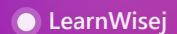


Mastering the Control Paint Event

Lesson Guide · Lab / Exam Guide



Lesson Guide

Server-painted controls done properly: PaintEventArgs.Graphics and ClipRectangle, geometry separated from drawing, disposing every pen, brush and font you create, property setters that call Invalidate() exactly once, and the paint cost budget that decides when a themed control is the better answer. What this means for an experienced .NET developer who already builds Wisej.NET screens and now needs custom graphics — and how to read this module.



Production focus

Server-painted controls done properly: PaintEventArgs.Graphics and ClipRectangle, geometry separated from drawing, disposing every pen, brush and font you create, property setters that call Invalidate() exactly once, and the paint cost budget that decides when a themed control is the better answer.

1. The production problem

Module 1 put one telemetry reading on four surfaces so you could see where each pixel is produced. VisualOperationsStudio now wants that reading in six places, and copying a `Paint` handler into six panels is how a screen ends up with six slightly different gauges, six pens that are never disposed, and a repaint whenever an unrelated field changes. The fix is to stop writing handlers and write a control: one `TelemetryGauge` whose public properties are the only thing that can change its picture, and whose cost you can state in kilobytes.

2. Key concepts

Concept	Why it matters
<code>Wisej.Web.PaintEventArgs</code>	The argument of the <code>Paint</code> event; it hands you a <code>System.Drawing.Graphics</code> you did not create and a rectangle to draw inside, and your code runs on the server.
<code>ClipRectangle</code>	The region Wisej.NET provided for this paint operation; layout can consult the control's client size, but drawing stays bounded by the rectangle you were given.
<code>GaugeGeometry</code> helper	A pure function from a <code>Rectangle</code> plus a normalised value to ring bounds, sweep angles, needle points and a label rectangle, so the maths can be checked without a live control.
<code>Invalidate()</code>	The one call that asks Wisej.NET to re-render the control's surface; a property setter makes it exactly once, and only when the change can alter a pixel.
Resource ownership	The handler disposes every <code>Pen</code> , <code>Brush</code> , <code>Font</code> , <code>GraphicsPath</code> and <code>Bitmap</code> it creates, and never disposes the <code>Graphics</code> the framework supplied.
Paint cost budget	Three numbers per painted control: server CPU spent drawing, server memory for the temporary image, and the bytes the browser must fetch and decode.
<code>AccessibleDescription</code>	Carries the same reading as text, so a value that exists as a drawn arc is still available to a user who never sees the arc.

3. Lesson narrative

The handler, the surface and the rectangle you were given

A Wisej.NET `Paint` handler looks familiar because the drawing object really is `System.Drawing.Graphics`, but the runtime around it is not a desktop. Your handler executes on the server and the picture it produces becomes the background the browser displays. The useful mental model is that you are painting a server-side bitmap, which is also why a repaint carries a network price.

The event supplies `Wisej.Web.PaintEventArgs`, and two members carry everything you need. `Graphics` is the active surface: you did not allocate it and you must not release it. `ClipRectangle` is the region Wisej.NET provided for this operation. Reason about layout with the control's client rectangle, but keep the drawing bounded by the rectangle in your hands.

That makes the first two lines of a production handler deliberately dull: check that the rectangle is large enough to draw anything, then hand it to the geometry helper. A control laid out at zero width, or squeezed during a responsive reflow, simply returns. No paint path is important enough to justify an exception reaching a user's screen.

Shapes, paths, text and transformed content

`Graphics` is a stateful, immediate drawing API, and the families the gauge needs are few. Outlines come from `DrawArc`, `DrawLine` and `DrawPolygon`; filled geometry from `FillPie`, `FillEllipse` and

`FillRectangle` ; text from `DrawString` with a `Font` , a `Brush` and a `StringFormat` ; compound shapes and curves from a `GraphicsPath` in `System.Drawing.Drawing2D` . Clipping and transforms round out the set.

Transforms are where handlers usually go wrong. Applying `TranslateTransform` and `RotateTransform` to swing the needle is easy; undoing them before the caption is drawn is not. Open a graphics container with `BeginContainer()` , draw inside a `try` , and put `EndContainer(state)` in the `finally` -- the managed drawing implementation Wisej.NET ships has no `Graphics.Save / Restore` pair, and the container is its equivalent. The same discipline reappears in Module 4 as the Canvas `Save()` and `Restore()` pair, which does carry those names.

Text deserves the same care: `MeasureString` and `StringFormat` alignment survive a translated caption, a hard-coded offset does not. Images need one extra rule, because `Paint` takes over the background. A watermark is drawn explicitly with `DrawImage` , from an image decoded once and owned by the control, never from a file opened inside the handler.

A reusable control whose properties invalidate it correctly

In the lab, `TelemetryGauge` becomes a real control rather than a handler bolted to a panel. Rendering lives inside the class and consumers set domain properties -- `Value` , `Minimum` , `Maximum` , `WarningThreshold` , `CriticalThreshold` . Wisej.NET also exposes a protected `OnWebPaint(Graphics)` hook on `Control` for library authors, while the public `Paint` event remains the simplest composition mechanism.

Every setter follows one recipe: clamp into a legal range, compare with what is stored, return when nothing changed, assign, then invalidate once. An out-of-range value is clamped rather than rejected, and an inverted range is normalised rather than allowed to produce a negative sweep. After that the drawing code may assume its inputs are sane.

```
public float Value
{
    get => _value;
    set
    {
        var low  = Math.Min(Minimum, Maximum); // an inverted range is
        var high = Math.Max(Minimum, Maximum); // normalised, not thrown on
        var next = Math.Clamp(value, low, high);
        if (Math.Abs(_value - next) < 0.001f)
            return; // nothing visual changed
        _value = next;
        AccessibleDescription = $"{Caption}: {_value:0.#}";
        Invalidate(); // exactly once per real change
    }
}
```

The second half of the control is `GaugeGeometry` : a helper that takes a rectangle and the normalised value and returns ring bounds, a start angle, a sweep, the needle points and the label rectangle. It touches no `Graphics` , so you can assert on its output in a plain test -- and rendering the same gauge off-screen in Module 6 then costs almost nothing.

Who owns which disposable

Ownership is what keeps a painted control out of trouble on a shared server. The handler owns whatever it creates -- pens, solid and gradient brushes, fonts, paths, any bitmap -- and a `using` declaration disposes each at the end of the method. It does not own `e.Graphics`, and disposing that is the classic mistake.

Two related habits matter just as much. Framework statics such as `Pens.Red` and `Brushes.Black` are shared and must be left alone. And a mutable pen parked in a `static` field is worse than a leak: a Wisej.NET server handles many sessions and may render for several at once, so rendering state stays local unless you deliberately design an immutable, thread-safe cache. The paint path also never opens a database connection, calls a service or starts a timer.

The cost budget, and the reading nobody can see

A painted control spends in three places: server CPU while drawing, server memory for the temporary image, and network plus browser decode for the picture sent. A status strip and a full-window heatmap are both just images to the browser, and their costs are nowhere near comparable, so measure all three when a surface grows. That is what `PaintCost.md` records.

Invalidation policy is the other half of the budget. Write the visual dependencies down -- typically value, caption, palette and size -- and repaint only for those. If a value moves ten times inside one request, update the state ten times but aim for one visual update at the end, and never run a server-side timer purely to animate an image. When the numbers stop justifying the pixels, the honest answer is a themed control, an HTML visual or a charting extension.

Finally, a drawn number is not an accessible number. A user relying on assistive technology receives an image, so the gauge publishes its reading through `AccessibleDescription` and the page keeps a textual equivalent beside the graphics. Add a fallback for a missing font or image and a minimal diagnostic state instead of an exception, and the control is safe to hand to another team.



Surface-first drawing habit

Choose the rendering surface before you write a single drawing call. Ask where the pixels are produced and what crosses the network: a control Paint handler runs on the server and ships an image, `CellPaint` does the same inside one grid cell, `Canvas` sends drawing commands the browser executes and keeps nothing across a resize, and `System.Drawing.Managed` produces image bytes with no control at all. Keep the semantic model in C# objects, make every renderer a deterministic function of that model, invalidate only when a visual dependency changed, and always leave a textual equivalent of anything important that exists only as pixels.



Learning objectives

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

- ✓ Handle the Paint event and work with `PaintEventArgs.Graphics` and `ClipRectangle`.
- ✓ Draw shapes, paths, text, images, gradients and transformed content with `System.Drawing`.
- ✓ Build a reusable custom painted control whose public properties invalidate it correctly.
- ✓ Apply resource-lifetime rules for pens, brushes, fonts, images and graphics objects.

- ✓ Recognize when painting cost, PNG transfer or accessibility makes another technique preferable.

Lab / Exam Guide

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



Hands-on lab

Add a reusable `TelemetryGauge` control to `VisualOperationsStudio` that paints a ring, coloured threshold zones, a needle, a numeric value and a caption. Give it `Value`, `Minimum`, `Maximum`, `WarningThreshold` and `CriticalThreshold` properties whose setters clamp the input, return early when nothing changed and call `Invalidate()` once. Put the coordinate maths in a pure `GaugeGeometry` helper that takes a `Rectangle` and returns the ring bounds, sweep angles, needle points and label rectangle, then draw from those values with using-scoped pens, brushes and a `GraphicsPath`, leaving the supplied `Graphics` alone. Handle a zero-sized control, an inverted range and a value outside the range without throwing, and expose the reading through `AccessibleDescription` so it is not pixels-only.

Starting point

You open `VisualOperationsStudio` from Module 1, where `VisualOperationsPage` already holds a single `TelemetrySample` model and shows its value on four surfaces, including a `Panel` with an inline `Paint` handler that draws a bar. In this lab that inline handler is replaced: you add a `TelemetryGauge` class to the project, drop three instances on the page, and bind each one to a reading from the same model.

Suggested approach

- ✓ Add a `TelemetryGauge` class deriving from a `Wisej.NET` control, give it backing fields plus `Value`, `Minimum`, `Maximum`, `WarningThreshold`, `CriticalThreshold` and `Caption`, and write one setter that clamps, compares, returns early and calls `Invalidate()` once -- then copy that shape to the others.
- ✓ Write `GaugeGeometry` as a pure helper: a method that takes the drawable `Rectangle` and the normalised value and returns ring bounds, the start and sweep angles, the needle points and the label rectangle, with no reference to `Graphics` anywhere in the file.
- ✓ Attach the `Paint` handler (or override the painting hook) and draw strictly from the geometry result: the inactive track, the warning and critical zones, the active arc, the needle built as a `GraphicsPath`, then the numeric value and caption measured with `MeasureString` and a `StringFormat`.
- ✓ Wrap every pen, brush, font and path you create in a `using` declaration, use framework statics such as `Brushes.Black` without disposing them, and wrap the needle's `RotateTransform` in `Graphics.BeginContainer()` and `EndContainer(state)` so the caption is drawn in untransformed coordinates.
- ✓ Guard the edges: return immediately when the rectangle is too small to draw, normalise an inverted range, clamp a value that falls outside it, set `AccessibleDescription` from the setter, then resize the browser, change the value from a button and record the picture size and repaint frequency in `PaintCost.md`.

Required deliverables

- ✓ TelemetryGauge control with clamping Value, Minimum, Maximum and threshold properties that Invalidate() once per real change
- ✓ Pure GaugeGeometry helper returning ring bounds, sweep angles, needle points and the label rectangle
- ✓ Paint handler that disposes every pen, brush, font and path it creates and never disposes e.Graphics
- ✓ Guards for zero size, inverted range and out-of-range values, with an AccessibleDescription carrying the reading
- ✓ PaintCost.md noting the repaint size, how often it repaints and when a themed control would be cheaper

Student review questions

- ✓ A colleague moves the ring and needle maths directly into the `Paint` handler because "it is only a few lines". Which two things become impossible to do cheaply afterwards, and which one of them does Module 6 depend on?
- ✓ Your gauge is bound to a machine record that also carries an operator name and a maintenance date. If those fields change, should the control invalidate? Explain your answer in terms of visual dependencies rather than in terms of the model.
- ✓ The same gauge is going onto a dashboard that shows sixty of them and refreshes every two seconds. What would you measure first, and what would you change if the measurements were bad?



Surface-first drawing habit

Choose the rendering surface before you write a single drawing call. Ask where the pixels are produced and what crosses the network: a control Paint handler runs on the server and ships an image, CellPaint does the same inside one grid cell, Canvas sends drawing commands the browser executes and keeps nothing across a resize, and System.Drawing.Managed produces image bytes with no control at all. Keep the semantic model in C# objects, make every renderer a deterministic function of that model, invalidate only when a visual dependency changed, and always leave a textual equivalent of anything important that exists only as pixels.