20250923-LearningGuide

Week 1: Review Guide

Learning Guide with Code Annotations and Breadcrumbs

PUZZLE 1: Basic Number Guessing (Variables & Functions)

What Students Should Discover:

Key Concept 1: Variables Store Information

```
let secretNumber = 7;        // 🎯 The computer's chosen number
let attempts = 0;           // 📊 Counter that tracks tries
let maxAttempts = 3;        // 🛑 Limit on guesses
```

Key Concept 2: Functions Are Mini-Programs

```
function playGame() {
    // This function coordinates everything
    let playerGuess = document.getElementById("userInput").value;  // INPUT
    attempts = attempts + 1;                                        // PROCESS
    let result = checkGuess(playerGuess);                          // PROCESS
    showMessage(result);                                           // OUTPUT
}
```

Key Concept 3: Document Methods (Introduced with Comments)

```
// getElementById("userInput") - finds the input box by its ID name
// .value - gets what the player typed inside it
// .innerText - changes the text inside an element
```

🍞 Breadcrumbs to Drop for Puzzle 1

Breadcrumb 1 (5 minutes):

> *"Look at the very top of the JavaScript file. What do you see being set up there? Those `let` statements are like labeling boxes before you put things in them."*

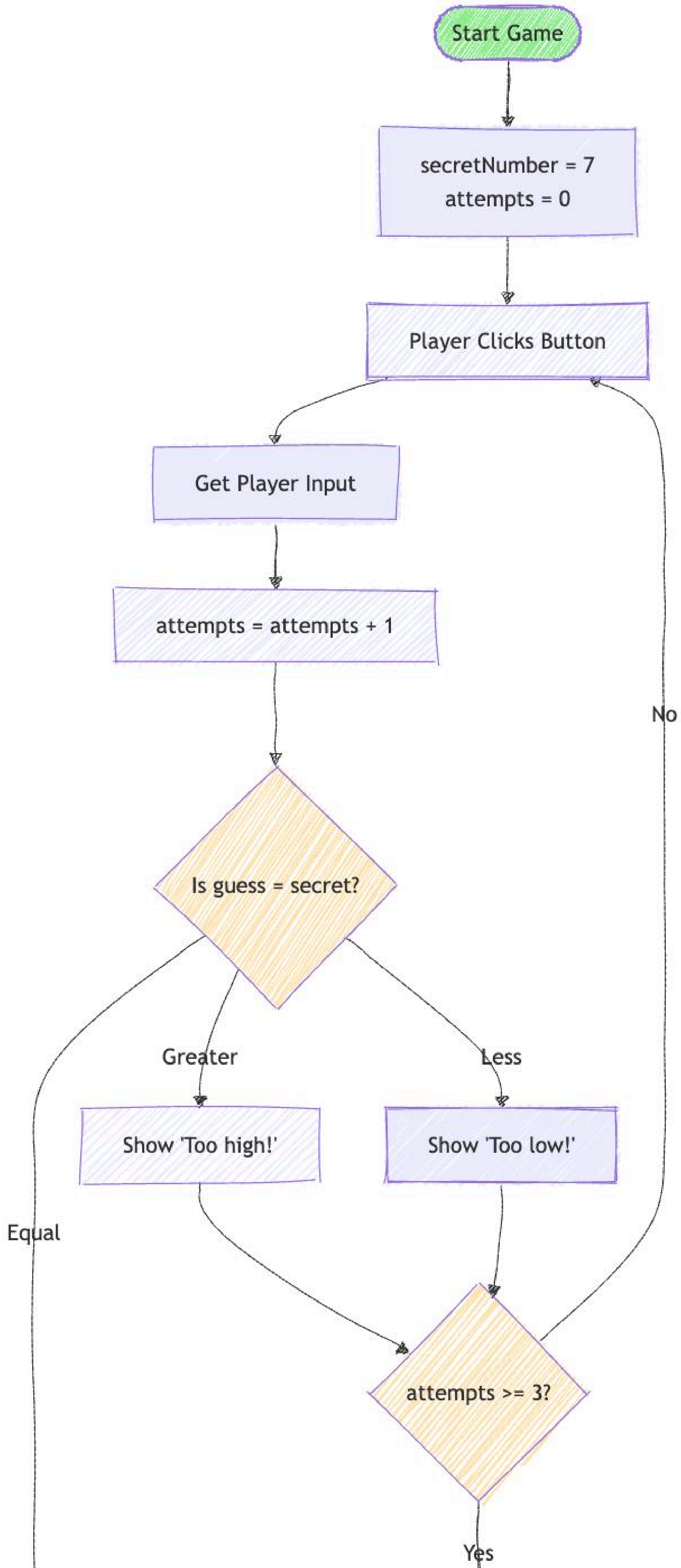
Breadcrumb 2 (10 minutes):

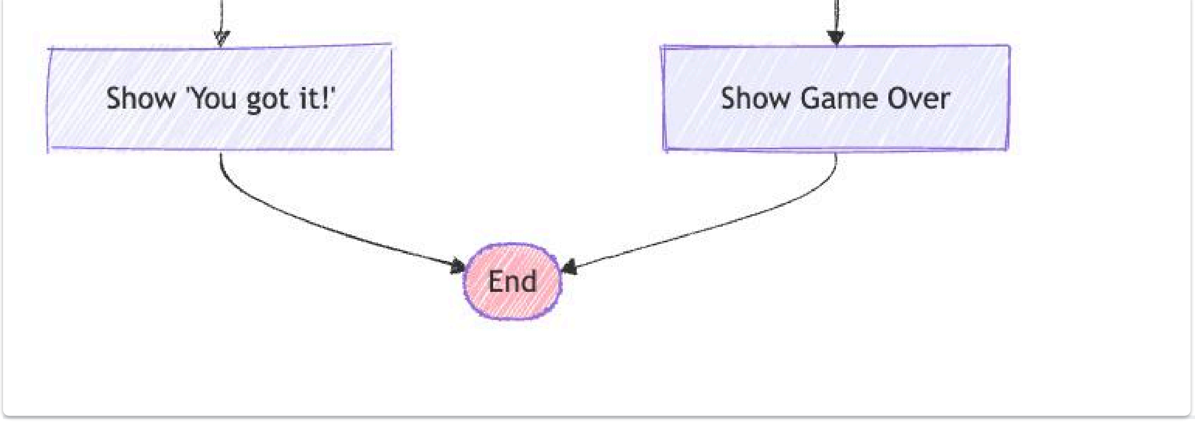
> *"Find where it says `onclick='playGame()'` in the HTML. That's the starting point - everything begins when someone clicks. Now follow what `playGame` does, step by step."*

Breadcrumb 3 (15 minutes):

> *"The function `checkGuess` has three `if` statements. What question is each one asking? Think of them like a series of doors - which door opens depends on your guess."*

Expected Student Flowchart:


Start Game
secretNumber = 7
attempts = 0
Player Clicks Button
Get Player Input
attempts = attempts + 1
Is guess = secret?
Greater
Less
Equal
Show 'Too high!'
Show 'Too low!'
attempts >= 3?
No
Yes



PUZZLE 2: Sequential Chain Pattern

What Students Should Discover:

Key Concept 4: State Management (Simplified)

```
// The game now remembers multiple things
let score = 0;              // 🏆 Accumulates across rounds
let currentRound = 1;       // 📍 Tracks progress
let attempts = 0;           // 🔄 Resets each round
```

Key Concept 5: Function Chains

```
// Each function calls the next like dominoes
function startChain() {
    captureGuess();  // Calls next function
}

function captureGuess() {
    // Do something...
    checkGuess(Number(guess));  // Calls next function
}
```

🍞 Breadcrumbs to Drop for Puzzle 2

Breadcrumb 1 (5 minutes):

> *"This game has rounds! Look for `currentRound` - how does it change? When does it go from 1 to 2?"*

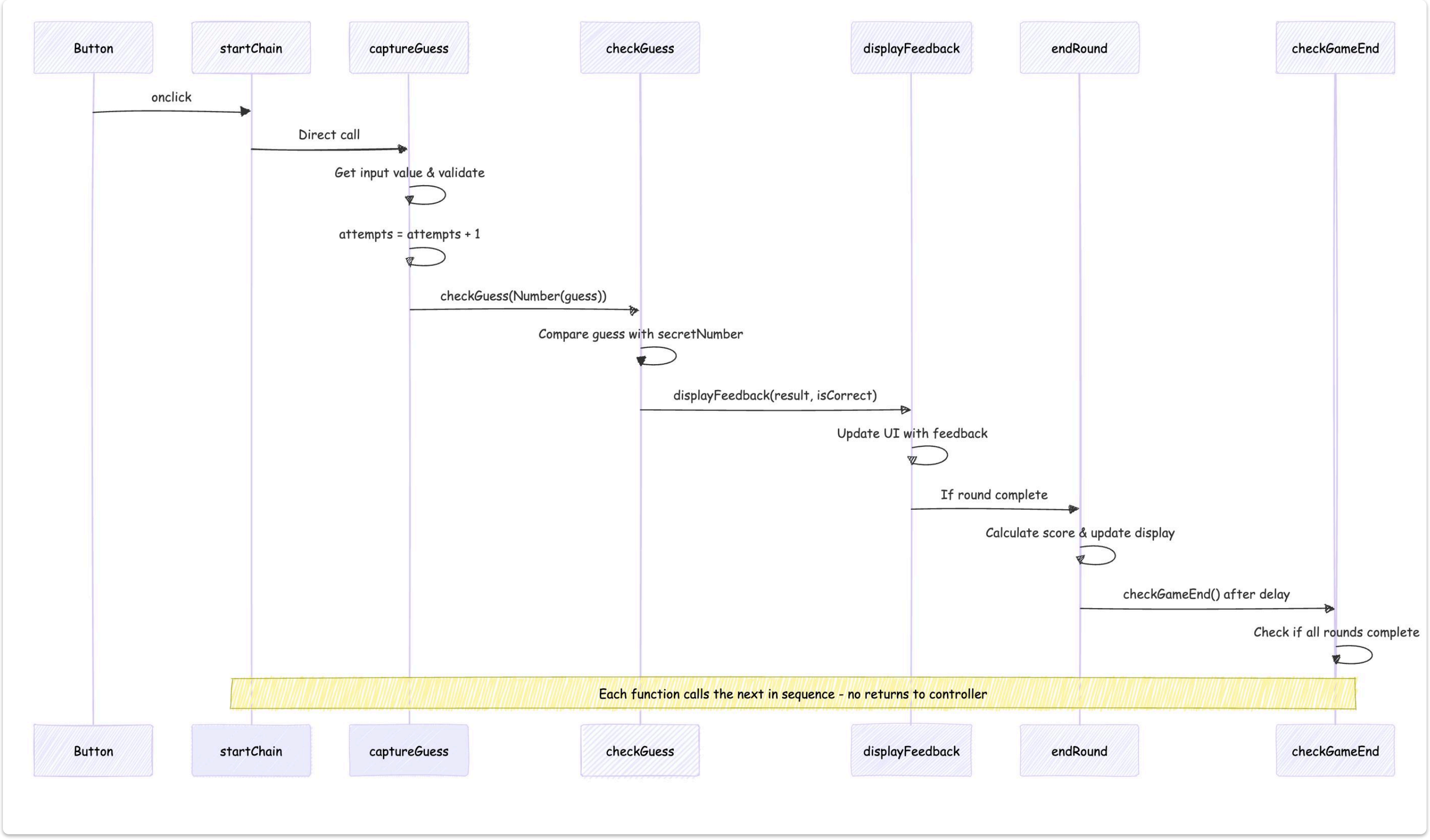
Breadcrumb 2 (10 minutes):

> *"Notice the function names: `startChain`, then what? It's like a relay race - each runner passes the baton. Follow the baton!"*

Breadcrumb 3 (15 minutes):

> *"The scoring formula is `(4 - attempts) * 10`. Work it out: if you guess in 1 try, what's your score? What about 3 tries? See the pattern?"*

Function Chain Sequence:


Button
startChain
captureGuess
checkGuess
displayFeedback
endRound
checkGameEnd
onclick
Direct call
Get input value & validate
attempts = attempts + 1
checkGuess(Number(guess))
Compare guess with secretNumber
displayFeedback(result, isCorrect)
Update UI with feedback
If round complete
Calculate score & update display
checkGameEnd() after delay
Check if all rounds complete
Each function calls the next in sequence - no returns to controller
Button
startChain
captureGuess
checkGuess
displayFeedback
endRound
checkGameEnd

PUZZLE 3: Orchestrator Pattern with Multiple Buttons

What Students Should Discover:

Key Concept 6: Central Controller

```
// One function manages everything
function gameController(action) {
    if (action === 'init') {
        initializeGame();
    } else if (action === 'guess') {
        processGuess();
    }
    // etc...
}
```

Key Concept 7: HTML/JavaScript Connection

```
<!-- All buttons visible all the time -->
<button onclick="gameController('init')">Start Game</button>
<button onclick="gameController('guess')">Submit Guess</button>
<button onclick="gameController('nextRound')">Next Round</button>
<button onclick="gameController('endGame')">End Game</button>
```

Key Concept 8: Defensive Programming with Game State

```
// Track if a round is active
let gameActive = false;

function processGuess() {
    // Check if a round is active
    if (!gameActive) {
        setMessage("resultText", "Start a new game or go to next round!");
        return;  // Exit early - defensive programming!
    }
    // ... rest of function
}
```

🍞 Breadcrumbs to Drop for Puzzle 3

Breadcrumb 1 (5 minutes):

> *"Look at the HTML - count how many buttons you see. Can you click any button at any time? Try it! What happens?"*

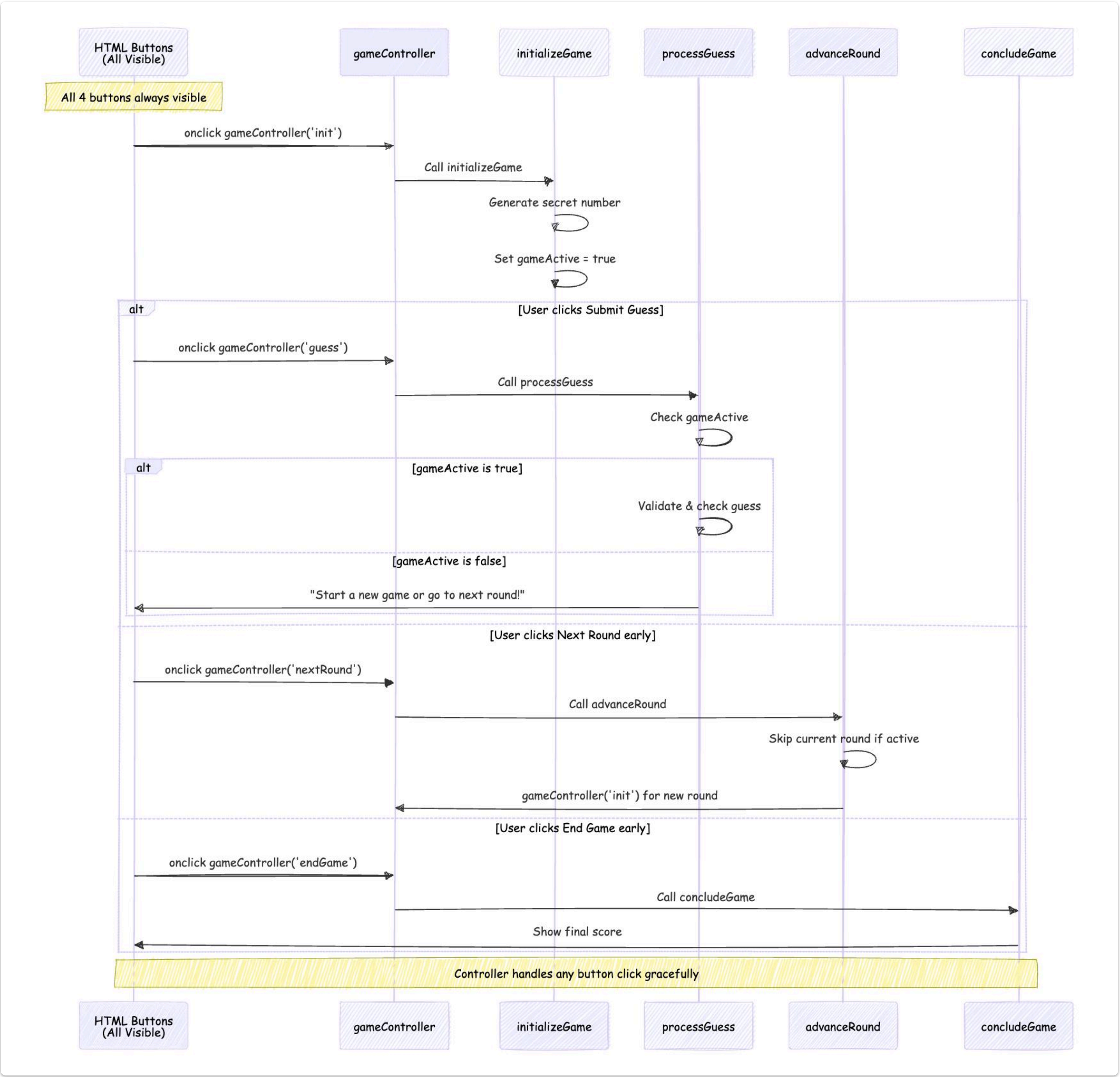
Breadcrumb 2 (10 minutes):

> *"Find the variable `gameActive` - it's like a traffic light. When is it green (true)? When is it red (false)? How does this protect the game?"*

Breadcrumb 3 (15 minutes):

> *"Try clicking buttons in the 'wrong' order. The game doesn't break! Look for `return` statements - they're like exit doors that stop a function early."*

Orchestrator Flow with State Protection:



HTML Buttons (All Visible)
gameController
initializeGame
processGuess
advanceRound
concludeGame
All 4 buttons always visible
onclick gameController('init')
Call initializeGame
Generate secret number
Set gameActive = true
alt
[User clicks Submit Guess]
onclick gameController('guess')
Call processGuess
Check gameActive
alt
[gameActive is true]
Validate & check guess
[gameActive is false]
"Start a new game or go to next round!"
[User clicks Next Round early]
onclick gameController('nextRound')
Call advanceRound
Skip current round if active
gameController('init') for new round
[User clicks End Game early]
onclick gameController('endGame')
Call concludeGame
Show final score
Controller handles any button click gracefully

Common Misconceptions to Address

For All Puzzles:

- "Functions run automatically" → They must be called
- "Order doesn't matter" → Trace execution line by line

- "Variables are global" → Some reset, some persist

Puzzle 2 Specific:

- "Arrays are needed" → We can count without storing
- "Chain can skip steps" → Each MUST call the next

Puzzle 3 Specific (NEW):

- "All buttons should work anytime" → Game state controls validity
- "Button text matters for function" → Only `onclick` matters
- "Clicking wrong button breaks game" → Defensive programming prevents this

---

Assessment Rubric

Can Identify (Basic - 1 point each):

- [ ] Variables and their values
- [ ] Function names and purpose
- [ ] The three document methods
- [ ] Pattern differences between puzzles
- [ ] NEW: Which button calls which controller action
- [ ] NEW: How `gameActive` protects from invalid actions

Can Explain (Intermediate - 2 points each):

- [ ] How chain pattern works
- [ ] How orchestrator pattern works
- [ ] Why Math.random() doesn't need importing
- [ ] How state persists between rounds
- [ ] NEW: What happens with "out of order" button clicks
- [ ] NEW: Difference between button text and onclick behavior

Can Predict (Advanced - 3 points each):

- [ ] What happens if you break the chain
- [ ] How to add a new controller action
- [ ] Which pattern suits which problem
- [ ] NEW: What happens if you remove the `gameActive` check
- [ ] NEW: How to add a fifth button with new action

Tuesday Discussion Prompts

After exploration:

1. **Pattern Recognition:**

   > *"What patterns appeared in all three puzzles?"*

2. **Architecture Choice:**

   > *"Chain or orchestrator - which felt more natural? Why?"*

3. **HTML/JS Connection:**

   > *"How do the buttons 'know' what to do when clicked?"*

4. **State Protection:**

   > *"What happens when you click buttons in the 'wrong' order? How does the game handle this?"*

5. Transfer to RPS:

   > *"For rock-paper-scissors, what buttons would you need? Should they all be visible all the time?"*

Progressive Diagram Introduction

Day 1 (Puzzle Work):

1. Puzzle 1: Basic flowchart (decisions and flow)
2. Puzzle 2: Sequence diagram (function calls)
3. Puzzle 3: State diagram with UI elements

Day 2 (Discussion):

- Compare all three diagrams
- Show how same logic can be visualized differently
- Discuss how HTML/CSS/JS work together
- Let students choose their preferred style

HTML/CSS/JS Teaching Points

The Three Layers:

1. HTML: Structure (buttons, text, input) - always visible
2. CSS: Appearance (colors, layout) - no dynamic hiding
3. JavaScript: Behavior (what happens on click) + state protection

Key Discoveries:

- Button text is just visual - `onclick` is behavior
- All UI elements stay visible - simplicity!
- JavaScript protects game state with `gameActive` flag
- Multiple buttons can call same function with different parameters
- Early `return` statements act as guards

Defensive Programming Introduction:

- State flags: `gameActive` tracks if actions are valid
- Early returns: Exit function if conditions aren't met
- User feedback: Tell user why their action didn't work
- Graceful handling: Game never "breaks" from wrong clicks

Homework Assessment Guide

Rock-Paper-Scissors diagrams should show:

- Variables for choices/scores
- Clear game flow
- Either chain OR orchestrator pattern
- Understanding of WHY they chose that pattern
- NEW: Which HTML buttons they'd need
- NEW: How to handle invalid button clicks

Good: Shows flow, uses week's concepts
Better: Includes all buttons and state protection
Excellent: Explains pattern choice, shows understanding of defensive programming

---

Extension Ideas

For students who finish early:

- Add a fifth button that resets the game to round 1
- Remove the `gameActive` check and see what breaks
- Add more descriptive error messages for each invalid action
- Create a "hint" button that shows if guess should be higher/lower
- Add a button that shows/hides the game rules

Remember: The goal is understanding patterns AND defensive programming!