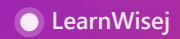


CSS, CssClass, CssStyle, States, and Runtime Theme Changes

Lesson Guide · Lab / Exam Guide



Lesson Guide

Adding targeted CSS safely with `CssClass` and a scoped application stylesheet, keeping `CssStyle` for one-off dynamic values, using `AppearanceKey` and custom states for semantic variants, and switching themes at startup or for one session at runtime. What this means for a Wisej.NET developer — and how to read this module.



Production focus

Adding targeted CSS safely with `CssClass` and a scoped application stylesheet, keeping `CssStyle` for one-off dynamic values, using `AppearanceKey` and custom states for semantic variants, and switching themes at startup or for one session at runtime.

1. The production problem

The `AdaptiveOps` theme from Module 2 now owns the identity of every standard control in the Adaptive Operations Console, and the requests that arrive next are all small: a metric card needs a rounded, elevated surface, an SLA badge needs to be compact, one progress bar needs a width computed from live data, the Assign command should look like the primary action, and the night shift wants a dark variant. Each request has a tempting shortcut — a `CssStyle` string here, a `BackColor` there, the shared theme edited in place — and each shortcut quietly moves a visual decision to a layer that does not own it. Six months later nobody can say what the theme renders, what the CSS overrides and what the code patches. This module puts each of those five requests in the mechanism that owns it.

2. Key concepts

Concept	Why it matters
CssClass	One or more class names added to the widget's main element in the browser: the reusable hook a scoped application stylesheet can target without touching the theme.
CssStyle	A CSS string rendered directly as the style attribute of the widget's main element; right for a single computed value, wrong as a general styling tool because inline styles are hard to inspect, theme or hand to a designer.
scoped application stylesheet	A CSS file loaded alongside the theme whose selectors start from class names you own, such as <code>.metric-card</code> , and never reach into the internal DOM of a standard Wisej widget.
legacy StyleSheet extender	An older component that adds CSS classes to eligible controls and hosts rules in its <code>StyleSheetSource</code> property; a migration aid, unnecessary in new code because every control now has <code>CssClass</code> and <code>CssStyle</code> .
AppearanceKey	The name of an appearance in the active theme that a control renders with, so C# expresses intent (<code>action-button</code>) and the theme owns the colours, fonts and metrics of every state.
custom states	Names added to a control's <code>States</code> collection that the theme can style like hovered or focused, for transient UI conditions such as stale, warning or dirty that a designer should be able to see and restyle.
startup theme vs Application.Theme	The application-wide theme is chosen before the app runs in <code>Web.config</code> or <code>Default.json</code> ; <code>Application.Theme</code> lets one session work with a theme object at runtime without changing what other sessions see.

3. Lesson narrative

Four tools, one rule each

Every Wisej.NET control exposes two client-side hooks. `CssClass` accepts one or more class names that are added to the widget's main element; `CssStyle` accepts a CSS string that is rendered directly as that element's style attribute. They sit next to each other in the property grid and behave very differently in a codebase. A class is a name you can search for, reuse on twenty controls and restyle in one file; an inline style is an anonymous override that wins over most of what the theme generates and stays invisible until someone opens the browser inspector.

The rule is short. Use `CssClass` whenever you want a reusable styling hook backed by an application CSS file. Use `CssStyle` only for very small one-off styles or dynamically computed values: a width that follows a percentage, a colour that arrives with the data. Overuse of `CssStyle` makes the console hard to inspect, hard to theme and hard for a designer to maintain, which is why the lab allows exactly one computed `CssStyle` value and asks you to write down why it could not be a class.

Two older mechanisms still appear in real projects. The `StyleSheet` extender adds CSS classes to eligible controls and hosts rules in its `StyleSheetSource` property; since every modern control already has

`CssClass` and `CssStyle`, it is a migration aid, not something to introduce in new code. The Theme Builder's Stylesheet section ships raw CSS rules inside the theme; it exists for third-party widgets, custom HTML and integration surfaces without a normal Wisej appearance, and on standard widgets the rules generated from appearances and states may override it.

Scoped classes for the console's own UI

The lab adds `AdaptiveOps.css` to the Adaptive Operations Console, a stylesheet the application loads next to the theme. Its job is narrow: style the parts of the console that are yours, not Wisej's. A `.metric-card` class gives the KPI panels their radius, padding and elevation, `.metric-card .metric-title` sets the small uppercase title, and `.compact-badge` turns a label into a pill for SLA and priority markers. Every selector starts from a class you own, so the stylesheet cannot accidentally restyle a button or a grid header elsewhere.

What stays out of that file matters as much as what goes in. Standard control styling — button hover and pressed colours, grid header fonts, the invalid editor border — belongs to the theme appearances from Module 2. Patching them from CSS means brittle selectors that depend on a widget's internal DOM and break silently on the next framework or theme update. Third-party widget CSS follows the same discipline: scoped to the widget's container and documented.

Before a class ships, review what can defeat it. A control with `BackColor`, `ForeColor` or `Font` set directly in the designer overrides both the theme and your class; a leftover `CssStyle` overrides everything. When a class "does not work", the checklist is: is the class on the element, is a property set on the control, is an inline style set, and is the rule scoped to the right container.

Semantic variants: AppearanceKey and custom states

Setting `BackColor`, `ForeColor` and `Font` on the Save button makes it look primary today and untouchable tomorrow. `AppearanceKey` is the alternative: the control renders with a named appearance from the active theme, and the theme owns the visual details for default, hovered, pressed, disabled, focused and invalid. The `action-button` appearance from Module 2 inherits from `button` and overrides only three states; assigning it to the primary command expresses the business intent in C# and leaves the pixels to the theme. The same pattern scales to `destructive-button`, `quiet-panel`, `kpi-card` or `sidebar-button` whenever the business concept is reusable.

Custom states cover the transient conditions an appearance key cannot. Every control has a `States` collection; adding a name such as `stale`, `warning`, `dirty` or `compact` lets the theme style that state exactly as it styles hovered or focused. The lab adds a stale state to the ticket grid rows or the status panel when a ticket has not been updated in time, and the theme decides how stale looks. A state is a UI condition the theme can represent visually, not a place to keep application data, and code should never read a state back to make a decision.

```
// MainPage.cs – each layer owns its own decision
this.cardOpen.CssClass      = "metric-card";      // scoped app CSS
this.lblPriorityBadge.CssClass = "compact-badge";
this.btnSave.AppearanceKey  = "action-button";    // the theme owns the states

// the one computed value: clip-path, not width – the layout engine rewrites
// an inline width on every layout pass and would erase it on the next resize
this.barFill.CssStyle = $"clip-path:inset(0 {100 - slaPercent}% 0 0)";

if (ticket.IsStale)
    row.States.Add("stale");                      // themed custom state
else
    row.States.Remove("stale");
```

Startup themes, session themes and the shared theme

A theme is applied before the application runs by naming it in Web.config or in Default.json. Changing the theme shown in the Visual Studio designer does not change the runtime theme; the console runs with AdaptiveOps only because the configuration says so. Named theme files are the right home for stable branding: versioned, reviewed and identical for every user.

Wisej.NET also exposes `Application.Theme`, through which a session can work with a theme object at runtime. That is where the lab's light/dark switch lives: the toolbar toggle loads the dark variant, assigns it to the current session and remembers the choice in `Application.Session`. The subtle danger is that a theme object can be modified in place. Mutating the theme shared by every session because one user asked for dark mode changes the console for the whole night shift; the safe pattern is a separate named theme file, or a per-session copy, never an edit of the shared object.

The decision rule: named theme files for stable branding, runtime theme changes for personalization, tenant accents, demos and controlled previews. Prove the boundary with two browser sessions side by side, one dark and one light, before you call the deliverable done.



Common pitfalls

`CssStyle` strings copied onto every button until nobody knows what the theme renders. Selectors that reach into a standard widget's internal DOM. `BackColor`, `ForeColor` or `Font` set in the designer silently overriding the theme and the class. Using `States` as a data store. A dark-mode toggle that edits the shared theme for every session.



Visual-system habit

Use the Wisej.NET mechanism that owns the problem: the theme for the identity of standard controls, `AppearanceKey` for semantic variants, `CssClass` for scoped app-specific CSS, `CssStyle` only for one-off dynamic values, layout containers instead of resize code, responsive properties for designer-set profile values, and `ResponsiveProfileChanged` for behaviour that is not simply visual.



Learning objectives

By the end of this module you'll be able to:

- ✓ Use `CssClass`, `CssStyle`, custom CSS, and legacy `StyleSheet` extender patterns correctly.
- ✓ Define scoped classes for app-specific UI while keeping standard control styling in themes.
- ✓ Use `AppearanceKey` and custom states to create semantic control variants.
- ✓ Apply startup themes and understand the difference between session-specific theme changes and global theme changes.

Lab / Exam Guide

What you'll build in the hands-on lab, the suggested approach, and the required deliverables.



Hands-on lab

Add a scoped AdaptiveOps.css stylesheet to the Adaptive Operations Console with classes for metric cards and compact badges, apply them through CssClass, and use CssStyle only for one computed value such as a progress width. Assign the action-button AppearanceKey to the primary command, add a custom stale state to the ticket grid rows or a status panel, and implement a light/dark theme switch in the toolbar that changes the theme for the current session only, without modifying the shared theme.

Starting point

Open the Adaptive Operations Console from Module 2: the docked shell with toolbar, navigation, workspace, details and status panels, the AdaptiveOps theme selected in Web.config or Default.json, its named tokens and the `action-button` appearance. The workspace already holds four metric panels, an SLA progress bar and the ticket grid, but they are still styled with default appearances or ad-hoc `BackColor`, `ForeColor` and `CssStyle` values, and the toolbar has no theme switch.

Suggested approach

- ✓ Create `AdaptiveOps.css`, load it with the application next to the theme, and write only selectors rooted in classes you own: `.metric-card`, `.metric-card .metric-title` and `.compact-badge`. Apply them through `CssClass` to the four metric panels and to the SLA and priority labels.
- ✓ Find the single value that must be computed at runtime, the SLA progress width, set it through `CssStyle` in one method, and write a comment explaining why a class could not express it. Remove every other `CssStyle` and every `BackColor`, `ForeColor` or `Font` override left on those controls.
- ✓ Assign `AppearanceKey = "action-button"` to the primary command (`btnSave`) and confirm that the hovered, pressed and disabled looks come from the theme with no code at all.
- ✓ In the Theme Builder add a `stale` state to the ticket row or status panel appearance, styled from the warning token, then add and remove `"stale"` through `States` when a ticket's last update exceeds the threshold. Keep the threshold in data and the look in the theme.
- ✓ Add a theme toggle to the toolbar that loads a second named theme file, AdaptiveOps-Dark, assigns it to `Application.Theme` for the current session and remembers the choice in `Application.Session`. Verify with two browser sessions that the shared theme is untouched.

Required deliverables

- ✓ Scoped AdaptiveOps.css with metric-card and compact-badge classes applied through CssClass
- ✓ CssStyle used for exactly one computed one-off value, with the reason documented
- ✓ Primary command using the action-button AppearanceKey
- ✓ Custom stale state added through States and styled in the theme

- ✓ Session-only light/dark theme switch that leaves the shared theme untouched

Student review questions

- ✓ A designer asks for a smaller corner radius on every metric card. Which file changes, and how many places would you have had to edit if the radius had been set with `CssStyle` ?
- ✓ Why does the Assign button need no hover or pressed code, and what exactly would you lose by replacing its `AppearanceKey` with `BackColor` and `ForeColor` values that match the palette?
- ✓ Two people are signed in and one of them switches to dark mode. Describe what each sees, and point at the line of your code that guarantees it.



Visual-system habit

Use the Wisej.NET mechanism that owns the problem: the theme for the identity of standard controls, `AppearanceKey` for semantic variants, `CssClass` for scoped app-specific CSS, `CssStyle` only for one-off dynamic values, layout containers instead of resize code, responsive properties for designer-set profile values, and `ResponsiveProfileChanged` for behaviour that is not simply visual.