');
    };
};
```

### **Additional Resources**

# **Recommended Reading**

- MDN Web Docs: ES6 Features
- "Understanding ECMAScript 6" by Nicholas Zakas
- "Exploring ES6" by Dr. Axel Rauschmayer

#### **Online Tools**

• Babel REPL for ES6+ experimentation

- ESLint for code quality
- Prettier for code formatting

### **Browser Support**

- Check caniuse.com for feature compatibility
- Use Babel for transpilation when needed
- Consider polyfills for older environments

#### **PowerPoint Slide Outlines**

### Slide Set 1: Introduction to Modern JavaScript (10 slides)

#### Slide 1: Module Overview

- Title: "Modern JavaScript Features (ES6+)"
- Learning objectives bullet points
- Timeline: 2 weeks, 8 sessions

#### Slide 2: JavaScript Evolution Timeline

- ES5 (2009) → ES6/ES2015 (2015) → ES2016+ (Annual releases)
- Key milestone features introduced
- Browser adoption timeline

#### Slide 3: Why Modern JavaScript Matters

- Industry adoption statistics
- Developer productivity improvements
- Code readability and maintainability
- Performance considerations

### Slide 4: Block Scope Revolution

- Side-by-side comparison: var vs let/const
- Hoisting behavior differences
- Temporal Dead Zone explanation
- Common use cases

#### Slide 5: Arrow Functions Deep Dive

- Syntax variations with examples
- 'this' binding comparison table
- When to use vs avoid arrow functions
- Performance implications

#### Slide 6: Template Literals Power

- String interpolation examples
- Multi-line string benefits
- Tagged templates introduction
- Real-world HTML generation

#### Slide 7: Destructuring Magic

- Array destructuring patterns
- Object destructuring patterns
- Function parameter destructuring
- Practical use cases

#### Slide 8: Spread/Rest Operators

- Visual representation of spreading
- Common patterns and use cases
- Performance considerations
- Best practices

#### Slide 9: Enhanced Objects and Classes

- Object literal enhancements
- Class syntax benefits
- Comparison with prototype pattern
- Inheritance made simple

#### Slide 10: Module 7.1 Recap

- Key concepts summary
- Common patterns to remember

Next session preview

### Slide Set 2: Advanced Concepts (12 slides)

#### Slide 11: Prototypal Inheritance Revisited

- Prototype chain visualization
- Object.create() vs constructor functions
- Modern class syntax as syntactic sugar
- When to use each approach

#### Slide 12: Symbols - Unique Identifiers

- Symbol creation and uniqueness
- Use cases for property keys
- Well-known symbols overview
- Privacy implementation patterns

#### Slide 13: Iterators and Iteration Protocol

- Iterator interface explanation
- Making objects iterable
- Built-in iterables examples
- Custom iterator implementation

#### Slide 14: Generator Functions

- Generator syntax and behavior
- yield keyword functionality
- Practical use cases
- Memory efficiency benefits

### Slide 15: Async Generators

- Combining async/await with generators
- Streaming data processing
- Error handling patterns
- Performance considerations

### Slide 16: Proxy and Reflect APIs

- Proxy trap operations
- Meta-programming possibilities
- Validation and logging use cases
- Performance implications

### Slide 17: WeakMap and WeakSet

- Garbage collection benefits
- Privacy implementation
- Memory leak prevention
- When to choose weak collections

### Slide 18: Regular Expressions Enhanced

- ES2018+ RegExp features
- Named capture groups
- Lookbehind assertions
- Unicode property escapes

#### Slide 19: Advanced Patterns Combination

- Mixing modern features effectively
- Design patterns with new syntax
- Performance optimization techniques
- Code organization strategies

### Slide 20: Real-World Applications

- Framework patterns using modern JS
- Library design considerations
- API design improvements
- Developer experience enhancements

### Slide 21: Browser Support and Transpilation

- Current browser support matrix
- Babel configuration
- Polyfill strategies

• Progressive enhancement

## Slide 22: Module 7 Complete Summary

- All concepts overview
- Practical application roadmap
- Next steps for continued learning

# **Live Coding Demonstrations**

# **Demo 1: University Student Portal (30 minutes)**

Setup Phase (5 minutes)

```
javascript
// Starting with old-style code
function Student(name, email, courses) {
    this.name = name;
    this.courses = email;
    this.courses = courses || [];
}

Student.prototype.addCourse = function(course) {
    this.courses.push(course);
};

Student.prototype.getCourseList = function() {
    return "Student " + this.name + " is enrolled in: " + this.courses.join(", ");
};

var alice = new Student("Alice", "alice@university.edu", ["Math", "Physics"]);
alice.addCourse("Chemistry");
console.log(alice.getCourseList());
```

## **Transformation Phase (20 minutes)**

```
javascript
```

```
// Step 1: Convert to class syntax
class Student {
  constructor(name, email, courses = []) {
     this.name = name:
     this.email = email:
     this.courses = [...courses]; // Use spread for array copying
  }
  // Step 2: Use destructuring and modern syntax
  addCourse({ name, code, credits = 3 }) {
     this.courses.push({ name, code, credits });
     return this; // Method chaining
  // Step 3: Template literals and destructuring
  getCourseList() {
     const { name, courses } = this;
     return `Student ${name} is enrolled in: ${courses.map(c => c.name).join(', ')}';
  // Step 4: Getters and computed properties
  get courseCount() {
     return this.courses.length;
  get totalCredits() {
     return this.courses.reduce((sum, { credits }) => sum + credits, 0);
  // Step 5: Static methods
  static compareByCredits(student1, student2) {
     return student2.totalCredits - student1.totalCredits;
// Step 6: Enhanced object creation
const alice = new Student("Alice", "alice@university.edu");
alice
  .addCourse({ name: "Advanced Mathematics", code: "MATH301" })
  .addCourse({ name: "Quantum Physics", code: "PHYS401", credits: 4 });
```

```
console.log(alice.getCourseList());
console.log(`Total credits: ${alice.totalCredits}`);
```

### **Advanced Features Phase (5 minutes)**

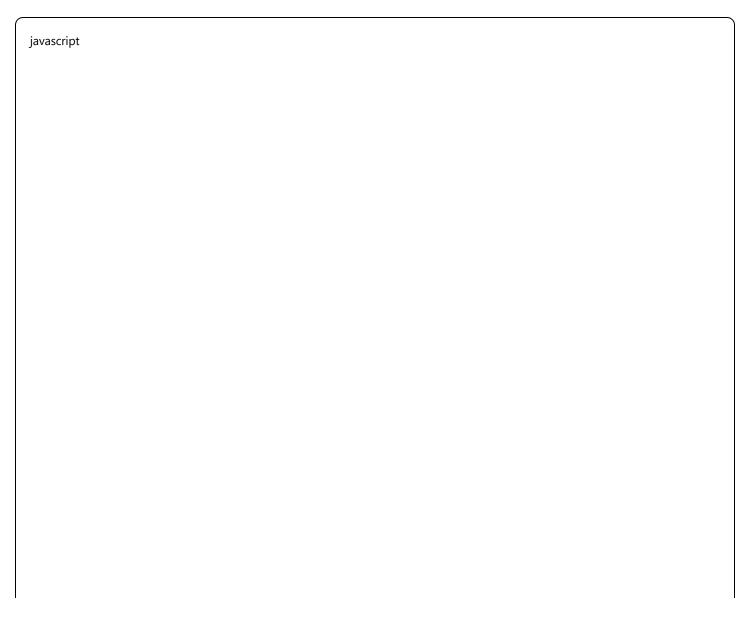
```
javascript
// Private data with WeakMap
const privateData = new WeakMap();
class SecureStudent extends Student {
  constructor(name, email, ssn) {
     super(name, email);
     privateData.set(this, { ssn, advisorNotes: [] });
  addAdvisorNote(note) {
     privateData.get(this).advisorNotes.push({
       note,
       timestamp: new Date().tolSOString()
    });
  getPrivateInfo(authorized = false) {
     if (!authorized) {
       throw new Error('Unauthorized access to private information');
     return privateData.get(this);
```

# Demo 2: Data Processing Pipeline (25 minutes)

**Traditional Approach (5 minutes)** 

javascript			

# Modern Approach (15 minutes)



```
// Modern way - using multiple ES6+ features
class StudentDataProcessor {
  constructor() {
     this.processors = new Map();
     this.setupProcessors();
  setupProcessors() {
     // Using arrow functions and destructuring
     this.processors.set('honor_students',
       ({ gpa, major }) => gpa >= 3.5 && major === 'Computer Science'
     );
     this.processors.set('standing_calculator',
       ({ gpa }) => gpa >= 3.8 ? 'Dean\'s List' : 'Honor Roll'
     );
  // Generator for memory-efficient processing
  *processStudents(students) {
     const honorFilter = this.processors.get('honor_students');
     const standingCalc = this.processors.get('standing_calculator');
     for (const student of students) {
       if (honorFilter(student)) {
          yield {
             name: student.name,
             gpa: student.gpa,
             standing: standingCalc(student),
             processedAt: new Date().tolSOString()
          };
  // Async generator for API data
  async *processFromAPI(apiEndpoint) {
       const response = await fetch(apiEndpoint);
        const students = await response.json();
       for (const student of this.processStudents(students)) {
          yield student;
```

```
} catch (error) {
       yield { error: `Failed to process from ${apiEndpoint}: ${error.message}` };
  // Template literals for reporting
  generateReport(processedStudents) {
     const studentsList = processedStudents
       .map(({ name, gpa, standing }) =>
          `• ${name} (GPA: ${gpa}) - ${standing}`
       .join('\n');
     return `
Honor Students Report
Generated: ${new Date().toLocaleDateString()}
Total Students: ${processedStudents.length}
${studentsList}
     `.trim();
// Usage demonstration
const processor = new StudentDataProcessor();
const students = [
  { name: 'Alice', gpa: 3.9, major: 'Computer Science' },
  { name: 'Bob', gpa: 3.6, major: 'Computer Science' },
  { name: 'Charlie', gpa: 3.4, major: 'Mathematics' }
];
const honorStudents = [...processor.processStudents(students)];
console.log(processor.generateReport(honorStudents));
```

## **Advanced Patterns (5 minutes)**

javascript

```
// Proxy for validation and logging
const createValidatedProcessor = (processor) => {
    return new Proxy(processor, {
        get(target, prop) {
            console.log(`Accessing method: ${prop}`);
            const value = target[prop];

        if (typeof value === 'function') {
            return function(...args) {
                console.log(`Calling ${prop} with:`, args);
                return value.apply(target, args);
            }
        }
        return value;
    }
    ));
    const validatedProcessor = createValidatedProcessor(new StudentDataProcessor());
}
```

# **Interactive Workshop Activities**

# **Activity 1: Syntax Transformation Challenge (20 minutes)**

**Instructions:** Convert the following ES5 code to modern JavaScript, using as many ES6+ features as appropriate.



```
// Original ES5 code
function CourseManager() {
  this.courses = [];
  this.students = []:
  this.enrollments = {};
CourseManager.prototype.addCourse = function(name, code, instructor, capacity) {
  var course = {
     name: name,
     code: code,
     instructor: instructor,
     capacity: capacity | 30,
     enrolled: []
  };
  this.courses.push(course);
  return course;
};
CourseManager.prototype.enrollStudent = function(studentId, courseCode) {
  var course = null;
  for (var i = 0; i < this.courses.length; i++) {</pre>
     if (this.courses[i].code === courseCode) {
       course = this.courses[i];
       break:
  if (!course) {
     throw new Error('Course not found: ' + courseCode);
  if (course.enrolled.length >= course.capacity) {
     throw new Error('Course is full');
  course.enrolled.push(studentId);
  if (!this.enrollments[studentId]) {
     this.enrollments[studentId] = [];
  this.enrollments[studentId].push(courseCode);
```

```
return {
    success: true,
    message: 'Student ' + studentId + ' enrolled in ' + course.name
    };
};
```

# **Expected Modern Solution:**

javascript	

```
class CourseManager {
  constructor() {
    this.courses = new Map();
    this.students = new Set():
    this.enrollments = new Map();
  }
  addCourse({ name, code, instructor, capacity = 30 }) {
    if (this.courses.has(code)) {
       throw new Error(`Course ${code} already exists`);
    const course = {
       name,
       code,
       instructor,
       capacity,
       enrolled: new Set()
    };
    this.courses.set(code, course);
    return course;
  enrollStudent(studentId, courseCode) {
    const course = this.courses.get(courseCode);
    if (!course) {
       throw new Error(`Course not found: ${courseCode}`);
    if (course.enrolled.size >= course.capacity) {
       throw new Error('Course is full');
    course.enrolled.add(studentId);
    if (!this.enrollments.has(studentId)) {
       this.enrollments.set(studentId, new Set());
    this.enrollments.get(studentId).add(courseCode);
    return {
```

```
success: true,
    message: `Student ${studentId} enrolled in ${course.name}`
};
}

// Bonus: Generator for enrolled students
*getEnrolledStudents(courseCode) {
    const course = this.courses.get(courseCode);
    if (course) {
        yield* course.enrolled;
    }
}
```

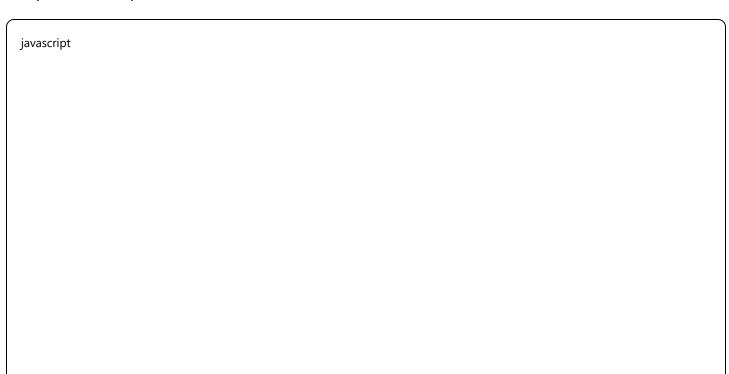
# **Activity 2: Design Pattern Implementation (30 minutes)**

Challenge: Implement a modern JavaScript Observer pattern using ES6+ features.

## Requirements:

- Use classes and private fields/methods
- Implement with Symbol keys for privacy
- Use generators where appropriate
- Include error handling with modern syntax

## **Template to Complete:**



```
const OBSERVERS = Symbol('observers');
const NOTIFY = Symbol('notify');
class Observable {
  constructor() {
    // TODO: Initialize private data
  subscribe(observer) {
    // TODO: Add observer with validation
  unsubscribe(observer) {
    // TODO: Remove observer
  [NOTIFY](data) {
    // TODO: Notify all observers with error handling
  // TODO: Add generator method for iterating observers
class CourseObservable extends Observable {
  constructor(courseName) {
    // TODO: Initialize course-specific data
  enrollStudent(student) {
    // TODO: Enroll student and notify observers
  dropStudent(student) {
    // TODO: Drop student and notify observers
```

# **Assessment Rubrics**

**Practical Assignment Rubric (100 points total)** 

ES6+ Feature Implementation (40 points)

- Excellent (36-40): Uses all major ES6+ features appropriately and effectively
- Good (32-35): Uses most ES6+ features correctly with minor issues
- Satisfactory (28-31): Uses some ES6+ features but misses opportunities
- Needs Improvement (0-27): Limited or incorrect use of modern features

### Code Quality and Organization (30 points)

- Excellent (27-30): Well-structured, readable, follows best practices
- Good (24-26): Generally well-organized with good practices
- Satisfactory (21-23): Adequate organization, some improvement needed
- Needs Improvement (0-20): Poor organization, hard to follow

### **Testing and Documentation (20 points)**

- Excellent (18-20): Comprehensive tests and clear documentation
- Good (16-17): Good test coverage and documentation
- Satisfactory (14-15): Basic tests and documentation
- Needs Improvement (0-13): Insufficient testing or documentation

### **Creative Application (10 points)**

- Excellent (9-10): Creative use of advanced features, innovative solutions
- Good (8): Some creative applications
- Satisfactory (7): Standard implementation
- Needs Improvement (0-6): Basic or minimal effort

#### Code Review Checklist

### **Modern Syntax Usage:**

Uses const and let instead of var				
☐ Implements arrow functions appropriately				
$lue{}$ Uses template literals for string interpolation				
☐ Applies destructuring for cleaner code				
<ul> <li>Uses spread/rest operators effectively</li> </ul>				

#### **Advanced Features:**

- Implements classes with proper structure
- Uses generators where beneficial

<ul><li>□ Applies symbols for privacy</li><li>□ Implements proper error handling</li><li>□ Uses modern iteration patterns</li></ul>
Best Practices:
<ul> <li>□ Follows naming conventions</li> <li>□ Includes proper documentation</li> <li>□ Handles edge cases</li> <li>□ Implements efficient algorithms</li> <li>□ Uses appropriate data structures</li> </ul>

# **Troubleshooting Common Issues**

## **Issue 1: Arrow Function Context Problems**

```
javascript
// Problem
const button = document.getElementById('myButton');
const handler = {
  count: 0,
  onClick: () => {
     this.count++; // `this` is not the handler object!
     console.log(this.count);
};
// Solution
const handler = {
  count: 0,
  onClick() {
     this.count++;
     console.log(this.count);
};
```

# **Issue 2: Destructuring with Dynamic Properties**

javascript			

```
// Problem
const propName = 'studentName';
const { propName } = student; // This doesn't work as expected

// Solution
const { [propName]: value } = student; // Computed property names
```

### Issue 3: Generator and Iterator Confusion

```
javascript
// Problem - treating generator like array
function* numbers() {
    yield 1;
    yield 2;
    yield 3;
}

const nums = numbers();
console.log(nums.length); // undefined
console.log(nums[0]); // undefined

// Solution - proper generator usage
const nums = numbers();
for (const num of nums) {
    console.log(num); // 1, 2, 3
}

// Or convert to array when needed
const numArray = [...numbers()];
```

### **Extension Activities for Advanced Students**

# **Challenge 1: Meta-Programming with Proxies**

Create a dynamic API client that:

- Automatically generates methods based on property access
- Logs all API calls
- Implements retry logic
- Caches responses

### **Challenge 2: Custom Iterator Implementation**

Build a tree traversal system using:

- Custom iterators for different traversal orders
- Generators for memory efficiency
- Symbol.iterator for for...of compatibility

## **Challenge 3: Advanced Class Patterns**

Implement a mixin system using:

- Class expressions
- Dynamic inheritance
- Symbol-based private methods
- Decorator pattern simulation

# **Summary**

Module 7 introduces students to the modern JavaScript ecosystem with ES6+ features that have become standard in contemporary development. Students learn to write more concise, readable, and powerful code while understanding the underlying concepts that make these features work.

Key takeaways:

- Modern syntax improves code readability and maintainability
- Advanced features like generators and proxies enable sophisticated programming patterns
- Understanding prototypal inheritance is crucial for mastering JavaScript
- Best practices help avoid common pitfalls with new syntax

The module prepares students for real-world JavaScript development and sets the foundation for advanced topics in subsequent modules.

**Next Module Preview:** Module 8 will focus on Error Handling and Debugging, building upon the modern JavaScript foundation to create robust, maintainable applications.