

Advanced JavaScript Concepts

Optional Deep Dive Material for Session 2 (Optional)

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-14

Abstract Course lectures and practices for JavaScript full-stack web development with AI-assisted workflows.

Table of contents

0.1	Optional Reading: Advanced DOM & Patterns	2	0.3.a	reduce() - The Swiss Army Knife	6
0.1.a	Part 3: DOM Intro	2	0.3.b	find() and findIndex()	6
0.1.b	querySelector: The Only Selector You Need	3	0.3.c	some() and every()	7
0.1.c	Updating Content: textContent	3	1	Advanced Error Handling	7
0.1.d	Event Listeners: Making Things Interactive	3	1.1	Try-Catch Patterns	7
0.1.e	AI Demo: Button + Fetch . .	4	1.1.a	Async Error Handling	7
0.1.f	The Vanilla JS Pattern	4	1.1.b	Error Boundaries in Vanilla JS .	7
0.1.g	What We're NOT Teaching About DOM	5	2	Browser Storage APIs	8
0.1.h	✗ Skipping (React Does This) .	5	2.1	localStorage and sessionStorage	8
0.1.i	Optional Why Skip?	5	2.1.a	Basic Usage	8
0.1.j	Key Takeaways: DOM Basics . .	5	2.1.b	Shopping Cart Implementation	8
0.1.k	Why React is Better	5	3	Advanced DOM Manipulation	9
0.1.l	Vanilla JS Problems	5	3.1	Event Delegation	9
0.1.m	React Solutions	5	3.2	Creating Complex Elements	9
0.1.n	Imagine Adding More Features... ..	6	4	Performance Optimization	10
0.2	Advanced Array Methods	6	4.1	Debouncing and Throttling	10
0.3	Beyond map() and filter()	6	4.1.a	Search Input Debouncing	10
			4.1.b	Scroll Event Throttling	10
			5	Modern JavaScript Features	10
			5.1	Optional Chaining	10
			5.2	Nullish Coalescing	11
			5.3	Destructuring Patterns	11

i Optional Advanced Material

This document contains **advanced JavaScript concepts** that were moved from the main Session 2 lecture to keep the core session focused and manageable.

Prerequisites: Complete Session 2 main content first.

When to read: When you're comfortable with JavaScript basics and want to deepen your understanding before or after Session 3.

0.1 Optional Reading: Advanced DOM & Patterns

💡 Optional Content - Read Later

The following content contains valuable DOM manipulation concepts, but they're **optional** for this session due to time constraints.

When to read: After Session 3 (React), when you want to understand what React does behind the scenes.

File reference: See `webdev_02_L_02C_JavaScriptAdvanced.qmd` for more advanced patterns.

0.1.a Part 3: DOM Intro

Original 30-Minute Content (Now Optional)

HTML (What You Write)

```
<!DOCTYPE html>
<html>
<head>
  <title>My Page</title>
```

```
</head>
<body>
  <h1 id="title">Hello</h1>
  <button id="btn">Click</button>
</body>
</html>
```

DOM (Browser's View)

```
// Browser creates JavaScript objects
document = {
  head: { /* ... */ },
  body: {
    children: [
      {type: "h1", id: "title", ...},
      {type: "button", id: "btn", ...}
    ]
  }
}

// You can manipulate these objects!
```

DOM = JavaScript's way to access and modify HTML

0.1.b querySelector: The Only Selector You Need

i Modern Element Selection

```
// Select by ID
const title = document.querySelector('#title');

// Select by class
const button = document.querySelector('.btn-primary');

// Select by tag
const firstDiv = document.querySelector('div');

// Complex selectors (like CSS!)
const item = document.querySelector('ul > li:first-child');
const button = document.querySelector('button[id="5"]');
```

Why just querySelector? - Works for everything (ID, class, tag, complex) - Same syntax as CSS selectors - Modern and flexible

Old methods (getElementById, getElementsByClassName) - forget them!

0.1.c Updating Content: textContent

☒ Safe Way: textContent

```
const title = document.querySelector('#title');

// Read content
console.log(title.textContent);
// "Hello"

// Update content (safe!)
```

```
title.textContent = "New Title";

// Even with user input (safe!)
const userInput = "<script>alert('hack')</script>";
title.textContent = userInput;
// Shows literal text, not code
```

⚠ Dangerous: innerHTML

```
const container = document.querySelector('#container');

// Read HTML
console.log(container.innerHTML);
// "<p>Hello</p>"

// Update HTML (dangerous!)
container.innerHTML = "<p>New</p>";

// User input (DANGEROUS!)
const userInput = "<script>alert('hack')</script>";
container.innerHTML = userInput;
// Executes the script! ⚠
```

Only use innerHTML with trusted content!

0.1.d Event Listeners: Making Things Interactive

```
// 1. Select the element
const button = document.querySelector('#loadButton');

// 2. Add event listener
button.addEventListener('click', () => {
  console.log('Button clicked!');
});

// With async function
button.addEventListener('click', async () => {
  const data = await fetch('https://')
```

```

api.example.com/data');
  const json = await data.json();
  console.log(json);
});

// Form submit
const form
= document.querySelector('#myForm');
form.addEventListener('submit',
(event) => {
  event.preventDefault(); // Stop
form from reloading page!
  console.log('Form submitted');
});

```

Pattern: `querySelector` + `addEventListener`
 = Interactive page

0.1.e 🤖 AI Demo: Button + Fetch

i Let's Make a Button Work!

Prompt to AI:

Create JavaScript code that:

1. Selects a button with id "loadBtn"
2. Adds a click event listener
3. When clicked, fetches a random user from `https://jsonplaceholder.typicode.com/users/1`
4. Displays the user's name in an element with id "output"
5. Uses `async/await` and `textContent`

HTML Setup (try in CodePen):

```

<button id="loadBtn">Load User</button>
<div id="output"></div>

```

Everyone try this - see vanilla JS in action!

0.1.f The Vanilla JS Pattern

// This is what you'll do in vanilla JavaScript:

```

// 1. Select elements
const button =
document.querySelector('#loadBtn');
const output
= document.querySelector('#output');

// 2. Add event listener
button.addEventListener('click',
async () => {

  // 3. Fetch data

```

```

const response =
await fetch('https://api.example.
com/products');
const products = await
response.json();

// 4. Update DOM manually (tedious!)
output.textContent = ''; // Clear
first
products.forEach(product => {
const div =
document.createElement('div');
div.textContent = `${product.name}
- ${product.price}`;
output.appendChild(div);
});
});

```

Notice: Manual DOM manipulation is tedious! Imagine 100 products, multiple updates...

0.1.g What We're NOT Teaching About DOM

0.1.h ✗ Skipping (React Does This)

- DOM tree navigation (parent, child, sibling)
- `getElementById`, `getElementsByClassName`
- `createElement`, `appendChild` patterns
- Event delegation
- Event propagation (capture/bubble)
- Form handling details
- `innerHTML` (dangerous anyway)
- `localStorage` (later project)
- Performance optimization

0.1.i ✗ Why Skip?

React replaces ALL of this!

```

// Vanilla JS: 20 lines
const div
= document.createElement('div');
div.textContent = product.name;
div.addEventListener('click',
handler);

```

```

document.body.appendChild(div);

// React: 1 line
<div onClick={handler}>
  {product.name}</div>

```

You'll learn the React way next week

0.1.j Key Takeaways: DOM Basics

1. **DOM** = JavaScript's view of HTML - You can change it
2. **querySelector** - The only selector you need
3. **textContent** - Safe way to update content
4. **addEventListener** - Make things interactive
5. **event.preventDefault()** - Stop default behavior (forms)
6. **This is tedious!** - React makes it 10x easier

0.1.k Why React is Better

0.1.l Vanilla JS Problems

```

// Manual DOM updates
productsDiv.innerHTML = '';
products.forEach(product => {
const div = document
.createElement('div');
div.textContent = product.name;
productsDiv.appendChild(div);
});

// Managing state manually
let isLoading = false;
let products = [];
let error = null;

// No automatic re-rendering
// You update DOM yourself!

```

0.1.m React Solutions

```

// Declarative rendering
return products.map(product => (

```

```

    <div>{product.name}</div>
  ));

  // State management built-in
  const [products, setProducts]
    = useState([]);

  // Automatic re-rendering
  setProducts(newProducts);
  // React updates DOM automatically!

  // Component reusability
  <ProductCard      product={p}
    onAddToCart={handleAdd} />

```

0.1.n Imagine Adding More Features...

Vanilla JS gets exponentially harder: - ✗

Add filter by price? Manual DOM rebuild - ✗

✗ Add sorting? More DOM manipulation - ✗

Add shopping cart? Track state everywhere - ✗

Add product details? More event listeners - ✗

Update individual items? Find element, update manually - ✗ Add animations? Complex DOM manipulation

React stays manageable: - ☑ Filter: `products.filter(...)` in render - ☑ Sort: `products.sort(...)` in render - ☑ Cart: `const [cart, setCart] = useState([])` - ☑ Details: `<ProductDetails product={selected} />` - ☑ Update: `setProducts(...)` - React handles DOM! - ☑ Animations: Built-in transition tools

0.2 Advanced Array Methods

0.3 Beyond map() and filter()

While Session 2 covers basic `map()` and `filter()`, here are additional powerful array methods:

0.3.a reduce() - The Swiss Army Knife

```

// Calculate total price
const prices = [19.99, 29.99, 39.99];
const total = prices.reduce((sum,
price) => sum + price, 0);
// Result: 89.97

// Group reviews by rating
const reviews = [
  { rating: 5, comment: "Great!" },
  { rating: 4, comment: "Good" },
  { rating: 5, comment: "Excellent!" }
];

const      groupedByRating      =
reviews.reduce((groups, review) => {
  const rating = review.rating;
  if (!groups[rating]) groups[rating]
    = [];
  groups[rating].push(review);
  return groups;
}, {});
// Result: { 4: [...], 5: [...] }

```

0.3.b find() and findIndex()

```

// Find specific review
const reviews = [
  { id: 1, rating: 5 },
  { id: 2, rating: 3 },
  { id: 3, rating: 4 }
];

const      highRating      =
reviews.find(review => review.rating
>= 4);
// Result: { id: 1, rating: 5 }

const
index = reviews.findIndex(review =>
review.id === 2);
// Result: 1

```

0.3.c some() and every()

```
// Check if any review is 5 stars
const hasExcellentReview =
  reviews.some(review => review.rating
    === 5);

// Check if all reviews are positive
(>= 3)
const
allPositive = reviews.every(review =>
  review.rating >= 3);
```

1 Advanced Error Handling

1.1 Try-Catch Patterns

1.1.a Async Error Handling

```
// Basic pattern (covered in Session
2)
async function fetchReviews() {
  try {
    const response = await fetch('/
api/reviews');
    const data = await
response.json();
    return data;
  } catch (error) {
    console.error('Error:', error);
    return [];
  }
}

// Advanced pattern with specific
error types
async function
fetchReviewsAdvanced(productId) {
  try {
    const response = await fetch(`/
api/products/${productId}/reviews`);

    if (!response.ok) {
      if (response.status === 404) {
```

```
        throw new Error('Product
not found');
      } else if (response.status ===
500) {
        throw new Error('Server error
- please try again later');
      } else {
        throw
new Error(`HTTP ${response.status}:
${response.statusText}`);
      }
    }

    const data = await
response.json();
    return data;
  } catch (error) {
    if (error instanceof TypeError) {
      // Network error
      throw new Error('Network error
- check your connection');
    }
    throw error; // Re-throw other
errors
  }
}
```

1.1.b Error Boundaries in Vanilla JS

```
// Global error handler
window.addEventListener('error',
(event) => {
  console.error('Global error:',
event.error);
  // Send to logging service
  // Show user-friendly message
});

// Unhandled promise rejection
handler
window.addEventListener('unhandledrejection',
(event) => {
  console.error('Unhandled promise
```

```

rejection:', event.reason);
  event.preventDefault(); // Prevent
  console.error
});

```

2 Browser Storage APIs

2.1 localStorage and sessionStorage

2.1.a Basic Usage

```

// Save user preferences
const userPrefs = {
  theme: 'dark',
  itemsPerPage: 10,
  sortBy: 'rating'
};

localStorage.setItem('userPrefs',
JSON.stringify(userPrefs));

// Retrieve preferences
const savedPrefs = JSON.parse(localStorage
|| '{}');

```

2.1.b Shopping Cart Implementation

```

class ShoppingCart {
  constructor() {
    this.items =
  this.loadFromStorage();
  }

  addItem(product) {
    const existingItem =
  this.items.find(item => item.id ===
product.id);

    if (existingItem) {
      existingItem.quantity += 1;
    } else {
      this.items.push({ ...product,

```

```

quantity: 1 }));
    }

    this.saveToStorage();
    this.updateUI();
  }

  removeItem(productId) {
    this.items =
  this.items.filter(item => item.id !
== productId);
    this.saveToStorage();
    this.updateUI();
  }

  loadFromStorage() {
    try {
      return JSON.parse(localStorage.getItem('cart')
|| '[]');
    } catch (error) {
      console.error('Error loading
cart:', error);
      return [];
    }
  }

  saveToStorage() {
    try {
      localStorage.setItem('cart',
JSON.stringify(this.items));
    } catch (error) {
      console.error('Error saving
cart:', error);
    }
  }

  updateUI() {
    // Update cart display
    const cartElement
= document.querySelector('#cart-
items');
    if (cartElement) {
      cartElement.innerHTML =

```

```

this.items.map(item => `
    <div class="cart-item">
      <span>${item.name}</span>
      <span>Qty: ${item.quantity}</span>
      <span>${(item.price * item.quantity).toFixed(2)}</span>
    </div>
`).join('');
}
}

// Usage
const cart = new ShoppingCart();

```

3 Advanced DOM Manipulation

3.1 Event Delegation

Instead of adding event listeners to every button:

```

// ✗ Bad: Multiple event listeners
document.querySelectorAll('.review-like-btn').forEach(btn => {
  btn.addEventListener('click', handleLike);
});

// ☑ Good: Single delegated event listener
document.addEventListener('click', (event) => {
  if (event.target.matches('.review-like-btn')) {
    handleLike(event);
  }
});

```

3.2 Creating Complex Elements

```

function createReviewCard(review) {
  const card = document.createElement('div');
  card.className = 'review-card bg-white p-4 rounded-lg shadow';

  // Create star rating
  const stars = Array.from({ length: 5 }, (_, i) => {
    const star = document.createElement('span');
    star.className = i < review.rating ? 'star-filled' : 'star-empty';
    star.textContent = '★';
    return star;
  });

  const starsContainer = document.createElement('div');
  starsContainer.className = 'flex gap-1';
  starsContainer.append(...stars);

  // Create content
  const title = document.createElement('h3');
  title.textContent = review.title;
  title.className = 'font-bold text-lg';

  const content = document.createElement('p');
  content.textContent = review.content;
  content.className = 'text-gray-700 mt-2';

  const author = document.createElement('div');
  author.textContent = `By ${review.author} on ${review.date}`;

```

```

    author.className = 'text-sm text-gray-500 mt-2';

    // Assemble card
    card.append(starsContainer, title,
    content, author);

    return card;
}

```

4 Performance Optimization

4.1 Debouncing and Throttling

4.1.a Search Input Debouncing

```

function debounce(func, delay) {
    let timeoutId;
    return function (...args) {
        clearTimeout(timeoutId);
        timeoutId = setTimeout(() =>
func.apply(this, args), delay);
    };
}

// Usage for search
const      searchInput      =
document.querySelector('#search');
const      debouncedSearch  =
debounce(performSearch, 300);

searchInput.addEventListener('input',
(event) => {

    debouncedSearch(event.target.value);
});

function performSearch(query) {
    console.log('Searching for:',
query);
    // Perform actual search
}

```

4.1.b Scroll Event Throttling

```

function throttle(func, delay) {
    let timeoutId;
    let lastExecTime = 0;

    return function (...args) {
        const currentTime = Date.now();

        if (currentTime - lastExecTime >
delay) {
            func.apply(this, args);
            lastExecTime = currentTime;
        } else {
            clearTimeout(timeoutId);
            timeoutId = setTimeout(() => {
                func.apply(this, args);
                lastExecTime = Date.now();
            }, delay - (currentTime -
lastExecTime));
        }
    };
}

// Usage for scroll events
const      throttledScrollHandler  =
throttle(handleScroll, 100);
window.addEventListener('scroll',
throttledScrollHandler);

```

5 Modern JavaScript Features

5.1 Optional Chaining

```

// Safe property access
const      userCity
= user?.profile?.address?.city;

// Safe method calls
const result = api.getData?.();

// Safe array access
const firstReview = reviews?.[0];

```

5.2 Nullish Coalescing

```
// Use default only for null/undefined
const username = user.name ?? 'Anonymous';
const port = process.env.PORT ?? 3000;

// Different from || operator
const count = 0;
console.log(count || 10); // 10 (falsy)
console.log(count ?? 10); // 0 (not nullish)
```

5.3 Destructuring Patterns

```
// Array destructuring with rest
const [first, second, ...rest] = reviews;

// Object destructuring with renaming
const { title: reviewTitle, rating: stars } = review;

// Nested destructuring
const { user: { name, profile: { age } } } = reviewData;

// Function parameter destructuring
function displayReview({ title, rating, author = 'Anonymous' }) {
  console.log(`${title} - ${rating} stars by ${author}`);
}
```

6 When to Use These Advanced Features

In React Development: - **Array methods:** Essential for rendering lists and data manipulation - **Error handling:** Critical for user experience - **Storage APIs:** Less needed (React has

state management) - **DOM manipulation:** Rarely needed (React handles DOM) - **Performance:** React has built-in optimizations

Recommendation: Focus on array methods and error handling first. The other topics become more relevant in specialized scenarios or when working outside React.