DOM Manipulation & Game Development Insights

> Notes from building interactive games: Guess My Number, Modal, and Pig Game

> 🎓 **CS Perspective:** The DOM (Document Object Model) is a **tree data structure** that represents HTML documents. Manipulating the DOM involves **tree traversal** and modification operations. Event handling implements the **Observer pattern**—a behavioral design pattern where objects (listeners) subscribe to events from a subject (DOM elements). The concept of "event-driven programming" is fundamental to GUI systems and differs from sequential execution. Understanding the DOM also introduces you to the browser's **rendering pipeline** and why efficient DOM manipulation matters for performance.

📌 Selecting DOM Elements

```
// Select by class
document.querySelector('.message');

// Select by ID
document.getElementById('score--0');

// Select multiple elements
document.querySelectorAll('.show-modal');
```

Method	Returns	Use Case
`querySelector()`	First match	Single element
`querySelectorAll()`	NodeList	Multiple elements
`getElementById()`	Element	By ID (faster)

📌 Reading & Changing Content

Text Content

```
// Read
document.querySelector('.message').textContent;

// Write
document.querySelector('.message').textContent = 'Correct! 🎉 ';
```

Input Values

```
// Read input value
document.querySelector('.guess').value;

// Set input value
document.querySelector('.guess').value = '';
```

innerHTML

```
// Insert HTML structure
containerMovements.innerHTML = '<div>New content</div>';
```

📌 Manipulating CSS Styles

```
// Change inline styles (use camelCase)
document.querySelector('body').style.backgroundColor = '#60b347';
document.querySelector('.number').style.width = '30rem';
```

⚠ **Note:** Inline styles have highest specificity. Values must be strings.

📌 Working with Classes

```
const modal = document.querySelector('.modal');

// Add class
modal.classList.add('hidden');

// Remove class
modal.classList.remove('hidden');

// Toggle class (add if missing, remove if present)
modal.classList.toggle('player--active');

// Check if has class
modal.classList.contains('hidden'); // true/false
```

Why Use Classes Over Inline Styles?

- ✅ Separation of concerns (CSS in stylesheets)
- ✅ Can change multiple properties at once
- ✅ Easier to maintain
- ✅ Better performance

📌 Event Handling

addEventListener

```
document.querySelector('.btn').addEventListener('click', function() {
  console.log('Button clicked!');
});
```

Event Types

Event	Triggers When
`click`	Element clicked
`keydown`	Key pressed
`keyup`	Key released
`load`	Page loaded
`submit`	Form submitted

Keyboard Events

```
document.addEventListener('keydown', function(e) {
  console.log(e.key); // Which key was pressed

  if (e.key === 'Escape') {
    closeModal();
  }
});
```

📌 The Event Object

```
document.addEventListener('keydown', function(e) {
  console.log(e);         // Full event object
  console.log(e.key);     // 'Escape', 'Enter', 'a', etc.
  console.log(e.code);    // 'Escape', 'KeyA', etc.
  console.log(e.target); // Element that triggered event
});
```

📌 State Management Pattern

Store application state in variables, then update DOM based on state:

```
// State variables
let secretNumber = Math.trunc(Math.random() * 20) + 1;
let score = 20;
let highscore = 0;
let currentScore = 0;
let activePlayer = 0;
let playing = true;

// Update state, then update DOM
score--;
document.querySelector('.score').textContent = score;
```

Why This Pattern?

- Single source of truth
- Predictable updates
- Easy to reset game state

📌 Refactoring: DRY Principle

Before (Repetitive)

```
document.querySelector('.message').textContent = '📈 Too high!';
document.querySelector('.message').textContent = '📉 Too low!';
document.querySelector('.message').textContent = '🎉 Correct!';
```

After (DRY with Helper Function)

```
const displayMessage = function(message) {
  document.querySelector('.message').textContent = message;
};

displayMessage('📈 Too high!');
displayMessage('📉 Too low!');
displayMessage('🎉 Correct!');
```

📌 Game Reset / Init Pattern

Encapsulate initial state in a function:

```
const init = function() {
  // Reset state
  scores = [0, 0];
```

```javascript
  currentScore = 0;
  activePlayer = 0;
  playing = true;

  // Reset DOM
  score0El.textContent = 0;
  score1El.textContent = 0;
  diceEl.classList.add('hidden');
  player0El.classList.remove('player--winner');
  player1El.classList.remove('player--winner');
};

// Call on game start
init();

// Call on "New Game" button
btnNew.addEventListener('click', init);
```

📌 Toggle Between Players

```javascript
const switchPlayer = function() {
  // Reset current player's score display
  document.getElementById(`current--${activePlayer}`).textContent = 0;
  currentScore = 0;

  // Switch player (0 → 1 or 1 → 0)
  activePlayer = activePlayer === 0 ? 1 : 0;

  // Toggle active class on both players
  player0El.classList.toggle('player--active');
  player1El.classList.toggle('player--active');
};
```

📌 Random Number Generation

```javascript
// Random integer 1-6 (dice)
const dice = Math.trunc(Math.random() * 6) + 1;

// Random integer 1-20
const secretNumber = Math.trunc(Math.random() * 20) + 1;

// Formula: Math.trunc(Math.random() * max) + 1
```

📌 Dynamic Element Selection

Use template literals for dynamic IDs/classes:

```
// Select element based on variable
document.getElementById(`score--${activePlayer}`);
document.querySelector(`.player--${activePlayer}`);

// Dynamic image source
diceEl.src = `dice-${dice}.png`;
```

📌 Guard Clauses with `playing` Flag

Prevent actions after game ends:

```
let playing = true;

btnRoll.addEventListener('click', function() {
  if (playing) {
    // Game logic here

    if (winCondition) {
      playing = false; // Disable further moves
    }
  }
});
```

📌 Multiple Event Listeners for Same Action

```
const closeModal = function() {
  modal.classList.add('hidden');
  overlay.classList.add('hidden');
};

// Multiple ways to close modal
btnCloseModal.addEventListener('click', closeModal);
overlay.addEventListener('click', closeModal);
document.addEventListener('keydown', function(e) {
  if (e.key === 'Escape' && !modal.classList.contains('hidden')) {
    closeModal();
  }
});
```

📌 Loop Through NodeList

```
const btnsOpenModal = document.querySelectorAll('.show-modal');

for (let i = 0; i < btnsOpenModal.length; i++) {
  btnsOpenModal[i].addEventListener('click', openModal);
}
```

🧪 Quick Reference Cheatsheet

```
// DOM Selection
document.querySelector('.class');
document.getElementById('id');
document.querySelectorAll('.class');

// Content
element.textContent = 'text';
element.innerHTML = '<div>html</div>';
inputElement.value = 'value';

// Styles
element.style.backgroundColor = 'red';
element.style.display = 'none';

// Classes
element.classList.add('class');
element.classList.remove('class');
element.classList.toggle('class');
element.classList.contains('class');

// Events
element.addEventListener('click', handler);
document.addEventListener('keydown', handler);

// Event object
e.key           // 'Escape', 'Enter'
e.target        // element that triggered
e.preventDefault() // stop default behavior
```

✅ Key Takeaways

1. **Separate state from DOM** - Store data in variables, update DOM from state
2. **Use classes over inline styles** - Better maintainability
3. **Create helper functions** - Avoid repetition (DRY)
4. **Use init functions** - Easy game reset
5. **Guard clauses** - Prevent invalid actions with flags
6. **Dynamic selectors** - Template literals for flexible selection
7. **Always use `'use strict'`** - Catch errors early

8. **Multiple listeners, one handler** - Same function for different events
9. **classList methods** - Prefer over style manipulation
10. **Event delegation** - Add listener to parent for multiple children