Module 4: Objects and Arrays

Weeks 6-7

Learning Objectives

By the end of this module, you will:

- Master array manipulation and iteration methods
- Create and work with complex objects
- Understand object-oriented programming concepts in JavaScript
- Use ES6 classes for structured programming
- Process JSON data effectively
- Implement advanced data structures

Arrays: Beyond the Basics

Array Creation Methods:

```javascript
// Literal notation (most common)
const students = ['Alice', 'Bob', 'Charlie'];

// Array constructor
const scores = new Array(85, 92, 78, 96);
const emptyArray = new Array(5); // Creates array with 5 empty slots

// Array.from() - convert iterable to array
const charArray = Array.from('Hello'); // ['H', 'e', 'l', 'l', 'o']
const numberArray = Array.from({length: 5}, (_, i) => i + 1); // [1, 2, 3, 4, 5]

// Array.of() - create array from arguments
const mixedArray = Array.of(1, 'hello', true, null); // [1, 'hello', true, null]
```

Essential Array Methods

Adding and Removing Elements:

```javascript
const courses = ['JavaScript', 'Python'];

// Add to end
courses.push('Java', 'C++');
console.log(courses); // ['JavaScript', 'Python', 'Java', 'C++']

// Add to beginning
courses.unshift('HTML', 'CSS');
console.log(courses); // ['HTML', 'CSS', 'JavaScript', 'Python', 'Java', 'C++']

// Remove from end
const lastCourse = courses.pop();
console.log(lastCourse); // 'C++'

// Remove from beginning
const firstCourse = courses.shift();
console.log(firstCourse); // 'HTML'

// Remove/add at specific position
const removed = courses.splice(1, 2, 'React', 'Node.js');
console.log(removed); // ['CSS', 'JavaScript'] (removed elements)
console.log(courses); // ['HTML', 'React', 'Node.js', 'Python', 'Java']
```

Array Slicing and Copying:

```javascript
const originalArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// slice() - extract a portion (doesn't modify original)
const portion = originalArray.slice(2, 6); // [3, 4, 5, 6]
const lastThree = originalArray.slice(-3); // [8, 9, 10]
const copy = originalArray.slice(); // Full copy

// Spreading for copying/concatenating
const copy2 = [...originalArray];
const combined = [...originalArray, 11, 12, 13];
```

Advanced Array Iteration

forEach - Execute function for each element:

javascript

```javascript
const students = [
  { name: 'Alice', gpa: 3.8 },
  { name: 'Bob', gpa: 3.2 },
  { name: 'Charlie', gpa: 3.9 }
];

// Basic forEach
students.forEach(student => {
  console.log(`${student.name}: GPA ${student.gpa}`);
});

// forEach with index and array parameters
students.forEach((student, index, array) => {
  console.log(`#${index + 1}: ${student.name} (${array.length} total students)`);
});
```

map - Transform each element:

javascript

```javascript
// Transform student objects to summary strings
const studentSummaries = students.map(student =>
  `${student.name} (${student.gpa >= 3.5 ? 'Excellent' : 'Good'})`
);

// Extract specific properties
const names = students.map(student => student.name);
const gpas = students.map(student => student.gpa);

// Complex transformations
const studentReports = students.map((student, index) => ({
  id: index + 1,
  name: student.name,
  gpaStatus: student.gpa >= 3.5 ? 'Dean\'s List' : 'Standard',
  needsImprovement: student.gpa < 3.0
}));
```

filter - Select elements based on criteria:

javascript

```
// High-achieving students
const topStudents = students.filter(student => student.gpa >= 3.5);

// Students needing academic support
const needsSupport = students.filter(student => student.gpa < 3.0);

// Complex filtering
const eligibleForHonors = students.filter(student => {
  return student.gpa >= 3.7 && student.courses && student.courses.length >= 4;
});

// Chaining filter with map
const topStudentNames = students
  .filter(student => student.gpa >= 3.5)
  .map(student => student.name);
```

---

reduce - The Most Powerful Array Method

Basic Reduction:

javascript

```
const scores = [85, 92, 78, 96, 89];

// Sum all scores
const totalScore = scores.reduce((sum, score) => sum + score, 0);

// Find maximum
const maxScore = scores.reduce((max, score) => Math.max(max, score), -Infinity);

// Calculate average
const average = scores.reduce((sum, score) => sum + score, 0) / scores.length;
```

Complex Reductions:

javascript

```javascript
const students = [
  { name: 'Alice', major: 'CS', gpa: 3.8, credits: 120 },
  { name: 'Bob', major: 'Math', gpa: 3.2, credits: 90 },
  { name: 'Charlie', major: 'CS', gpa: 3.9, credits: 110 },
  { name: 'Diana', major: 'Physics', gpa: 3.6, credits: 95 }
];

// Group students by major
const studentsByMajor = students.reduce((groups, student) => {
  const major = student.major;
  if (!groups[major]) {
    groups[major] = [];
  }
  groups[major].push(student);
  return groups;
}, {});

// Calculate statistics
const stats = students.reduce((acc, student) => {
  acc.totalGPA += student.gpa;
  acc.totalCredits += student.credits;
  acc.count += 1;

  if (student.gpa > acc.highestGPA) {
    acc.highestGPA = student.gpa;
    acc.topStudent = student.name;
  }

  return acc;
}, {
  totalGPA: 0,
  totalCredits: 0,
  count: 0,
  highestGPA: 0,
  topStudent: null
});

stats.averageGPA = stats.totalGPA / stats.count;
stats.averageCredits = stats.totalCredits / stats.count;
```

Array Search Methods

find and findIndex:

javascript

```
const students = [
  { id: 1, name: 'Alice', email: 'alice@university.edu' },
  { id: 2, name: 'Bob', email: 'bob@university.edu' },
  { id: 3, name: 'Charlie', email: 'charlie@university.edu' }
];

// Find specific student
const alice = students.find(student => student.name === 'Alice');
const studentById = students.find(student => student.id === 2);

// Find index
const bobIndex = students.findIndex(student => student.name === 'Bob');

// Check if element exists
const hasAlice = students.some(student => student.name === 'Alice');
const allHaveEmails = students.every(student => student.email.includes('@'));

// includes() for primitive values
const courseList = ['JavaScript', 'Python', 'Java', 'C++'];
const hasJavaScript = courseList.includes('JavaScript'); // true
const hasRuby = courseList.includes('Ruby'); // false
```

Array Sorting

Basic Sorting:

javascript

```
const names = ['Charlie', 'Alice', 'Bob', 'Diana'];
const sortedNames = [...names].sort(); // ['Alice', 'Bob', 'Charlie', 'Diana']

const numbers = [10, 5, 40, 25, 1000, 1];
const sortedNumbers = [...numbers].sort((a, b) => a - b); // [1, 5, 10, 25, 40, 1000]
```

Advanced Sorting:

```javascript
const students = [
    { name: 'Alice', gpa: 3.8, year: 3 },
    { name: 'Bob', gpa: 3.2, year: 2 },
    { name: 'Charlie', gpa: 3.9, year: 4 },
    { name: 'Diana', gpa: 3.6, year: 2 }
];

// Sort by GPA (descending)
const byGPA = [...students].sort((a, b) => b.gpa - a.gpa);

// Sort by multiple criteria
const byYearThenGPA = [...students].sort((a, b) => {
    // First by year
    if (a.year !== b.year) {
        return b.year - a.year; // Descending year
    }
    // Then by GPA
    return b.gpa - a.gpa; // Descending GPA
});

// Custom sort function
function createSorter(sortKey, ascending = true) {
    return (a, b) => {
        const aValue = a[sortKey];
        const bValue = b[sortKey];

        if (typeof aValue === 'string') {
            return ascending
                ? aValue.localeCompare(bValue)
                : bValue.localeCompare(aValue);
        }

        return ascending
            ? aValue - bValue
            : bValue - aValue;
    };
}

const sortedByName = [...students].sort(createSorter('name', true));
```

Objects: Advanced Concepts

Object Creation Patterns:

javascript

```javascript
// Object literal
const student1 = {
  name: 'Alice Johnson',
  age: 20,
  major: 'Computer Science',
  gpa: 3.8
};

// Object constructor
const student2 = new Object();
student2.name = 'Bob Smith';
student2.age = 19;
student2.major = 'Mathematics';

// Object.create()
const studentTemplate = {
  getFullInfo() {
    return `${this.name}, ${this.age} years old, majoring in ${this.major}`;
  }
};

const student3 = Object.create(studentTemplate);
student3.name = 'Charlie Brown';
student3.age = 21;
student3.major = 'Physics';
```

Object Property Access:

javascript

```
const student = {
  'first-name': 'Alice',
  'last-name': 'Johnson',
  age: 20,
  courses: ['JavaScript', 'Data Structures']
};

// Dot notation (preferred when possible)
console.log(student.age);
console.log(student.courses);

// Bracket notation (required for special characters or dynamic access)
console.log(student['first-name']);
console.log(student['last-name']);

// Dynamic property access
const propertyName = 'age';
console.log(student[propertyName]);

// Computed property names (ES6)
const dynamicKey = 'major';
const student2 = {
  name: 'Bob',
  [dynamicKey]: 'Mathematics',
  [`${dynamicKey}Code`]: 'MATH'
};
```

Object Methods and this

Method Definition:

javascript

```
const calculator = {
    result: 0,

    // Traditional method
    add: function(value) {
        this.result += value;
        return this; // Enable chaining
    },

    // ES6 shorthand method
    subtract(value) {
        this.result -= value;
        return this;
    },

    // Arrow functions (be careful with 'this')
    multiply: (value) => {
        // 'this' doesn't refer to calculator object!
        // this.result *= value; // This won't work as expected
    },

    reset() {
        this.result = 0;
        return this;
    },

    getValue() {
        return this.result;
    }
};

// Method chaining
const result = calculator
    .add(10)
    .subtract(3)
    .add(5)
    .getValue(); // 12
```

Understanding 'this' Context:

javascript

```
const student = {
  name: 'Alice',
  courses: ['JavaScript', 'Python'],

  addCourse(courseName) {
    this.courses.push(courseName);
  },

  getCourseCount() {
    return this.courses.length;
  },

  printCourses() {
    this.courses.forEach(function(course, index) {
      // 'this' is undefined here in strict mode
      console.log(`${index + 1}: ${course}`);
    });
  },

  printCoursesCorrect() {
    // Solution 1: Arrow function preserves 'this'
    this.courses.forEach((course, index) => {
      console.log(`${this.name}'s course ${index + 1}: ${course}`);
    });
  },

  printCoursesAlternative() {
    // Solution 2: Store reference to 'this'
    const self = this;
    this.courses.forEach(function(course, index) {
      console.log(`${self.name}'s course ${index + 1}: ${course}`);
    });
  }
};
```

Object Destructuring

Basic Destructuring:

```
javascript
```

```javascript
const student = {
  name: 'Alice Johnson',
  age: 20,
  major: 'Computer Science',
  gpa: 3.8,
  address: {
    street: '123 Main St',
    city: 'University City',
    state: 'CA'
  }
};

// Extract properties
const { name, age, major } = student;

// Rename variables
const { name: studentName, gpa: gradePoint } = student;

// Default values
const { minor = 'Undeclared', year = 1 } = student;

// Nested destructuring
const { address: { city, state } } = student;

// Function parameters destructuring
function printStudentInfo({ name, major, gpa }) {
  console.log(`${name} is majoring in ${major} with a ${gpa} GPA`);
}

printStudentInfo(student);
```

Advanced Destructuring:

javascript

```javascript
// Rest properties
const { name, ...otherInfo } = student;

// Array destructuring
const courses = ['JavaScript', 'Python', 'Java', 'C++'];
const [first, second, ...remaining] = courses;

// Destructuring in loops
const students = [
  { name: 'Alice', gpa: 3.8 },
  { name: 'Bob', gpa: 3.2 },
  { name: 'Charlie', gpa: 3.9 }
];

students.forEach(({ name, gpa }) => {
  console.log(`${name}: ${gpa}`);
});

// Swapping variables
let a = 1, b = 2;
[a, b] = [b, a]; // a = 2, b = 1
```

ES6 Classes

Basic Class Definition:

javascript

```
class Student {
    constructor(name, major, gpa = 0.0) {
        this.name = name;
        this.major = major;
        this.gpa = gpa;
        this.courses = [];
        this.id = Student.generateId();
    }

    // Instance method
    addCourse(courseName, grade) {
        this.courses.push({ name: courseName, grade: grade });
        this.updateGPA();
    }

    // Private method (using convention)
    updateGPA() {
        if (this.courses.length === 0) return;

        const totalPoints = this.courses.reduce((sum, course) => sum + course.grade, 0);
        this.gpa = totalPoints / this.courses.length;
    }

    // Getter
    get courseCount() {
        return this.courses.length;
    }

    // Setter
    set studentName(newName) {
        if (typeof newName === 'string' && newName.length > 0) {
            this.name = newName;
        }
    }

    // Static method
    static generateId() {
        return Math.floor(Math.random() * 10000);
    }

    // Static property
    static maxGPA = 4.0;
    // Instance method
    getStatus() {
        if (this.gpa >= 3.5) return 'Excellent';
        if (this.gpa >= 3.0) return 'Good';
        if (this.gpa >= 2.0) return 'Satisfactory';
        return 'Needs Improvement';
    }

    // Method returning formatted string
    toString() {
        return `${this.name} (${this.major}) - GPA: ${this.gpa.toFixed(2)}`;
    }
}

// Usage
const alice = new Student('Alice Johnson', 'Computer Science');
alice.addCourse('JavaScript Fundamentals', 3.8);
alice.addCourse('Data Structures', 4.0);

console.log(alice.toString());
console.log(`Course count: ${alice.courseCount}`);
console.log(`Status: ${alice.getStatus()}`);
```

Class Inheritance

Extending Classes:

javascript

```
// Base class
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    introduce() {
        return `Hi, I'm ${this.name} and I'm ${this.age} years old.`;
    }

    getCategory() {
        return 'Person';
    }
}

// Extended class
class Student extends Person {
    constructor(name, age, major, studentId) {
        super(name, age); // Call parent constructor
        this.major = major;
        this.studentId = studentId;
        this.courses = [];
        this.gpa = 0.0;
    }

    // Override parent method
    introduce() {
        return super.introduce() + ` I'm studying ${this.major}.`;
    }

    // Override parent method
    getCategory() {
        return 'Student';
    }

    // New methods specific to Student
    enroll(courseName) {
        this.courses.push(courseName);
        return `Enrolled in ${courseName}`;
    }

    calculateGPA(grades) {
```

```
    if (grades.length === 0) return 0;
    const sum = grades.reduce((total, grade) => total + grade, 0);
    this.gpa = sum / grades.length;
    return this.gpa;
  }
}

// Further extension
class GraduateStudent extends Student {
  constructor(name, age, major, studentId, advisor) {
    super(name, age, major, studentId);
    this.advisor = advisor;
    this.researchArea = null;
    this.thesisTitle = null;
  }

  introduce() {
    return super.introduce() + ` I'm working with Professor ${this.advisor}.`;
  }

  setResearch(area, thesisTitle) {
    this.researchArea = area;
    this.thesisTitle = thesisTitle;
  }

  getResearchInfo() {
    return {
      area: this.researchArea,
      thesis: this.thesisTitle,
      advisor: this.advisor
    };
  }
}

// Usage
const alice = new Student('Alice Johnson', 20, 'Computer Science', 'S12345');
const bob = new GraduateStudent('Bob Smith', 25, 'AI Research', 'G67890', 'Dr. Williams');

console.log(alice.introduce());
console.log(bob.introduce());

bob.setResearch('Machine Learning', 'Neural Networks in Natural Language Processing');
console.log(bob.getResearchInfo());
```

Private Fields and Methods (Modern JavaScript)

Private Properties:

javascript

```
class BankAccount {
  // Private fields (prefix with #)
  #balance = 0;
  #accountNumber;
  #transactions = [];

  constructor(accountNumber, initialBalance = 0) {
    this.#accountNumber = accountNumber;
    this.#balance = initialBalance;
    this.#addTransaction('Initial deposit', initialBalance);
  }

  // Public methods
  deposit(amount) {
    if (amount <= 0) {
      throw new Error('Deposit amount must be positive');
    }

    this.#balance += amount;
    this.#addTransaction('Deposit', amount);
    return this.#balance;
  }

  withdraw(amount) {
    if (amount <= 0) {
      throw new Error('Withdrawal amount must be positive');
    }

    if (amount > this.#balance) {
      throw new Error('Insufficient funds');
    }

    this.#balance -= amount;
    this.#addTransaction('Withdrawal', -amount);
    return this.#balance;
  }

  // Getter for balance (read-only access)
  get balance() {
    return this.#balance;
  }

  get accountNumber() {
        return this.#accountNumber;
    }

    getTransactionHistory() {
        return [...this.#transactions]; // Return copy
    }

    // Private method
    #addTransaction(type, amount) {
        this.#transactions.push({
            type,
            amount,
            date: new Date(),
            balance: this.#balance
        });
    }
}

const account = new BankAccount('ACC123', 1000);
console.log(account.balance); // 1000
account.deposit(500);
console.log(account.balance); // 1500

// console.log(account.#balance); // SyntaxError: Private field '#balance' must be declared in an enclosing class
```

Working with JSON

JSON Basics:

javascript

```
// JavaScript object
const student = {
  name: 'Alice Johnson',
  age: 20,
  major: 'Computer Science',
  courses: ['JavaScript', 'Python', 'Data Structures'],
  isEnrolled: true,
  gpa: 3.8
};

// Convert to JSON string
const jsonString = JSON.stringify(student);
console.log(jsonString);
// {"name":"Alice Johnson","age":20,"major":"Computer Science","courses":["JavaScript","Python","Data Structures"],"isEnrol

// Parse JSON string back to object
const parsedStudent = JSON.parse(jsonString);
console.log(parsedStudent.name); // "Alice Johnson"
```

Advanced JSON Operations:

javascript

```javascript
// Custom JSON serialization
const student = {
  name: 'Alice Johnson',
  age: 20,
  password: 'secret123', // Sensitive data
  lastLogin: new Date(),
  calculateGPA: function() { return this.gpa; }, // Function (won't be serialized)
  gpa: 3.8
};

// Custom replacer function
const jsonString = JSON.stringify(student, (key, value) => {
  // Exclude sensitive data
  if (key === 'password') return undefined;

  // Handle dates
  if (value instanceof Date) return value.toISOString();

  // Exclude functions
  if (typeof value === 'function') return undefined;

  return value;
});

// Custom reviver function
const parsed = JSON.parse(jsonString, (key, value) => {
  // Convert ISO strings back to dates
  if (typeof value === 'string' && /^\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}.\d{3}Z$/.test(value)) {
    return new Date(value);
  }
  return value;
});

// Pretty printing JSON
const prettyJson = JSON.stringify(student, null, 2);
console.log(prettyJson);
```

Complex Data Structures

University Management System Example:

javascript

```
class University {
  constructor(name) {
    this.name = name;
    this.students = new Map();
    this.courses = new Map();
    this.instructors = new Map();
    this.enrollments = [];
  }

  addStudent(studentData) {
    const student = new Student(
      studentData.name,
      studentData.age,
      studentData.major,
      this.generateStudentId()
    );

    this.students.set(student.studentId, student);
    return student;
  }

  addCourse(courseData) {
    const course = {
      id: this.generateCourseId(),
      name: courseData.name,
      code: courseData.code,
      credits: courseData.credits,
      instructor: courseData.instructor,
      maxEnrollment: courseData.maxEnrollment || 30,
      enrolledStudents: []
    };

    this.courses.set(course.id, course);
    return course;
  }

  enrollStudent(studentId, courseId) {
    const student = this.students.get(studentId);
    const course = this.courses.get(courseId);

    if (!student) throw new Error('Student not found');
    if (!course) throw new Error('Course not found');
```

```javascript
    if (course.enrolledStudents.length >= course.maxEnrollment) {
      throw new Error('Course is full');
    }

    // Check if already enrolled
    if (course.enrolledStudents.includes(studentId)) {
      throw new Error('Student already enrolled in this course');
    }

    course.enrolledStudents.push(studentId);
    student.enroll(course.name);

    const enrollment = {
      id: this.generateEnrollmentId(),
      studentId,
      courseId,
      enrollmentDate: new Date(),
      grade: null
    };

    this.enrollments.push(enrollment);
    return enrollment;
  }

  getStudentsByMajor(major) {
    return Array.from(this.students.values())
      .filter(student => student.major === major);
  }

  getCourseEnrollmentStats() {
    return Array.from(this.courses.values()).map(course => ({
      courseName: course.name,
      enrolled: course.enrolledStudents.length,
      capacity: course.maxEnrollment,
      utilizationRate: (course.enrolledStudents.length / course.maxEnrollment * 100).toFixed(1) + '%'
    }));
  }

  getStudentTranscript(studentId) {
    const student = this.students.get(studentId);
    if (!student) throw new Error('Student not found');

    const studentEnrollments = this.enrollments.filter(e => e.studentId === studentId);
    const transcript = studentEnrollments.map(enrollment => {
            const course = this.courses.get(enrollment.courseId);
            return {
                courseName: course.name,
                courseCode: course.code,
                credits: course.credits,
                grade: enrollment.grade,
                enrollmentDate: enrollment.enrollmentDate
            };
        });

        return {
            student: {
                name: student.name,
                id: student.studentId,
                major: student.major
            },
            courses: transcript,
            totalCredits: transcript.reduce((sum, course) => sum + course.credits, 0),
            gpa: student.gpa
        };
    }

    generateStudentId() {
        return 'STU' + Math.floor(Math.random() * 100000).toString().padStart(5, '0');
    }

    generateCourseId() {
        return 'CRS' + Math.floor(Math.random() * 10000).toString().padStart(4, '0');
    }

    generateEnrollmentId() {
        return 'ENR' + Math.floor(Math.random() * 1000000).toString().padStart(6, '0');
    }
}

// Usage example
const techUniversity = new University('Tech University');

// Add students
const alice = techUniversity.addStudent({
    name: 'Alice Johnson',
    age: 20,
    major: 'Computer Science'
});
```

```
const bob = techUniversity.addStudent({
    name: 'Bob Smith',
    age: 19,
    major: 'Mathematics'
});

// Add courses
const jsCourse = techUniversity.addCourse({
    name: 'JavaScript Programming',
    code: 'CS101',
    credits: 3,
    instructor: 'Dr. Smith',
    maxEnrollment: 25
});

const mathCourse = techUniversity.addCourse({
    name: 'Calculus I',
    code: 'MATH101',
    credits: 4,
    instructor: 'Prof. Johnson'
});

// Enroll students
techUniversity.enrollStudent(alice.studentId, jsCourse.id);
techUniversity.enrollStudent(alice.studentId, mathCourse.id);
techUniversity.enrollStudent(bob.studentId, mathCourse.id);

// Get statistics
console.log('CS Students:', techUniversity.getStudentsByMajor('Computer Science').length);
console.log('Course Stats:', techUniversity.getCourseEnrollmentStats());
console.log('Alice Transcript:', techUniversity.getStudentTranscript(alice.studentId));
```

Performance Considerations

Array vs Object Performance:

javascript

```
// Array operations are generally faster for:
// - Sequential access
// - Iterating through all elements
// - Adding/removing from end

// Object operations are faster for:
// - Key-based lookups
// - Adding/removing arbitrary properties
// - Sparse data

// Example: Large dataset processing
function performanceComparison() {
    const size = 100000;

    // Array creation and access
    console.time('Array operations');
    const arr = [];
    for (let i = 0; i < size; i++) {
        arr.push(i);
    }

    // Sequential access
    let sum = 0;
    for (let i = 0; i < arr.length; i++) {
        sum += arr[i];
    }
    console.timeEnd('Array operations');

    // Object creation and access
    console.time('Object operations');
    const obj = {};
    for (let i = 0; i < size; i++) {
        obj[i] = i;
    }

    // Property access
    let objSum = 0;
    for (let key in obj) {
        objSum += obj[key];
    }
    console.timeEnd('Object operations');
}
```

```
performanceComparison();
```

Memory-Efficient Patterns:

```javascript
// Use Object.freeze() for immutable objects
const constants = Object.freeze({
  MAX_STUDENTS: 1000,
  MIN_GPA: 0.0,
  MAX_GPA: 4.0,
  GRADE_SCALE: Object.freeze(['A', 'B', 'C', 'D', 'F'])
});

// Use Map for better performance with many keys
class StudentRegistry {
  constructor() {
    this.students = new Map(); // Better than object for frequent additions/deletions
    this.cache = new Map();
  }

  addStudent(id, student) {
    this.students.set(id, student);
    this.cache.clear(); // Clear cache when data changes
  }

  getStudentsByMajor(major) {
    if (this.cache.has(major)) {
      return this.cache.get(major);
    }

    const result = Array.from(this.students.values())
      .filter(student => student.major === major);

    this.cache.set(major, result);
    return result;
  }
}
```

Assignment 4: University Management System

Requirements:

Part 1: Core Data Structures

1. Create Student, Course, and Instructor classes
2. Implement inheritance hierarchy (Person → Student/Instructor)
3. Use private fields for sensitive data
4. Add comprehensive validation

Part 2: Advanced Array Operations

1. Implement search and filtering functions
2. Create reporting functions using reduce()
3. Build sorting utilities for different criteria
4. Add data aggregation and statistics

Part 3: Complex Object Management

1. Build a University class to manage all entities
2. Implement enrollment system with constraints
3. Create transcript generation
4. Add JSON import/export functionality

Code Structure Template:

javascript

```
class University {
  constructor(name) {
    // Initialize data structures
  }

  // Student management methods
  addStudent(studentData) { }
  updateStudent(id, updates) { }
  removeStudent(id) { }

  // Course management methods
  addCourse(courseData) { }
  updateCourse(id, updates) { }
  removeCourse(id) { }

  // Enrollment methods
  enrollStudent(studentId, courseId) { }
  dropStudent(studentId, courseId) { }
  assignGrade(enrollmentId, grade) { }

  // Reporting methods
  generateTranscript(studentId) { }
  getCourseRoster(courseId) { }
  getEnrollmentStatistics() { }

  // Data persistence
  exportToJSON() { }
  importFromJSON(jsonData) { }
}
```

Best Practices Summary

Array Best Practices:

1. **Use appropriate methods**: map() for transformation, filter() for selection, reduce() for aggregation
2. **Avoid mutating operations**: Use slice(), spread operator, or array methods that return new arrays
3. **Chain methods wisely**: Balance readability with performance
4. **Handle empty arrays**: Always check length before operations

Object Best Practices:

1. **Use meaningful property names**: Avoid single letters or abbreviations
2. **Implement proper validation**: Check data types and ranges
3. **Consider immutability**: Use Object.freeze() or create new objects instead of modifying
4. **Use classes for complex entities**: Organize related data and methods

Performance Tips:

1. **Cache computed values**: Store expensive calculations
2. **Use Map for frequent lookups**: Better performance than objects for large datasets
3. **Minimize object creation**: Reuse objects when possible
4. **Profile your code**: Use browser dev tools to identify bottlenecks

---

Next Module Preview

Module 5: DOM Manipulation

- Understanding the DOM tree structure
- Selecting and modifying elements
- Event handling and user interaction
- Creating dynamic web interfaces
- Building interactive applications

Preparation:

- Practice working with complex data structures
- Master array and object manipulation
- Understand class inheritance concepts
- Review HTML and CSS basics

Questions for Review

1. When should you use map() vs forEach()?
2. What are the advantages of classes over constructor functions?
3. How do private fields improve code security?

4. When is reduce() the best choice for array processing?
5. What are the trade-offs between arrays and objects for data storage?

Practice Exercises:

- Build a shopping cart system with complex calculations
- Create a gradebook with statistical analysis
- Implement a simple database-like query system
- Design a hierarchical organization structure