# 7b Theory JS API pages 114 - 120 Creating A Hangman Game Part II

Nicholas Bruzzese

July 27, 2025

## 1 Lecture: Understanding Loops in JavaScript

Loops are a fundamental concept in programming that allow us to repeat a block of code multiple times. In JavaScript, loops are particularly useful for tasks like processing lists, repeating actions until a condition is met, or iterating through data. This lecture will explain the two types of loops found in the provided text: the `for` loop and the `while` loop. We will explore how they work, their structure, and their practical use in a Hangman game, keeping the explanation simple and clear.

### 1.1 The `for` Loop

A `for` loop is used when you know how many times you want to repeat a block of code. It's like telling the computer, "Do this task a specific number of times." In the context of the Hangman game, the `for` loop is used to set up an array and to check a player's guess against a word.

The structure of a `for` loop looks like this:

```
  for (var i = 0; i < word.length; i++) {
    // Code to repeat
  }

```

Let's break down the three parts inside the parentheses:

1. **Initialisation** (`var i = 0`): This creates a variable, often called `i`, and sets it to 0. Think of `i` as a counter that keeps track of how many times the loop has run.
2. **Condition** (`i < word.length`): This is the rule that decides whether the loop should keep going. For example, if `word` is "pancake" (7 letters), the loop will run as long as `i` is less than 7.
3. **Update** (`i++`): After each loop, `i` increases by 1 (this is what `i++` means). This ensures the loop doesn't run forever.

In the Hangman game, a `for` loop is used to create an array called `answerArray`. This array starts with underscores (_) to represent each letter in the secret word. For example, if the word is "monkey" (6 letters), the loop runs 6 times, adding an underscore each time:

```
var answerArray = [];
for (var i = 0; i < word.length; i++) {
  answerArray[i] = '_';
}
```

When this loop finishes, `answerArray` looks like `['_', '_', '_', '_', '_', '_']` for "monkey". The loop uses `i` to set each position in the array, ensuring it matches the length of the word.

Another use of the `for` loop in the game is to check the player's guess. If the player guesses a letter, like "a", the loop goes through each letter in the word to see if it matches:

```
for (var j = 0; j < word.length; j++) {
  if (word[j] == guess) {
    answerArray[j] = guess;
    remainingLetters--;
  }
}
```

Here, the loop uses a variable `j` (to avoid confusion with the `i` used earlier). For a word like "pancake", it checks each letter (`p`, `a`, `n`, `c`, `a`, `k`, `e`) against the guess "a". If there's a match (like at positions where `a` appears), it updates `answerArray` with the letter and reduces `remainingLetters` by 1.

## 1.2 The `while` Loop

A `while` loop is used when you don't know exactly how many times you need to repeat something, but you want to keep going until a condition changes. In the Hangman game, the `while` loop is used to keep the game running as long as there are letters left to guess.

The structure of a `while` loop is simpler:

```
while (remainingLetters > 0) {
  // Code to repeat
}
```

The loop keeps running as long as the condition (`remainingLetters > 0`) is true. In the game, `remainingLetters` starts as the number of letters in the secret word (e.g., 7 for "pancake"). Each time the player guesses a correct letter, `remainingLetters` decreases. When it reaches 0, the loop stops because all letters have been guessed.

The `while` loop in the Hangman game contains several steps:

1. **Show Progress**: It displays the current state of `answerArray` (e.g., `_ a _ _ a _ _` for "pancake" after guessing "a").
2. **Get Input**: It asks the player for a guess using a prompt.
3. **Check Input**: It ensures the guess is valid (a single letter) or stops the game if the player cancels.

4. **Update Game State**: If the guess is valid, it updates `answerArray` and `remainingLetters` using the `for` loop described earlier.

Here's the `while` loop from the game:

```
  while (remainingLetters > 0) {
    alert(answerArray.join(' '));
    var guess = prompt('Guess a letter, or click Cancel to stop playing.');
    if (guess == null) {
      break;
    } else if (guess.length != 1) {
      alert('Please enter a single letter.');
    } else {
      for (var j = 0; j < word.length; j++) {
        if (word[j] == guess) {
          answerArray[j] = guess;
          remainingLetters--;
        }
      }
    }
  }

```

## 1.3 The `break` Statement

Sometimes, you need to stop a loop early, even if the condition is still true. This is where the `break` statement comes in. In the Hangman game, if the player clicks "Cancel" on the prompt, the guess becomes `null`. The code checks for this:

```
  if (guess == null) {
    break;
  }

```

The `break` statement immediately stops the `while` loop, ending the game. This is useful because it lets the player quit at any time, no matter how many letters are left to guess.

## 1.4 Practical Example: How Loops Work Together

In the Hangman game, loops work together to create a smooth experience. The `for` loop sets up the initial `answerArray` with underscores, matching the length of the secret word. The `while` loop runs the game, repeatedly asking for guesses and showing progress. Inside the `while` loop, another `for` loop checks each guess against the word, updating the game state. The `break` statement provides a way to exit early if the player cancels.

For example, if the word is "pancake":

- The first `for` loop creates `answerArray` as `['_', '_', '_', '_', '_', '_', '_']`.
- The `while` loop shows this array and asks for a guess.

- If the player guesses "a", the inner `for` loop checks each letter, updating `answerArray` to `['_', 'a', '_', '_', 'a', '_', '_']` and reducing `remainingLetters` by 2 (because "a" appears twice).
- The `while` loop continues until `remainingLetters` is 0, then shows the final word and congratulates the player.

## 1.5 Conclusion

Loops in JavaScript, like `for` and `while`, are powerful tools for repeating tasks. The `for` loop is great for a known number of repetitions, such as setting up an array or checking each letter in a word. The `while` loop is ideal for repeating until a condition changes, like continuing a game until all letters are guessed. The `break` statement adds flexibility by allowing early exits from loops. By combining these loops, as seen in the Hangman game, you can create interactive programs that respond to user input and manage complex tasks step by step. Understanding how to use loops effectively is a key skill for building dynamic JavaScript applications.