Module 5: DOM Manipulation

Weeks 8-9

Learning Objectives

By the end of this module, you will:

- Understand the DOM tree structure and how browsers represent HTML
- Select and modify DOM elements using various methods
- Handle user interactions through event listeners
- Create dynamic web interfaces that respond to user input
- Implement form validation and data processing
- Build interactive web applications

Understanding the DOM

What is the DOM?

- **Document Object Model**: Programming interface for HTML documents
- **Tree Structure**: Hierarchical representation of HTML elements
- **Live Object**: Changes to DOM immediately affect the rendered page
- **Language Agnostic**: Can be manipulated by JavaScript, Python, etc.

DOM Tree Structure:

```
html
```

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Student Portal</title>
  </head>
  <body>
    <header>
      <h1>University Portal</h1>
      <nav>
        <ul>
          <li><a href="#students">Students</a></li>
          <li><a href="#courses">Courses</a></li>
        </ul>
      </nav>
    </header>
    <main>
      <section id="students">
        <h2>Student List</h2>
        <div class="student-card">
          <h3>Alice Johnson</h3>
          <p>Computer Science</p>
        </div>
      </section>
    </main>
  </body>
</html>
```

DOM Node Types:

javascript

```javascript
// Different types of DOM nodes
console.log(Node.ELEMENT_NODE);      // 1 - HTML elements like <div>, <p>
console.log(Node.TEXT_NODE);         // 3 - Text content
console.log(Node.COMMENT_NODE);      // 8 - HTML comments
console.log(Node.DOCUMENT_NODE);     // 9 - The document itself

// Checking node types
const element = document.querySelector('h1');
console.log(element.nodeType);       // 1 (ELEMENT_NODE)
console.log(element.firstChild.nodeType); // 3 (TEXT_NODE)
```

Element Selection Methods

getElementById - Most Efficient:

```javascript
// Select by unique ID
const studentForm = document.getElementById('student-form');
const welcomeMessage = document.getElementById('welcome-msg');

// Returns null if not found
const nonExistent = document.getElementById('does-not-exist');
console.log(nonExistent); // null
```

getElementsByClassName - Returns HTMLCollection:

```javascript
// Select all elements with class name
const studentCards = document.getElementsByClassName('student-card');
console.log(studentCards.length); // Number of elements

// HTMLCollection is live - updates automatically
const newCard = document.createElement('div');
newCard.className = 'student-card';
document.body.appendChild(newCard);
console.log(studentCards.length); // Increased by 1

// Convert to array for array methods
const cardsArray = Array.from(studentCards);
cardsArray.forEach(card => {
  console.log(card.textContent);
});
```

getElementsByTagName - Select by Tag:

```javascript
```

```javascript
// Select all paragraphs
const allParagraphs = document.getElementsByTagName('p');

// Select all input elements
const allInputs = document.getElementsByTagName('input');

// Scope selection to specific element
const header = document.querySelector('header');
const headerLinks = header.getElementsByTagName('a');
```

querySelector - CSS Selector (Single Element):

javascript

```javascript
// Select first matching element
const firstStudent = document.querySelector('.student-card');
const courseTitle = document.querySelector('#course-title');
const firstParagraph = document.querySelector('section p');

// Complex selectors
const activeNavItem = document.querySelector('nav .active');
const requiredInputs = document.querySelector('input[required]');
const lastChild = document.querySelector('ul li:last-child');

// Attribute selectors
const emailInput = document.querySelector('input[type="email"]');
const externalLinks = document.querySelector('a[href^="http"]');
```

querySelectorAll - CSS Selector (All Elements):

javascript

```
// Select all matching elements (returns NodeList)
const allStudents = document.querySelectorAll('.student-card');
const allButtons = document.querySelectorAll('button');
const formInputs = document.querySelectorAll('#student-form input');

// NodeList supports forEach directly
allStudents.forEach((student, index) => {
  console.log(`Student ${index + 1}: ${student.textContent}`);
});

// Convert to array for other array methods
const studentsArray = Array.from(allStudents);
const studentNames = studentsArray.map(card =>
  card.querySelector('h3').textContent
);
```

Modifying Element Content

textContent vs innerHTML:

javascript

```javascript
const studentCard = document.querySelector('.student-card');

// textContent - plain text only
console.log(studentCard.textContent); // "Alice Johnson Computer Science"
studentCard.textContent = "Bob Smith Mathematics";

// innerHTML - includes HTML tags
console.log(studentCard.innerHTML); // "<h3>Alice Johnson</h3><p>Computer Science</p>"
studentCard.innerHTML = '<h3>Charlie Brown</h3><p>Physics</p><span class="gpa">3.8</span>';

// innerText - considers styling (slower)
console.log(studentCard.innerText); // Respects display:none, etc.

// Security consideration - avoid innerHTML with user input
const userInput = '<script>alert("XSS Attack")</script>';
// DON'T DO THIS:
// studentCard.innerHTML = userInput;

// DO THIS INSTEAD:
studentCard.textContent = userInput; // Treats as plain text
```

Working with Attributes:

javascript

```
const profileImage = document.querySelector('#profile-img');

// Get attribute value
const currentSrc = profileImage.getAttribute('src');
const altText = profileImage.getAttribute('alt');

// Set attribute value
profileImage.setAttribute('src', 'new-profile.jpg');
profileImage.setAttribute('alt', 'Student profile picture');

// Check if attribute exists
if (profileImage.hasAttribute('data-student-id')) {
  const studentId = profileImage.getAttribute('data-student-id');
}

// Remove attribute
profileImage.removeAttribute('title');

// Property vs Attribute
const checkbox = document.querySelector('#terms-checkbox');
checkbox.checked = true; // Property (current state)
checkbox.setAttribute('checked', 'checked'); // Attribute (initial HTML)
```

Modifying Styles:

javascript

```
const studentCard = document.querySelector('.student-card');

// Individual style properties
studentCard.style.backgroundColor = '#f0f8ff';
studentCard.style.padding = '20px';
studentCard.style.borderRadius = '8px';
studentCard.style.boxShadow = '0 2px 4px rgba(0,0,0,0.1)';

// CSS properties with hyphens become camelCase
studentCard.style.fontSize = '16px';
studentCard.style.fontWeight = 'bold';

// Get computed styles
const computedStyle = window.getComputedStyle(studentCard);
console.log(computedStyle.backgroundColor);
console.log(computedStyle.margin);

// Better approach: Use CSS classes
studentCard.classList.add('highlighted');
studentCard.classList.remove('hidden');
studentCard.classList.toggle('selected');

// Check if class exists
if (studentCard.classList.contains('active')) {
  console.log('Card is active');
}
```

Creating and Manipulating Elements

Creating New Elements:

javascript

```
// Create elements
const studentCard = document.createElement('div');
const studentName = document.createElement('h3');
const studentMajor = document.createElement('p');
const editButton = document.createElement('button');

// Set properties and content
studentCard.className = 'student-card';
studentCard.id = 'student-' + Date.now();

studentName.textContent = 'Diana Prince';
studentMajor.textContent = 'Psychology';
studentMajor.className = 'student-major';

editButton.textContent = 'Edit';
editButton.className = 'btn btn-primary';
editButton.type = 'button';

// Build structure
studentCard.appendChild(studentName);
studentCard.appendChild(studentMajor);
studentCard.appendChild(editButton);

// Add to document
const studentsContainer = document.querySelector('#students-container');
studentsContainer.appendChild(studentCard);
```

Advanced Element Creation:

javascript

```
function createStudentCard(studentData) {
  // Create container
  const card = document.createElement('div');
  card.className = 'student-card';
  card.dataset.studentId = studentData.id;

  // Create and populate elements
  const elements = {
    name: document.createElement('h3'),
    email: document.createElement('p'),
    major: document.createElement('p'),
    gpa: document.createElement('span'),
    actions: document.createElement('div')
  };

  // Set content and classes
  elements.name.textContent = studentData.name;
  elements.name.className = 'student-name';

  elements.email.textContent = studentData.email;
  elements.email.className = 'student-email';

  elements.major.textContent = studentData.major;
  elements.major.className = 'student-major';

  elements.gpa.textContent = `GPA: ${studentData.gpa.toFixed(2)}`;
  elements.gpa.className = `gpa ${studentData.gpa >= 3.5 ? 'high' : 'normal'}`;

  elements.actions.className = 'student-actions';

  // Create action buttons
  const editBtn = document.createElement('button');
  editBtn.textContent = 'Edit';
  editBtn.className = 'btn btn-secondary';
  editBtn.onclick = () => editStudent(studentData.id);

  const deleteBtn = document.createElement('button');
  deleteBtn.textContent = 'Delete';
  deleteBtn.className = 'btn btn-danger';
  deleteBtn.onclick = () => deleteStudent(studentData.id);

  elements.actions.appendChild(editBtn);
  elements.actions.appendChild(deleteBtn);
```

```javascript
    // Assemble card
    Object.values(elements).forEach(element => {
        card.appendChild(element);
    });

    return card;
}

// Usage
const newStudent = {
    id: 'STU001',
    name: 'Alice Johnson',
    email: 'alice@university.edu',
    major: 'Computer Science',
    gpa: 3.85
};

const card = createStudentCard(newStudent);
document.querySelector('#students-container').appendChild(card);
```

Template Literals for HTML:

javascript

```javascript
function createStudentCardHTML(studentData) {
    const gpaClass = studentData.gpa >= 3.5 ? 'high' : 'normal';

    const cardHTML = `
        <div class="student-card" data-student-id="${studentData.id}">
            <h3 class="student-name">${studentData.name}</h3>
            <p class="student-email">${studentData.email}</p>
            <p class="student-major">${studentData.major}</p>
            <span class="gpa ${gpaClass}">GPA: ${studentData.gpa.toFixed(2)}</span>
            <div class="student-actions">
                <button class="btn btn-secondary" onclick="editStudent('${studentData.id}')">Edit</button>
                <button class="btn btn-danger" onclick="deleteStudent('${studentData.id}')">Delete</button>
            </div>
        </div>
    `;

    // Create temporary container to convert HTML string to element
    const temp = document.createElement('div');
    temp.innerHTML = cardHTML;
    return temp.firstElementChild;
}
```

Event Handling

Adding Event Listeners:

javascript

```
// Basic event listener
const submitButton = document.querySelector('#submit-btn');
submitButton.addEventListener('click', function(event) {
    console.log('Button clicked!');
    console.log('Event type:', event.type);
    console.log('Target element:', event.target);
});

// Arrow function event handler
const cancelButton = document.querySelector('#cancel-btn');
cancelButton.addEventListener('click', (event) => {
    event.preventDefault(); // Prevent default behavior
    console.log('Operation cancelled');
});

// Multiple event listeners on same element
const inputField = document.querySelector('#student-name');
inputField.addEventListener('focus', () => {
    inputField.classList.add('focused');
});

inputField.addEventListener('blur', () => {
    inputField.classList.remove('focused');
});

inputField.addEventListener('input', (event) => {
    console.log('Current value:', event.target.value);
});
```

Event Object Properties:

```
javascript
```

```
function handleClick(event) {
    console.log('Event properties:');
    console.log('Type:', event.type);           // 'click'
    console.log('Target:', event.target);       // Element that triggered event
    console.log('Current target:', event.currentTarget); // Element with event listener
    console.log('Timestamp:', event.timeStamp); // When event occurred
    console.log('Mouse position:', event.clientX, event.clientY);

    // Prevent default behavior (for links, forms, etc.)
    event.preventDefault();

    // Stop event from bubbling up
    event.stopPropagation();
}
```

Common Event Types:

```
javascript
```

```javascript
const form = document.querySelector('#student-form');
const nameInput = document.querySelector('#name-input');
const emailInput = document.querySelector('#email-input');

// Form events
form.addEventListener('submit', handleFormSubmit);
form.addEventListener('reset', handleFormReset);

// Input events
nameInput.addEventListener('input', validateName);
nameInput.addEventListener('focus', highlightField);
nameInput.addEventListener('blur', validateField);

// Keyboard events
nameInput.addEventListener('keydown', handleKeyDown);
nameInput.addEventListener('keyup', handleKeyUp);
nameInput.addEventListener('keypress', handleKeyPress);

// Mouse events
const card = document.querySelector('.student-card');
card.addEventListener('click', selectCard);
card.addEventListener('dblclick', editCard);
card.addEventListener('mouseenter', showTooltip);
card.addEventListener('mouseleave', hideTooltip);

function handleFormSubmit(event) {
    event.preventDefault();

    // Get form data
    const formData = new FormData(form);
    const studentData = {
        name: formData.get('name'),
        email: formData.get('email'),
        major: formData.get('major')
    };

    // Validate and process
    if (validateStudentData(studentData)) {
        addStudent(studentData);
        form.reset();
    }
}
```

```
function handleKeyDown(event) {
  // Enter key to submit
  if (event.key === 'Enter' && event.ctrlKey) {
    form.dispatchEvent(new Event('submit'));
  }

  // Escape key to cancel
  if (event.key === 'Escape') {
    form.reset();
  }
}
```

Event Delegation

Why Use Event Delegation?

- **Performance**: One listener instead of many
- **Dynamic Content**: Works with elements added later
- **Memory Efficient**: Fewer event listeners

Implementation:

```
javascript
```

```
// Instead of adding listeners to each button individually
// Add one listener to the parent container
const studentsContainer = document.querySelector('#students-container');

studentsContainer.addEventListener('click', function(event) {
  const target = event.target;
  const studentCard = target.closest('.student-card');

  if (!studentCard) return; // Click wasn't on a student card

  const studentId = studentCard.dataset.studentId;

  // Handle different button clicks
  if (target.classList.contains('edit-btn')) {
    editStudent(studentId);
  } else if (target.classList.contains('delete-btn')) {
    deleteStudent(studentId);
  } else if (target.classList.contains('view-btn')) {
    viewStudent(studentId);
  }
});

// Functions for handling actions
function editStudent(studentId) {
  console.log('Editing student:', studentId);
  // Show edit modal or inline editing
}

function deleteStudent(studentId) {
  if (confirm('Are you sure you want to delete this student?')) {
    const studentCard = document.querySelector(`[data-student-id="${studentId}"]`);
    studentCard.remove();
    console.log('Student deleted:', studentId);
  }
}

function viewStudent(studentId) {
  console.log('Viewing student details:', studentId);
  // Show detailed view or navigate to profile page
}
```

Advanced Event Delegation:

```javascript
```

```
// Comprehensive event delegation for multiple event types
class StudentInterface {
  constructor(containerSelector) {
    this.container = document.querySelector(containerSelector);
    this.setupEventListeners();
  }

  setupEventListeners() {
    // Delegate click events
    this.container.addEventListener('click', (event) => {
      this.handleClick(event);
    });

    // Delegate input events for inline editing
    this.container.addEventListener('input', (event) => {
      this.handleInput(event);
    });

    // Delegate focus events
    this.container.addEventListener('focus', (event) => {
      this.handleFocus(event);
    }, true); // Use capture phase for focus events
  }

  handleClick(event) {
    const { target } = event;
    const action = target.dataset.action;
    const studentCard = target.closest('.student-card');

    if (!studentCard || !action) return;

    const studentId = studentCard.dataset.studentId;

    switch (action) {
      case 'edit':
        this.editStudent(studentId, studentCard);
        break;
      case 'delete':
        this.deleteStudent(studentId, studentCard);
        break;
      case 'save':
        this.saveStudent(studentId, studentCard);
        break;
```

```
            case 'cancel':
                this.cancelEdit(studentId, studentCard);
                break;
        }
    }

    handleInput(event) {
        const { target } = event;
        if (target.classList.contains('editable-field')) {
            this.validateField(target);
        }
    }

    editStudent(studentId, card) {
        // Make fields editable
        const nameElement = card.querySelector('.student-name');
        const majorElement = card.querySelector('.student-major');

        nameElement.innerHTML = `<input type="text" value="${nameElement.textContent}" class="editable-field" data-f
        majorElement.innerHTML = `<input type="text" value="${majorElement.textContent}" class="editable-field" data-

        // Change buttons
        const actionsDiv = card.querySelector('.student-actions');
        actionsDiv.innerHTML = `
            <button class="btn btn-success" data-action="save">Save</button>
            <button class="btn btn-secondary" data-action="cancel">Cancel</button>
        `;
    }

    saveStudent(studentId, card) {
        const nameInput = card.querySelector('[data-field="name"]');
        const majorInput = card.querySelector('[data-field="major"]');

        // Validate inputs
        if (!this.validateStudentData(nameInput.value, majorInput.value)) {
            return;
        }

        // Save data (would typically involve API call)
        const updatedData = {
            id: studentId,
            name: nameInput.value,
            major: majorInput.value
        };
        // Update display
        card.querySelector('.student-name').textContent = updatedData.name;
        card.querySelector('.student-major').textContent = updatedData.major;

        // Restore action buttons
        this.restoreActionButtons(card);

        console.log('Student updated:', updatedData);
    }

    validateStudentData(name, major) {
        if (!name.trim()) {
            alert('Name is required');
            return false;
        }

        if (!major.trim()) {
            alert('Major is required');
            return false;
        }

        return true;
    }

    restoreActionButtons(card) {
        const actionsDiv = card.querySelector('.student-actions');
        actionsDiv.innerHTML = `
            <button class="btn btn-secondary" data-action="edit">Edit</button>
            <button class="btn btn-danger" data-action="delete">Delete</button>
        `;
    }
}

// Initialize the interface
const studentInterface = new StudentInterface('#students-container');
```

Form Handling and Validation

Real-time Validation:

```javascript
```

```
class FormValidator {
  constructor(formSelector) {
    this.form = document.querySelector(formSelector);
    this.errors = new Map();
    this.setupValidation();
  }

  setupValidation() {
    // Validate on input (real-time)
    this.form.addEventListener('input', (event) => {
      this.validateField(event.target);
    });

    // Validate on blur
    this.form.addEventListener('blur', (event) => {
      this.validateField(event.target);
    }, true);

    // Handle form submission
    this.form.addEventListener('submit', (event) => {
      event.preventDefault();
      this.validateForm();
    });
  }

  validateField(field) {
    const value = field.value.trim();
    const fieldName = field.name;
    let isValid = true;
    let errorMessage = '';

    // Clear previous error
    this.clearFieldError(field);

    // Required field validation
    if (field.hasAttribute('required') && !value) {
      isValid = false;
      errorMessage = `${this.getFieldLabel(field)} is required`;
    }

    // Specific field validations
    switch (fieldName) {
      case 'email':
            if (value && !this.isValidEmail(value)) {
                isValid = false;
                errorMessage = 'Please enter a valid email address';
            }
            break;

        case 'gpa':
            const gpa = parseFloat(value);
            if (value && (isNaN(gpa) || gpa < 0 || gpa > 4.0)) {
                isValid = false;
                errorMessage = 'GPA must be between 0.0 and 4.0';
            }
            break;

        case 'name':
            if (value && value.length < 2) {
                isValid = false;
                errorMessage = 'Name must be at least 2 characters long';
            }
            break;

        case 'phone':
            if (value && !this.isValidPhone(value)) {
                isValid = false;
                errorMessage = 'Please enter a valid phone number';
            }
            break;
    }

    if (!isValid) {
        this.showFieldError(field, errorMessage);
        this.errors.set(fieldName, errorMessage);
    } else {
        this.errors.delete(fieldName);
    }

    this.updateSubmitButton();
    return isValid;
}

validateForm() {
    const inputs = this.form.querySelectorAll('input, select, textarea');
    let isFormValid = true;
```

```
        inputs.forEach(input => {
            if (!this.validateField(input)) {
                isFormValid = false;
            }
        });

        if (isFormValid) {
            this.submitForm();
        } else {
            this.showFormErrors();
        }
    }

    submitForm() {
        const formData = new FormData(this.form);
        const data = Object.fromEntries(formData.entries());

        console.log('Submitting form data:', data);

        // Here you would typically send data to server
        this.showSuccessMessage('Form submitted successfully!');
        this.form.reset();
        this.errors.clear();
        this.updateSubmitButton();
    }

    // Utility methods
    isValidEmail(email) {
        const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        return emailRegex.test(email);
    }

    isValidPhone(phone) {
        const phoneRegex = /^\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$/;
        return phoneRegex.test(phone);
    }

    getFieldLabel(field) {
        const label = this.form.querySelector(`label[for="${field.id}"]`);
        return label ? label.textContent : field.name;
    }

    showFieldError(field, message) {
        field.classList.add('error');
        // Create or update error message
        let errorDiv = field.parentNode.querySelector('.error-message');
        if (!errorDiv) {
            errorDiv = document.createElement('div');
            errorDiv.className = 'error-message';
            field.parentNode.appendChild(errorDiv);
        }
        errorDiv.textContent = message;
    }

    clearFieldError(field) {
        field.classList.remove('error');
        const errorDiv = field.parentNode.querySelector('.error-message');
        if (errorDiv) {
            errorDiv.remove();
        }
    }

    updateSubmitButton() {
        const submitButton = this.form.querySelector('button[type="submit"]');
        submitButton.disabled = this.errors.size > 0;
    }

    showSuccessMessage(message) {
        const successDiv = document.createElement('div');
        successDiv.className = 'success-message';
        successDiv.textContent = message;

        this.form.insertBefore(successDiv, this.form.firstChild);

        setTimeout(() => {
            successDiv.remove();
        }, 3000);
    }

    showFormErrors() {
        const errorMessages = Array.from(this.errors.values());
        alert('Please fix the following errors:\n' + errorMessages.join('\n'));
    }
}
```

```
// Initialize form validation
const validator = new FormValidator('#student-form');
```

Assignment 5: Interactive Student Portal

Requirements:

Part 1: Dynamic Content Management

1. Create functions to add, edit, and remove student cards
2. Implement real-time search and filtering
3. Add sorting capabilities (by name, GPA, major)
4. Create modal dialogs for detailed views

Part 2: Form Handling

1. Build comprehensive form validation
2. Implement auto-save functionality
3. Add form field dependencies (conditional fields)
4. Create bulk operations interface

Part 3: User Interface Enhancements

1. Add loading states and progress indicators
2. Implement keyboard navigation
3. Create responsive design elements
4. Add animations and transitions

Code Structure Template:

```
html
```

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Student Portal</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <div id="student-portal">
        <header>
            <h1>Student Management Portal</h1>
            <nav>
                <!-- Navigation elements -->
            </nav>
        </header>

        <main>
            <section id="controls">
                <!-- Search, filter, sort controls -->
            </section>

            <section id="students-container">
                <!-- Dynamic student cards -->
            </section>
        </main>

        <div id="modal-container">
            <!-- Modal dialogs -->
        </div>
    </div>

    <script src="student-portal.js"></script>
</body>
</html>
```

Best Practices Summary

DOM Selection:

1. **Use specific selectors**: getElementById is fastest, querySelectorAll for complex selections

2. **Cache DOM references**: Store frequently accessed elements in variables
3. **Minimize DOM queries**: Query once, store the result
4. **Use event delegation**: For dynamic content and better performance

Element Manipulation:

1. **Batch DOM updates**: Use DocumentFragment for multiple insertions
2. **Avoid layout thrashing**: Read all measurements first, then make changes
3. **Use CSS classes**: Instead of inline styles for better maintainability
4. **Validate user input**: Always sanitize and validate before using in DOM

Event Handling:

1. **Use event delegation**: For dynamic content and performance
2. **Remove event listeners**: When elements are removed to prevent memory leaks
3. **Debounce input events**: For search and validation to improve performance
4. **Handle keyboard navigation**: Make interfaces accessible

---

Next Module Preview

Module 6: Asynchronous JavaScript

- Understanding the event loop and call stack
- Working with callbacks and callback hell
- Mastering Promises and async/await
- Fetching data from APIs
- Handling asynchronous errors

Preparation:

- Practice DOM manipulation techniques
- Build interactive interfaces
- Understand event handling patterns
- Review JavaScript fundamentals

Questions for Review

1. What's the difference between textContent and innerHTML?
2. When should you use event delegation?
3. How do you prevent memory leaks with event listeners?
4. What are the benefits of using DocumentFragment?
5. How can you make DOM manipulation more performant?

Practice Exercises:

- Build a dynamic todo list application
- Create an interactive photo gallery
- Implement a drag-and-drop interface
- Design a real-time search interface