Month 1, Week 4: The Unabridged Learning Guide

JavaScript - The Language Core (Part 2)

Introduction: From Logic to Structure Welcome, architects. In Week 3, we forged the foundational tools of logic: variables to store information and conditionals to make decisions. We learned how to create a single path of execution. This week, we elevate our craft. We will learn to create systems—to manage repetition, to build reusable components, and to structure complex data. This is the week we move from writing simple scripts to architecting the foundations of real applications.

What We Will Cover This Week This document is your definitive guide for the week. It is designed to be read, reviewed, and referenced. It complements the in-class slides by providing a deeper, more textual exploration of each concept.

Module	Topic	Core concepts
0	**Revisiting the Foundation**	Truthy/Falsy values in practice, the `switch` statement's mechanics (including fall-through), and the power of Logical Short-Circuiting.
1	**Automating Repetition**	The `for` loop (detailed execution trace), the `while` loop, the `do...while` loop, `break` and `continue` keywords for fine-grained control.
2	**The Art of Reusability: Functions**	Function Declarations vs. Expressions (Hoisting), Arrow Functions, Parameters vs. Arguments, the `return` keyword, the concept of Pure Functions.
3	**Architecting Data**	Arrays (Indexes, properties, essential methods: `push`, `pop`, `shift`, `unshift`), Objects (Key-Value pairs, dot vs. bracket notation, modifying data).
4	**Building an Application**	A comprehensive take-home assignment to synthesize all learned concepts into a functional To-Do List Manager.

Module 0: Revisiting the Foundation - A Deeper Look at Conditionals A deep understanding of how a program makes decisions is non-negotiable. Let's solidify

this foundation.

The Nuance of Truthy & Falsy We learned the six falsy values: `false`, `0`, `""`, `null`, `undefined`, and `NaN`. Everything else is truthy. But why does this exist? It allows for more expressive and concise code that often reads closer to plain English.

Verbose Check	Pragmatic (Truthy/Falsy) Check	Reads As...
`if (itemsInCart !== 0)`	`if (itemsInCart)`	"If we have items in the cart..."
`if (username !== '')`	`if (username)`	"If a username exists..."
`if (error !== null)`	`if (error)`	"If there is an error..."

This pattern is a hallmark of professional JavaScript code. Master it.

The `switch` Statement: A Closer Look The `switch` statement is a specialized conditional for comparing one value against a list of variants.

The Crucial Role of `break` and "Fall-Through" The `break` keyword is essential. When a `case` is matched, the code executes until it hits a `break` statement, at which point it exits the `switch` block entirely. If you omit the `break`, something interesting called "fall-through" happens: the code will continue executing the code in the next `case` block, regardless of whether its value matches.

```
const userRole = 'editor';

switch (userRole) {
  case 'admin':
    console.log('Full access granted.');
    // No break!
  case 'editor':
    console.log('Content writing access granted.'); // This will run
    break; // Exits here
  case 'viewer':
    console.log('View-only access granted.');
    break;
}
// Output: "Content writing access granted."
```

While sometimes used intentionally, accidental fall-through is a common source of bugs. **Always include a `break` unless you have a specific reason for fall-through.**

Module 1: Automating Repetition - Loops Loops are the solution to the universal programming problem: "How do I do something over and over again?"

The `for` Loop: A Step-by-Step Execution Trace Let's dissect this loop with surgical precision:

```
for (let i = 0; i < 3; i++) {
    console.log(i);
}
```

1. Initialization (`let i = 0`): This happens one time only, before anything else. A variable `i` is created and set to `0`. `i` is now `0`.
2. Condition Check (`i < 3`): Is `0 < 3`? Yes, it's `true`. So, the loop body will execute.
3. Loop Body Executes: `console.log(i)` prints `0`.
4. Final Expression (`i++`): The value of `i` is incremented. `i` is now `1`.
5. Condition Check (`i < 3`): Is `1 < 3`? Yes, it's `true`. The body will execute again.
6. Loop Body Executes: `console.log(i)` prints `1`.
7. Final Expression (`i++`): The value of `i` is incremented. `i` is now `2`.
8. Condition Check (`i < 3`): Is `2 < 3`? Yes, it's `true`. The body will execute again.
9. Loop Body Executes: `console.log(i)` prints `2`.
10. Final Expression (`i++`): The value of `i` is incremented. `i` is now `3`.
11. Condition Check (`i < 3`): Is `3 < 3`? No, it's `false`. The loop terminates.

The `while` loop: Ask Now, Act Later A `while` loop in JavaScript is a control flow statement that repeatedly executes a block of code as long as a specified condition evaluates to `true`.

The loop checks the condition before executing the code block. If the condition is `true`, the code runs. This cycle continues until the condition becomes `false`, at which point the loop stops.

Syntax The basic syntax for a `while` loop is:

```
while (condition) {
  // code to be executed as long as the condition is true
}
```

- `condition`: An expression that is evaluated before each pass through the loop. If it evaluates to `true`, the code block is executed. If it evaluates to `false`, the loop terminates.

How It Works

1. The `condition` is checked.
2. If the condition is `true`:

- The code inside the `{}` block is executed.
- The process goes back to step 1.

3. If the condition is `false`:

- The loop is skipped, and the program continues with the code that follows the loop.

Code Examples Example 1: A Simple Counter

This is a classic example that counts from 1 to 5 and prints each number to the console.

```
let count = 1; // 1. Initialize the counter

while (count <= 5) { // 2. Set the condition
  console.log("The count is: " + count);
  count++; // 3. IMPORTANT: Increment the counter
}

console.log("Loop finished!");
```

Explanation:

1. We start with a variable `count` set to `1`.
2. The `while` loop checks if `count <= 5`. Since `1` is less than or equal to `5`, the condition is `true`, and the loop body executes.
3. Inside the loop, it prints the current value of `count` and then increments it by one (`count++`).
4. The loop checks the condition again. Now `count` is `2`. `2 <= 5` is still `true`, so the loop runs again.
5. This process continues until `count` becomes `6`. At that point, the condition `6 <= 5` is `false`, the loop terminates, and "`Loop finished!`" is printed.

Warning: Infinite Loops

If you forget to include a way for the condition to become `false` (like `count++` in the example above), you will create an infinite loop, which will crash your program or browser tab.

```
// DANGER: This is an infinite loop!
let i = 0;
while (i < 10) {
  console.log("This will run forever!");
  // 'i' is never incremented, so 'i < 10' is always true.
}
```

Example 2: Processing an Array

A `while` loop is useful when you want to modify an array until it's empty, as the number of iterations depends on the array's changing size.

```
let tasks = ["Wash dishes", "Do laundry", "Walk the dog"];

while (tasks.length > 0) {
  let currentTask = tasks.shift(); // .shift() removes and returns the first item
  console.log("Processing task: " + currentTask);
}

console.log("All tasks completed!");
console.log("Tasks left: " + tasks.length); // Outputs: Tasks left: 0
```

Explanation:

1. The loop continues as long as `tasks.length > 0` (the array is not empty).
2. Inside the loop, `tasks.shift()` removes the first element from the array.
3. This removed element is processed (in this case, printed to the console).
4. Each iteration shortens the array, so eventually `tasks.length` becomes `0`, the condition is `false`, and the loop stops.

The `do...while` Loop: Act First, Ask Later A lesser-known cousin of the `while` loop is the `do...while` loop. Its unique feature is that the loop body is **guaranteed to execute at least once**, because the condition is checked after the first iteration.

```
let userResponse;
do {
  userResponse = prompt("Please enter 'yes' to continue:"); // This will always run o
} while (userResponse !== 'yes');

console.log("You entered 'yes'!");
```

This is useful for scenarios like prompting a user for input until it's valid.

Module 2: The Art of Reusability - Functions Functions allow us to package logic into reusable, named blocks. They are the single most important tool for organizing complex programs.

Function Declarations vs. Function Expressions There are two primary ways to create a function. The difference is subtle but important.

- **Function Declaration:**

  ```
  // Hoisted: Can be called before it's defined in the code.
  console.log(add(5, 10)); // 15

  function add(a, b) {
      return a + b;
  }
  ```

  Declarations are "hoisted" to the top of their scope by the JavaScript interpreter, meaning you can call them before they appear in your code.

- Function Expression: (Assigning an anonymous function to a variable)

  ```
  // Not Hoisted: Must be defined before it is called.
  // console.log(subtract(10, 5)); // This would cause a ReferenceError!

  const subtract = function(a, b) {
      return a - b;
  };
  ```

Modern best practice, especially with `const`, is to use function expressions (or arrow functions) to prevent issues related to hoisting and to enforce a more linear, top-to-bottom code readability.

The Concept of Pure Functions A "pure function" is a function that, given the same input, **will always return the same output** and has **no side effects**. A side effect is any interaction with the outside world from within the function (e.g., changing a global variable, writing to the console, making a network request).

- Pure Function:

```
const calculatePrice = (base, taxRate) => base * (1 + taxRate);
```

- Impure Function (has a side effect):

```
let totalSales = 0;
function addToTotal(amount) {
    totalSales += amount; // Side effect: modifies a variable outside itself.
    return totalSales;
}
```

Striving to write pure functions makes your code more predictable, easier to test, and less prone to bugs.

Module 3: Architecting Data - Complex Structures

Arrays: A Deeper Look An array is more than just a list; it's a powerful object with built-in properties and methods.

- `length` **Property:** Not a method! It's a property that tells you how many elements are in the array.

```
const team = ['Alex', 'Jane', 'Peter'];
console.log(team.length); // 3
```

- **Essential Methods (Actions that modify the array):**
  - **`push(item)`:** Adds an item to the **end**.
  - **`pop()`:** Removes (and returns) the item from the **end**.
  - **`unshift(item)`:** Adds an item to the **beginning**.
  - **`shift()`:** Removes (and returns) the item from the **beginning**.

Objects: The Building Blocks of Models Objects allow us to model real-world things by grouping related data.

Adding, Modifying, and Deleting Properties:

Once an object is created, it is mutable. You can change its contents.

```
const car = {
  make: 'Honda',
  model: 'Civic'
};

// 1. Adding a new property
car.year = 2022;
car['color'] = 'blue';
```

```
// 2. Modifying an existing property
car.model = 'Accord';

// 3. Deleting a property
delete car.make;

console.log(car); // { model: 'Accord', year: 2022, color: 'blue' }
```

Using Variables as Keys (Bracket Notation's Superpower):

Bracket notation allows you to use the value of a variable to access a property. This is impossible with dot notation and is essential for dynamic programming.

```
const propertyToAccess = 'model';
const myCar = { make: 'Toyota', model: 'Camry' };

console.log(myCar[propertyToAccess]); // 'Camry'
// console.log(myCar.propertyToAccess); // undefined, because it looks for a key na
```

Take-Home Assignment: The To-Do List Manager Objective: To build a complete mini-application using arrays, objects, functions, and loops. This is a significant step towards building real applications.

1. **Setup:** Create a file `todo.js`.
2. **The Data Structure:** Your to-do list will be stored in an array of objects. Each object will represent a single task.

   ```
   const todoList = [
       { id: 1, task: 'Buy groceries', completed: false },
       { id: 2, task: 'Finish Week 4 assignment', completed: true }
   ];

   let nextId = 3;
   ```

3. **The Functions:**

- **`displayTasks()`:** A function that takes no arguments. It should loop through the `todoList` array and print each task to the console, indicating whether it's complete or not (e.g., `"[x] Buy groceries"` or `"[ ] Clean room"`).
- **`addTask(taskText)`:** A function that takes a string as an argument. It should create a new task object (with a unique `id`, the provided `taskText`, and `completed: false`), add it to the `todoList` array, and increment the `nextId` counter.
- **`markTaskComplete(taskId)`:** A function that takes a number (an `id`) as an argument. It should find the task object in the array with the matching `id` and change its `completed` status to `true`.

4. **The Simulation:** At the bottom of your file, write code to simulate using your application:

   ```
   console.log("--- Initial To-Do List ---");
   displayTasks();
   ```

```
console.log("\n--- Adding New Tasks ---");
addTask("Clean the kitchen");
addTask("Read a chapter of a book");
displayTasks();

console.log("\n--- Completing a Task ---");
markTaskComplete(1); // Mark 'Buy groceries' as complete
displayTasks();
```

5. **Bonus Challenge:** Create a function `deleteTask(taskId)` that removes a task object from the array based on its `id`. This is more complex and will require you to research array methods like `findIndex()` and `splice()`.
6. **Submission:** Submit your `todo.js` file via a Pull Request to your personal assignments repository, following the professional branching workflow we learned in Week 2.