Debugging Your Web Projects

Methods for Troubleshooting HTML, CSS, and JavaScript

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Definition and Purpose of Debugging

Debugging is the systematic process of identifying, analyzing, and eliminating errors (commonly referred to as "bugs") in software. In web development, this involves ensuring that HTML, CSS, and JavaScript code performs as expected and meets the intended design and behavior of a web application.

The purpose of debugging is to:

- **Identify Errors**: Detect coding issues or deviations from expected behavior in web applications.
- **Understand Causes**: Trace the root cause of an error to specific lines of code, logical flaws, or misconfigured settings.
- **Correct Issues**: Apply fixes to ensure that code functions correctly, consistently, and efficiently.

Debugging is a fundamental practice for every developer because it ensures that applications provide reliable and consistent user experiences, adhere to security standards, and are optimized for performance.

Common Types of Errors

Syntax Errors

These occur when code is not written following proper syntax rules of the language. For example, missing closing tags in HTML, unclosed brackets in JavaScript, or incorrect property names in CSS. Syntax errors are often caught by linters or compilers but can still appear in runtime.

```
function greet() {
    console.log("Hello, World!");
}
```

This snippet will throw a syntax error because of the missing closing parenthesis.

Runtime Errors

These errors occur while the program is running. They often stem from operations that cannot be executed successfully, such as trying to call a function on an undefined variable or accessing an element that does not exist.

```
let button = document.getElementById("nonExistentButton");
button.addEventListener("click", () => {
    console.log("Button clicked!");
});
```

If the button element does not exist in the HTML, a runtime error will occur when attempting to add an event listener.

Logical Errors

These are errors in the logic of the code that cause unexpected or incorrect behavior. Logical errors don't necessarily produce error messages but result in incorrect output or broken features.

```
function isEven(number) {
    return number % 2 === 0 ? false : true;
}
```

This function incorrectly returns false for even numbers and true for odd numbers due to a logical flaw.

Overview of Browser Developer Tools

Browser developer tools are powerful debugging utilities that come built into modern web browsers. These tools enable developers to:

- **Inspect HTML and CSS**: Check the structure and style of the web page to identify layout issues, broken styles, or incorrect attributes.
- **Debug JavaScript**: Set breakpoints, view the call stack, and inspect variable values to understand errors or unexpected behavior in scripts.
- **Monitor Network Requests**: Analyze network traffic between the browser and server to identify failed requests, missing resources, or performance bottlenecks.
- **Profile Page Performance**: Analyze page loading and runtime performance, such as rendering delays or inefficient JavaScript execution.
- **Test Accessibility**: Ensure the page meets accessibility standards

Getting Started with Developer Tools

Browser developer tools provide a suite of features to inspect, debug, and optimize your web applications. Accessing these tools is straightforward but varies slightly between browsers.

Accessing Developer Tools in Different Browsers

Google Chrome

- **Right-click Inspect**: Right-click on any element of a web page and select "Inspect" from the context menu. This will open the developer tools with the "Elements" panel active, highlighting the selected element in the HTML structure.
- **Keyboard Shortcut**: Press F12 or Ctrl + Shift + I (Windows/Linux), or Cmd + Option + I (Mac).
- **Menu Access**: Click the three-dot menu at the top-right corner of the browser, navigate to "More tools," and select "Developer tools."

Mozilla Firefox

- **Right-click Inspect**: Right-click any element on the page and choose "Inspect" from the context menu to open the developer tools with the "Inspector" panel active.
- **Keyboard Shortcut**: Press Ctrl + Shift + I (Windows/Linux) or Cmd + Option + I (Mac).
- **Menu Access**: Click the menu button (three horizontal lines) in the top-right corner of the browser, go to "Web Developer," and select "Toggle Tools."

Microsoft Edge

- **Right-click Inspect**: Right-click on any element and select "Inspect" to open the developer tools at the "Elements" panel.
- **Keyboard Shortcut**: Press F12 or Ctrl + Shift + I (Windows/Linux), or Cmd + Option + I (Mac).
- **Menu Access**: Click the three-dot menu in the top-right corner, go to "More tools," and choose "Developer tools."

Safari

- **Enable Web Inspector**: If the developer tools do not appear:
  - Open Safari's "Preferences" by pressing Cmd + ,.
  - Go to the "Advanced" tab and check the option "Show Develop menu in menu bar."
- **Access Developer Tools**:
  - Right-click on any page element and choose "Inspect Element."
  - Alternatively, press Cmd + Option + I to toggle the Web Inspector.

Overview of the Interface and Primary Panels

Elements (or Inspector) Panel

Allows you to inspect and modify the HTML and CSS of the current page.

Key Features:

- **HTML Tree**: Displays the live HTML structure of the page, often represented as a collapsible tree.
- **Styles Pane**: Shows the CSS rules applied to the selected element, organized by source. You can directly modify or add new styles to see immediate changes.
- **Computed Styles**: Lists all the final computed CSS styles applied to the selected element, giving insights into how styles cascade.
- **Box Model**: Visualizes the selected element's layout (padding, borders, margins).

Console Panel

Provides a live JavaScript environment for logging messages, evaluating expressions, and viewing error messages.

Key Features:

- **Logging Output**: Displays errors, warnings, and log messages from JavaScript running on the page.
- **Interactive Console**: Allows you to execute JavaScript code interactively. This can be useful for testing functions, manipulating DOM elements, and checking variable values.
- **Error Stack Traces**: Shows detailed stack traces for runtime errors, helping to trace the source of issues.

Network Panel

Analyzes all network requests made by the page, useful for monitoring resource loading and debugging AJAX calls.

Key Features:

- **Request List**: Shows all HTTP requests made by the page, including their status, size, type, and timing.
- **Headers and Data**: Displays detailed information for each request, such as request/response headers, cookies, and response data.
- **Waterfall View**: Visualizes the timeline of each request to identify performance bottlenecks.

Sources Panel

A JavaScript debugging tool that allows you to examine and step through your code.

Key Features:

- **File Explorer**: Lets you navigate through the project's file structure and view source files.
- **Breakpoints**: Set breakpoints to pause execution at specific lines to examine variable values and execution flow.
- **Call Stack**: Shows the sequence of function calls leading to the current point, helping trace the execution path.
- **Scope and Watch Variables**: Inspect local and global variables and add custom watch expressions.

Performance Panel

Profiles page performance to help identify slow-loading resources or inefficient code.

Key Features:

- **Recording**: Captures the loading and runtime performance of the page.
- **Flame Chart**: Visualizes CPU usage and function calls across time.
- **Frames and Events**: Shows the frame rate and how events or functions impact rendering.

Application Panel

Provides tools to inspect browser storage (cookies, local storage, session storage), service workers, and other application-level components.

Key Features:

- **Storage Inspection**: View and edit data in cookies, local storage, and session storage.
- **Service Workers**: Monitor registered service workers and their cache status.
- **Manifest and Web App Features**: Review the web app manifest and debug progressive web app features.

Configuring Developer Tools Layout

Docking Developer Tools

Most browsers allow you to "dock" the developer tools to various locations:

- **Bottom Dock**: The tools are displayed at the bottom of the browser window, giving you a full-width view while still displaying the webpage above.
- **Right Dock**: The tools appear to the right of the browser window, offering a taller but narrower view of the developer tools.
- **Left Dock**: Some browsers allow docking on the left side of the window, helpful if your webpage layout fits better in this configuration.
- **Undocked/Separate Window**: Developer tools can be displayed in their own window, which is beneficial for working with multiple screens.

Changing Dock Position:

- **Chrome and Edge**: Click the "Customize and Control" (three vertical dots) menu in the developer tools window, then hover over the "Dock side" option and choose your preferred position.

- **Firefox**: Click the "Dock to right" or "Dock to bottom" icon at the top right of the developer tools window to change the position.
- **Safari**: Click the "Dock at Bottom" or "Separate Window" button at the top left of the developer tools window.

Rearranging Panels and Tabs

Developer tools consist of multiple panels or tabs that focus on different aspects like inspecting HTML, debugging JavaScript, or analyzing network requests. Most browsers offer the ability to rearrange or customize these panels:

- **Changing Panel Order**: You can often drag tabs to reorder them, ensuring that frequently used tools are quickly accessible.
- **Adding/Removing Tabs**:
  - **In Chrome/Edge**: Right-click on the tab bar and select "Customize" to choose which tabs are displayed.
  - **In Firefox**: Use the settings menu (gear icon) to toggle panels on or off.
  - **In Safari**: All panels are visible by default, but you can switch between them by clicking the appropriate tab.

Personalization Options

- **Dark or Light Themes**: Most developer tools offer different color schemes to suit your working environment. You can usually change this setting under the "Preferences" or "Settings" option within the developer tools menu.
- **Custom Shortcuts**: Some tools provide the ability to modify or add custom keyboard shortcuts to streamline your workflow.

Additional Settings:

- **Persistence**: In Chrome and Firefox, options are available to "preserve log" or "disable cache" to maintain state between page reloads.
- **Device Emulation**: Simulate different screen sizes and resolutions to preview how your web page will appear on various devices.

Debugging HTML with Developer Tools

The Elements panel in browser developer tools allows you to inspect, modify, and experiment with the HTML structure of a web page in real-time.

Using the Elements Panel to Inspect and Modify the HTML Structure

Accessing the Elements Panel:

- Open your browser's developer tools (typically by right-clicking on the page and selecting "Inspect," or using keyboard shortcuts like Ctrl + Shift + I or Cmd + Option + I).
- Navigate to the "Elements" tab or panel, which shows the HTML structure of the currently loaded page.

Understanding the Elements Panel Layout:

- **DOM Tree**: The left pane displays the HTML DOM tree, representing the structure of the web page.
- **Styles Pane**: The right pane shows the styles applied to the selected element and allows you to modify them directly.
- **Breadcrumbs**: At the bottom of the panel, you will find a breadcrumbs trail, showing the hierarchy of the selected element.

Inspecting Page Elements:

- **Highlight Elements**: Hover over any node in the DOM tree or on the webpage itself to highlight the corresponding element visually.
- **Select Elements**: Click on any element in the DOM tree or the page to display its properties, attributes, and styles.

Modifying HTML Elements Directly

Edit HTML Nodes:

- Right-click on a node in the DOM tree and choose "Edit as HTML" to change the inner HTML of the selected element.
- Alternatively, double-click on the tag name to edit attributes like `class`, `id`, or `src`.

Add/Remove Elements:

- To add a new element, right-click on an existing node and choose "Edit HTML" or "Add Element."
- You can also drag and drop nodes within the DOM tree to reposition them.

Delete Elements:

- Right-click an element in the DOM tree and select "Delete element" to remove it.

Identifying Missing or Misplaced Elements

Check for Missing Elements:

- If you suspect a missing element, ensure that the necessary tag is present in the HTML.
- Verify that any dynamically added content (through JavaScript) is being inserted correctly.

Finding Misplaced Elements:

- Look for logical inconsistencies in the DOM tree, such as misplaced tags or elements within the wrong parent node.
- Use the breadcrumbs trail to verify the parent-child relationship of the selected element.
- If misplaced, use drag-and-drop or cut and paste to move nodes within the DOM tree.

Modifying HTML Attributes and Experimenting with Changes Live

Change Attributes:

- Double-click on any attribute name or value in the DOM tree to edit it directly.
- Right-click an element and select "Attributes" to add new attributes.

Live Experimentation:

- Modify or add attributes like `class`, `id`, `href`, or `src` to see immediate changes on the web page.
- Experiment with modifying content by directly editing text or replacing images.

Reverting Changes:

- Remember that modifications in the Elements panel are temporary and will be lost upon reloading the page.
- To keep these changes, copy the modified HTML and paste it into your source code or use the browser's "Local Overrides" feature.

Debugging CSS with Developer Tools

Debugging CSS involves understanding how styles are applied to web page elements and diagnosing layout issues. The developer tools provide the necessary features to inspect, analyze, and modify CSS styles in real-time.

Inspecting CSS Properties Using the Styles Panel

Accessing the Styles Panel:

- Open the developer tools and select the "Elements" tab or equivalent (e.g., "Inspector" in Firefox).
- The left side shows the DOM tree, while the right side shows the Styles panel with the applied CSS rules.

Inspecting Applied Styles:

- Click on an element in the DOM tree or on the web page to see its styles in the Styles panel.
- The panel shows the styles applied to the selected element, including inherited styles and those overridden by more specific rules.

Understanding the Cascade and Specificity Issues

Cascade and Inheritance:

- **Cascade**: The Styles panel lists the styles in order of precedence. Specific rules (e.g., inline styles) are shown above more general ones (e.g., external stylesheets).
- **Inheritance**: Some properties (like color and font-size) are inherited by child elements from their parent. The Styles panel clearly marks inherited styles.

Specificity Calculation:

- The Styles panel displays the origin of each rule and its specificity.
- Rules are applied based on specificity, calculated using a combination of tag, class, and ID selectors:
  - Inline styles have the highest specificity.
  - ID selectors are more specific than class selectors.
  - Class selectors are more specific than tag selectors.

Resolving Specificity Issues:

- If a property is overridden due to specificity, it is marked as "struck out."
- To resolve specificity issues:
  - Increase the specificity of your rule by adding more qualifiers.
  - Avoid using the `!important` directive excessively, which can lead to maintainability problems.

Detecting Layout Problems: Box Model Issues, Overflow, and Alignment

Box Model Overview:

- The Styles panel contains a "Box Model" section that visually displays the margin, border, padding, and content dimensions of the selected element.
- **Inspect Element Size**: Hover over each section to highlight it on the web page.

Common Layout Problems:

- **Box Model Issues**:
  - Incorrect use of `box-sizing` can result in unexpected content sizes.
  - Margins can collapse or stack unexpectedly, especially with nested or adjacent elements.
- **Overflow**:
  - Elements may overflow if the parent container is too small.
  - Check for properties like `overflow` or `white-space` that might be clipping content.
- **Alignment**:
  - Misalignment often occurs due to mixed units (percentages vs. pixels).
  - Check the layout using Flexbox or Grid tools, if available.

Debugging Layout Issues:

- **Highlight Problematic Areas**: Hover over elements in the DOM tree to visualize their boundaries.
- **Toggle Styles**: Temporarily disable CSS properties affecting layout, like margins or floats, to isolate problematic rules.
- **Experiment**: Modify the box model properties live to fix spacing or overflow problems.

Using Computed Styles

Accessing the Computed Panel:

- In the developer tools, select the "Elements" tab and choose an element by clicking on it in the DOM tree or directly on the web page.
- Find the "Computed" panel/tab on the right. It shows the final, calculated values of all CSS properties that apply to the selected element.

Understanding Computed Values:

- Computed values are the final values of CSS properties after all styles have been cascaded, inherited, and applied according to browser rules.

- This panel can be particularly helpful for properties where the computed value differs from the declared value (e.g., relative units, percentages, or default values).
- For each computed property, the panel shows:
  - The property name and its final value.
  - The source styles or rules that contribute to this value.

Troubleshooting with Computed Styles:

- **Identifying Unexpected Values**: If a computed property doesn't reflect what you expected, inspect which rules are affecting it. Use the arrow to expand the property details and see the originating styles.
- **Examining Box Model Dimensions**: Properties like width, height, padding, and margin are useful for resolving layout issues by showing exact pixel dimensions.
- **Checking Inheritance and Defaults**: Verify if a computed style is inherited or has a browser default value.

Debugging CSS Grid Layouts

Inspecting Grid Elements:

- Select an element with a CSS Grid layout in the DOM tree or on the web page itself to focus on it.
- In the "Styles" panel, look for properties such as `display: grid` or `display: inline-grid` to confirm that Grid is applied.
- Toggle the "Grid" badge (or similar indicator) next to the `display` property to highlight the grid lines overlaid on the web page.

Visualizing Grid Lines and Tracks:

- **Grid Overlay**: Once enabled, the Grid overlay shows the boundaries of grid tracks (rows and columns), making it easier to identify alignment or sizing issues.
- **Grid Gaps**: Gaps between grid items (defined using `grid-gap` or similar properties) are visible and clearly marked.
- **Named Areas**: If your grid layout uses named areas, the overlay will display the names within each grid cell.

Analyzing Grid Layout Issues:

- Hover over any grid item in the DOM tree or web page to highlight it visually.
- Verify if grid items are placed in the expected rows and columns and check alignment within the grid.
- In the "Layout" panel (or equivalent), adjust the grid properties live to see how changes affect the overall structure.

Customizing Grid Debugging:

- Developer tools may allow customization of the grid overlay, such as changing the grid line colors or labels to improve visibility.
- Some tools also offer specific grid debugging features, like isolating individual grid items for focused analysis.

Debugging Flexbox Layouts

Inspecting Flexbox Containers:

- Select an element with a Flexbox layout by finding properties like `display: flex` or `display: inline-flex` in the Styles panel.
- Look for the "Flex" badge (or similar indicator) next to the `display` property to confirm the presence of a flex container.

Visualizing Flexbox Layout:

- **Flexbox Overlay**: Enable the Flexbox overlay, which visually displays the direction of the main axis, along with the alignment and order of flex items.
- **Alignment Information**: The overlay indicates where flex items are aligned along the main and cross axes.

Analyzing Flexbox Layout Issues:

- Check for issues like overflowing content, unexpected item ordering, or incorrect alignment.
- Use the Styles panel to adjust flex properties like `flex-direction`, `justify-content`, or `align-items` live to see immediate changes.

Flexbox Debugging Tools:

- In the "Layout" panel (or similar), interactively modify the flex container properties.
- Experiment with the "Align" tool to visualize how different alignment options affect item positioning.

Experimenting with Live Changes

Modifying Properties Directly:

- In the Styles panel, directly edit or add new properties to see their impact instantly.
- Toggle checkboxes to enable/disable specific styles without removing them.
- Use browser autocomplete to suggest valid CSS property values.

Inline Styles via the Element Inspector:

- Use the `element.style` section in the Styles panel to add inline styles to any element.

- Changes take effect immediately, providing quick feedback.

Try New CSS Rules:

- Click "+ Add Rule" or right-click on an element to add a new CSS selector with styles.
- Experiment with new selectors to see how they override or affect existing rules.

Persisting Changes:

- Remember that live changes are temporary and lost upon page refresh. Copy any desired modifications and apply them to your source files.
- Some browsers, like Chrome, offer "Local Overrides" to persist changes across page reloads by storing modified files locally.

Debugging JavaScript with Developer Tools

Debugging JavaScript requires a clear understanding of errors and how they impact your application's behavior. The Console panel in browser developer tools is the primary place where errors and messages are logged, providing a starting point for investigating and fixing issues.

Using the Console to Identify Syntax and Runtime Errors

The Console is a command-line interface that displays errors, warnings, and general log messages, and allows you to interact directly with JavaScript.

Understanding Error Types:

- **Syntax Errors**: Occur when code violates JavaScript syntax rules, making the code unexecutable. The Console shows a red error message indicating the file name, line number, and a brief description of the syntax error.

```
Uncaught SyntaxError: Unexpected token ')' in main.js:5
```

- **Runtime Errors**: Occur when the script is executed but encounters an issue, such as calling an undefined function or accessing a null value. The Console displays these as red errors with a description and a stack trace.

```
Uncaught TypeError: undefined is not a function in app.js:10
```

Locating Errors in Your Code:

- Each error message typically includes the file name and line number where the error occurred. Click on the link in the Console message to navigate directly to the offending line in the "Sources" panel.

- The stack trace (the chain of function calls leading up to the error) helps identify how the code reached this problematic state.

Inspecting Variable Values:

- **Log Values**: Use `console.log()` in your code to print variable values to the Console and understand their state.

```
let count = 0;
console.log("Count value:", count);
```

- **Interactively Inspect Values**: Enter expressions directly into the Console to evaluate them immediately. For instance, typing `myVar` will display the current value of `myVar` if it's in the current scope.

Filtering Console Messages:

- Use the filter options to focus on specific message types (errors, warnings, logs) or text.
- Clear the Console regularly using the trash icon to remove old messages and keep the panel manageable.

Further Tips:

- **Break Down Errors**: If multiple errors occur at once, focus on fixing the first error to see if subsequent issues resolve.
- **Warnings**: Pay attention to warnings, as they often highlight practices that could lead to errors.
- **Preserve Log**: Enable "Preserve log" to retain Console messages across page reloads, making it easier to track down errors that occur during initialization.

Debugging with Breakpoints

Breakpoints allow you to pause the execution of JavaScript at specific lines of code or under certain conditions, enabling you to inspect the state of your application at that point. By stepping through the code, you can identify issues with control flow, variable states, or other logical errors.

Conditional, Event Listener, and DOM Breakpoints

- **Setting Standard Breakpoints**:
  - In the "Sources" panel, navigate to the JavaScript file you want to inspect.
  - Click on the line number where you want to pause execution. The breakpoint will appear as a blue marker.
  - Once execution reaches that line, it will pause, and you can inspect variables and the call stack.
- **Conditional Breakpoints**:

  - A conditional breakpoint pauses execution only if a specified condition is met.
  - Right-click on a line number in the "Sources" panel and choose "Add conditional breakpoint."
  - Enter a JavaScript expression that evaluates to true or false. The execution will pause only if the expression is true.
  - **Example Condition**:

    ```
    counter > 10
    ```

    This example will pause execution only when the value of `counter` exceeds 10.

- **Event Listener Breakpoints**:
  - Event listener breakpoints let you pause execution when a specific event type is triggered (e.g., click, input change, or form submission).
  - In the "Sources" panel, find the "Event Listener Breakpoints" section (usually in the right sidebar).
  - Expand the event category (e.g., "Mouse"), then check the desired event type (e.g., "click").
  - The execution will pause whenever the specified event is triggered, allowing you to inspect the state at that moment.
- **DOM Breakpoints**:
  - DOM breakpoints allow you to pause execution when an element's properties are modified.
  - Select an element in the DOM tree and right-click to access "Break on..."
  - Choose one of the following options:
    * **Subtree modifications**: Pauses execution when child elements of the selected node are added or removed.
    * **Attribute modifications**: Pauses execution when attributes of the selected element are changed.
    * **Node removal**: Pauses execution when the selected element is removed from the DOM.

Stepping Through Code to Understand Control Flow

Stepping Controls:

- **Resume**: Continue running the script until the next breakpoint or the end of execution.
- **Step Over**: Execute the current line and move to the next line in the same function. If the current line is a function call, the function is executed without stepping into it.
- **Step Into**: Move into the first function call on the current line, allowing you to inspect its execution line by line.

- **Step Out**: Complete the execution of the current function and return to the calling function.
- **Deactivate Breakpoints**: Temporarily disable all breakpoints while keeping them in place for future debugging.

Inspecting State and Control Flow:

- **Variables**: Use the "Scope" panel to inspect variable values in the current scope, including local, global, and closure variables.
- **Call Stack**: Review the function call hierarchy leading to the current execution point to understand how the code reached this state.
- **Watch Expressions**: Add custom expressions to the "Watch" panel to monitor variable changes across different execution points.

Analyzing Function Calls and Inspecting Values

Understanding the sequence of function calls and the values of variables across different execution points is crucial for debugging complex JavaScript applications. The "Call Stack," "Watch," and "Scope" panels provide the necessary insights to analyze control flow and identify where things might be going wrong.

Analyzing Function Calls with the Call Stack

- **Overview of the Call Stack**:
  - The Call Stack is a visual representation of the sequence of functions called up to the current execution point.
  - Each entry on the Call Stack represents a function call and the associated line of code that triggered the call.
  - The most recent function is at the top, while the initial call is at the bottom.
- **Using the Call Stack Panel**:
  - In the "Sources" panel, locate the "Call Stack" section on the right sidebar.
  - When execution is paused at a breakpoint, the Call Stack lists all active function calls.
  - Click on any function in the stack to jump to the corresponding line of code and inspect the variable values or control flow.
- **Diagnosing Issues with the Call Stack**:
  - **Trace Execution Flow**: Identify how the execution reached a particular function and trace back the originating events.
  - **Check Unexpected Calls**: Ensure that function calls occur in the expected order. If a function is called unintentionally, inspect preceding calls for logical errors.

  - **Error Tracking**: For runtime errors, the stack trace helps trace back to the origin of the problem.

Inspecting Values with the Watch and Scope Panels

- **Scope Panel**:
  - The Scope panel displays variables and their values within the current execution context.
  - It categorizes variables into different scopes: local, global, block, and closure.
  - **Inspect Local Variables**: Check local variables to confirm their state within the current function.
  - **Closure Variables**: Variables from outer functions available due to closures are also shown.
  - **Global Scope**: See all global variables, including those declared on the `window` object.
- **Watch Panel**:
  - The Watch panel allows you to monitor custom expressions and variable values across different execution points.
  - **Add Expressions**:
    * Click the "+" button or right-click inside the Watch panel to add an expression.
    * Type in any valid JavaScript expression, such as a variable name or an arithmetic operation.
  - **Monitor Changes**:
    * The Watch panel updates the values of expressions every time the execution pauses.
    * Track variables across different functions and breakpoints to verify how and when their values change.

Using These Tools Together

- **Identify Logical Errors**: Combine the insights from the Call Stack and Scope panels to ensure functions are executed correctly and variables hold the expected values at each step.
- **Validate Program Logic**: Use the Watch panel to monitor key variables and expressions, validating assumptions about the code's state at different points.