Pig Game Project

> 🎓 **CS Perspective:** This project implements a **finite state machine (FSM)** with multiple states: playing, player 1's turn, player 2's turn, and game over. The `playing` boolean acts as a **guard flag** preventing actions in invalid states. The game state is stored in variables (`scores`, `currentScore`, `activePlayer`)—this is **state management** at its simplest. Using an array index (`scores[activePlayer]`) to access player-specific data demonstrates **indirect addressing**—the same code works for both players by parameterizing the index. The `init()` function implements the **reset pattern**, restoring the system to its initial state. Template literals for dynamic element selection (`current--${activePlayer}`) show **string interpolation** for computed property access.

📌 Game State Management

Store all game state in variables:

```
let scores, currentScore, activePlayer, playing;

const init = function () {
  scores = [0, 0];          // Array: both players' total scores
  currentScore = 0;          // Current round score
  activePlayer = 0;          // 0 or 1
  playing = true;            // Game active flag
};
```

Why Arrays for Scores?

```
// ✅ Can use activePlayer as index
scores[activePlayer] += currentScore;

// ❌ Without array - needs conditionals
if (activePlayer === 0) score0 += currentScore;
else score1 += currentScore;
```

📌 Initialization / Reset Pattern

The `init()` function resets everything to starting state:

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

  // Reset UI
  score0El.textContent = 0;
  score1El.textContent = 0;
  current0El.textContent = 0;
  current1El.textContent = 0;

  // Reset classes
  diceEl.classList.add('hidden');
  player0El.classList.remove('player--winner');
  player1El.classList.remove('player--winner');
  player0El.classList.add('player--active');
  player1El.classList.remove('player--active');
};

// Call on load
init();

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

📌 Dynamic Element Selection

Use template literals to select elements dynamically:

```
// Instead of:
if (activePlayer === 0) {
  current0El.textContent = currentScore;
} else {
  current1El.textContent = currentScore;
}

// Use:
document.getElementById(`current--${activePlayer}`).textContent =
currentScore;
```

This works because element IDs follow a pattern: `current--0`, `current--1`, `score--0`, `score--1`.

📌 Switching Players

Toggle between players using:

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

Ternary for Binary Toggle

```
activePlayer = activePlayer === 0 ? 1 : 0;

// Equivalent to:
if (activePlayer === 0) {
  activePlayer = 1;
} else {
  activePlayer = 0;
}
```

📌 classList.toggle()

Adds class if missing, removes if present:

```
// Both players toggle simultaneously
player0El.classList.toggle('player--active');
player1El.classList.toggle('player--active');
```

Initial State	After Toggle
P0: active, P1: inactive	P0: inactive, P1: active
P0: inactive, P1: active	P0: active, P1: inactive

📌 Guard Conditions with `playing`

Prevent actions when game is over:

```
btnRoll.addEventListener('click', function () {
  if (playing) {
    // Game logic only runs if game is active
  }
});

btnHold.addEventListener('click', function () {
  if (playing) {
    // ...
  }
});
```

When a player wins:

```
if (scores[activePlayer] >= 100) {
  playing = false;  // Disables all game actions
  // ...
}
```

📌 Random Dice Roll

Generate random integer 1-6:

```
const dice = Math.trunc(Math.random() * 6) + 1;
```

Step	Expression	Range
1	`Math.random()`	0 to < 1
2	`* 6`	0 to < 6
3	`Math.trunc()`	0, 1, 2, 3, 4, 5
4	`+ 1`	1, 2, 3, 4, 5, 6

📌 Dynamic Image Source

Change dice image based on roll:

```
diceEl.src = `dice-${dice}.png`;

// dice = 1 → "dice-1.png"
// dice = 2 → "dice-2.png"
// etc.
```

📌 Game Flow Summary

```
┌──────────────────────────────────────────────────────┐
│                     ROLL DICE                        │
```

```
├─────────────────────────────────────────────────────┤
│  if playing:                                        │
│    1. Generate random 1–6                           │
│    2. Show dice image                               │
│    3. If dice ≠ 1: add to currentScore              │
│       If dice = 1: switchPlayer()                   │
└─────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────┐
│                     HOLD                            │
├─────────────────────────────────────────────────────┤
│  if playing:                                        │
│    1. Add currentScore to scores[activePlayer]      │
│    2. If score ≥ 100: WINNER! (playing = false)     │
│       Else: switchPlayer()                          │
└─────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────┐
│                   NEW GAME                          │
├─────────────────────────────────────────────────────┤
│  Call init() → Reset all state and UI               │
└─────────────────────────────────────────────────────┘
```