7-Debugging and Testing

Chapter 7: Debugging and Testing

Debugging and testing are essential skills for every programmer. They help identify and fix errors and ensure your code works as expected.

7.1 Understanding Errors

Errors are a natural part of programming. In JavaScript, errors generally fall into three categories:

1. **Syntax Errors:** Occur when your code violates the syntax rules of the language.
   - Example:

```
console.log("Missing closing parenthesis;
```

2. **Runtime Errors:** Occur during code execution, often caused by invalid operations.
   - Example:

```
let num = 5;
console.log(num.toUpperCase()); // Attempting to call a non-existent method
```

3. **Logical Errors:** The program runs without crashing, but the output isn't as expected.
   - Example:

```
function add(a, b) {
  return a - b; // Logical error: subtraction instead of addition
}
```

7.2 Debugging with the Browser Console

Modern browsers like Chrome and Firefox come with built-in developer tools that make debugging easier.

1. **Opening the Console:**
   - Press `Ctrl+Shift+I` (Windows/Linux) or `Cmd+Option+I` (Mac) to open DevTools.
   - Navigate to the **Console** tab.
2. **Using `console.log`:**
   - Output values to track program execution.

```
let x = 10;
console.log("Value of x:", x); // Output: "Value of x: 10"
```

3. **Using Breakpoints:**
   - Go to the **Sources** tab in DevTools.
   - Find your script file, click on the line number to set a breakpoint.
   - The code execution will pause at the breakpoint, allowing you to inspect variables.
4. **Inspecting Variables:**
   - Use the **Watch** panel to monitor specific variables.
   - Hover over variables in the code to view their current values.

7.3 Common Debugging Techniques

1. **Isolate the Problem:**
   - Narrow down the code that could be causing the issue by commenting out sections or running them independently.
2. **Check for Typos:**
   - Pay close attention to variable names, method calls, and syntax.
3. **Trace Execution Flow:**
   - Use `console.log` to trace the order in which functions are called and how variables change.
4. **Use `try...catch`:**
   - Catch runtime errors gracefully.

```
try {
  let result = someUndefinedFunction();
} catch (error) {
  console.error("An error occurred:", error.message);
}
```

7.4 Writing Test Cases

Testing ensures your code behaves as expected. JavaScript has various testing frameworks, but for this chapter, let's cover some core concepts.

7.4.1 Manual Testing

Manually verify the functionality by running your code with different inputs and checking outputs.

Example:

```
function multiply(a, b) {
  return a * b;
}

console.log(multiply(2, 3)); // Expected output: 6
console.log(multiply(0, 10)); // Expected output: 0
```

7.4.2 Unit Testing

Unit tests check individual functions or components in isolation. Tools like Jest or Mocha are commonly used, but here's how you can manually write a simple unit test:

Example:

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

// Test Cases
function testAdd() {
  console.assert(add(2, 3) === 5, "Test Case 1 Failed");
  console.assert(add(-1, 1) === 0, "Test Case 2 Failed");
  console.assert(add(0, 0) === 0, "Test Case 3 Failed");
}

testAdd(); // Runs the tests
```

If a test fails, `console.assert` outputs an error message.

7.4.3 Edge Cases

Test your functions with edge cases to ensure they handle unexpected inputs.

Example:

```
function divide(a, b) {
  if (b === 0) {
    return "Cannot divide by zero";
  }
  return a / b;
}

console.log(divide(10, 2)); // Expected: 5
console.log(divide(10, 0)); // Expected: "Cannot divide by zero"
```

7.4.4 Automated Testing

If you're ready to explore testing libraries, here's a quick overview:

1. **Jest:**
   - Install: `npm install jest --save-dev`
   - Write Tests:

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

     test("adds 1 + 2 to equal 3", () => {
       expect(sum(1, 2)).toBe(3);
     });
     ```

2. **Mocha and Chai:**
   - Install: `npm install mocha chai --save-dev`
   - Write Tests:

     ```
     const { expect } = require("chai");

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

     describe("subtract", () => {
       it("should return 5 when subtracting 10 - 5", () => {
     ```

```
      expect(subtract(10, 5)).to.equal(5);
    });
  });
```

7.5 Practical Debugging Example

Let's put debugging and testing into practice with a real-world example.

Example: Finding the Average of Numbers

```
function calculateAverage(numbers) {
  if (!Array.isArray(numbers) || numbers.length === 0) {
    throw new Error("Input must be a non-empty array");
  }

  let sum = 0;

  numbers.forEach((num) => {
    if (typeof num !== "number") {
      throw new Error("Array must contain only numbers");
    }
    sum += num;
  });

  return sum / numbers.length;
}

// Test Cases
try {
  console.log(calculateAverage([10, 20, 30])); // Expected: 20
  console.log(calculateAverage([])); // Error: Input must be a non-empty
array
} catch (error) {
  console.error("Test failed:", error.message);
}

// Debugging: Add console.log to trace the issue
```

7.6 Debugging Tools and Resources

Here are some tools and resources to improve your debugging skills:

1. **Debugger Keyword:**
   - Use `debugger` to pause execution at a specific point in your code.

```
function debugExample() {
  let x = 10;
  debugger; // Opens the browser debugger at this line
  console.log(x);
}

debugExample();
```

2. **Linting Tools:**
   - ESLint helps identify and fix common errors before running the code.

```
npm install eslint --save-dev
```

3. **Online Debugging:**
   - Platforms like JSFiddle and CodePen let you test JavaScript code online.