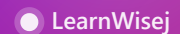


DataGridView Cell Validation and Row Errors

Lesson Guide · Lab / Exam Guide



Lesson Guide

Validation at grid scale: CellValidating with RowIndex, ColumnIndex and the uncommitted FormattedValue, CellValidated for cleanup, cell ErrorText versus row ErrorText, DataError with ThrowException set to false, and new-row and bound-row rules that do not break editing. What this means for a developer who knows basic C# and is beginning Wisej.NET form development — and how to read this module.



Production focus

Validation at grid scale: CellValidating with RowIndex, ColumnIndex and the uncommitted FormattedValue, CellValidated for cleanup, cell ErrorText versus row ErrorText, DataError with ThrowException set to false, and new-row and bound-row rules that do not break editing.

1. The production problem

Modules 1 to 5 protected the ValidationClinic intake form, one contact at a time. The contacts grid is different: a user can tab through twenty rows, type an email without an @, enter "abc" as an age and mark a contact Closed without a closed date, all without opening a form. A grid cell is not a TextBox, so the `Validating` handlers and the `ErrorProvider` you already wrote do not reach it. This module gives the grid its own field rules, row rules and data-source rules, and makes sure none of them turns a normal typing mistake into a crash or a stuck screen.

2. Key concepts

Concept	Why it matters
<code>CellValidating</code>	The grid event that fires when the current cell is about to lose focus; it is the grid's version of a control's <code>Validating</code> event and the place to reject a single value.
<code>e.FormattedValue</code>	The value exactly as the user typed it in the editor, before it is committed to the cell or the data source; reading the cell's <code>Value</code> instead gives you the old value.
<code>e.Cancel</code>	Setting it to <code>true</code> refuses to finish the edit, so the cursor stays in the bad cell until the user corrects it or cancels the edit.
<code>CellValidated</code>	Fires after a cell has passed validation; a safe place to clear an error that no longer applies or refresh a summary.
Cell <code>ErrorText</code>	A message on one cell, shown with an error icon inside that cell; use it for a problem with a single value.
Row <code>ErrorText</code>	A message on the whole row, shown in the row header; use it for a rule that depends on several cells in the same row.
<code>DataError</code>	Raised when the grid cannot convert, format or commit a value; with <code>e.ThrowException = false</code> you replace the exception with a friendly message.

3. Lesson narrative

CellValidating and CellValidated: when the grid checks a value

On a form, each `TextBox` raises its own `Validating` event. A `DataGridView` has one event for every cell: `CellValidating`. It fires when the user leaves the cell they were editing, by pressing Tab or Enter or clicking another cell. Because one handler serves every column, the first thing it does is work out which cell it is looking at, and then it runs only the rule for that column. If the value is wrong, you set `e.Cancel = true` and the grid keeps the user in that cell, exactly as `e.Cancel` held focus on a `TextBox` in Module 1.

`CellValidated` is the calm follow-up. It fires only after a cell has been accepted, with ordinary `DataGridViewCellEventArgs`, so it is a good place to remove an error that is now stale or to update a count of remaining problems. In `ValidationClinic` you wire both events on `gridContacts`: the first decides, the second tidies up.

RowIndex, ColumnIndex and FormattedValue

The event arguments, `DataGridViewCellValidatingEventArgs`, carry three things you need. `e.RowIndex` and `e.ColumnIndex` say where the edit happened. Use the column index to look up the column, then compare `grid.Columns[e.ColumnIndex].Name` with a stable name such as `"EmailColumn"`. A name survives someone reordering columns in the designer; a raw number does not.

The third is `e.FormattedValue`, and it is the most important idea in this module. While the user is typing, the new text lives in the editor, not in the cell. The cell's `Value` still holds what was there before the edit

started. If your rule reads `Value`, it validates the old, already-accepted value and lets the new bad one through. Always convert `e.FormattedValue` to text, trim it, and test that.

```
private void gridContacts_CellValidating(object sender, DataGridViewCellValidatingEventArgs e)
{
    var grid = (DataGridView)sender;
    var cell = grid.Rows[e.RowIndex].Cells[e.ColumnIndex];
    var text = Convert.ToString(e.FormattedValue)?.Trim() ?? "";
    cell.ErrorText = "";

    if (grid.Columns[e.ColumnIndex].Name == "EmailColumn" && !text.Contains('@'))
    {
        e.Cancel = true;
        cell.ErrorText = "Enter a valid email address.";
    }
}
```

Notice the order: clear the cell's old message first, then check, then set a new message together with `e.Cancel`. An error message without `e.Cancel` only decorates the cell; the bad value is still committed.

Cell errors and row errors

The grid has two places to show a problem. `grid.Rows[r].Cells[c].ErrorText` belongs to one cell and draws an error icon inside it, with the message as a tooltip. Use it for rules about a single value: the email is blank, the age is outside 0 to 120. `row.ErrorText` belongs to the whole row and appears in the row header. Use it when the rule needs more than one cell to decide.

ValidationClinic's example is the Closed rule: a contact whose status is Closed must have a closed date. Neither cell is wrong on its own, so a `ValidateRow(rowIndex)` method reads `row.Cells["StatusColumn"].Value` and the closed-date cell, sets `row.ErrorText` to "Closed contacts require a closed date." and also marks the `ClosedDateColumn` cell so the user can see which cell to fix. When the rule passes, the method clears both. Clearing matters as much as setting: a row that still shows yesterday's message teaches users to ignore every message.

DataError: conversion problems without scary exceptions

Some mistakes are not about your rules at all. If the Age column is bound to an integer and the user types "abc", the grid cannot convert the text, and if a data source refuses a value, committing it fails. In both cases the grid raises `DataError` with `DataGridViewDataErrorEventArgs`. Unhandled, the exception surfaces as raw .NET text.

A good handler does three things. It sets `e.ThrowException = false` so the exception is not rethrown. It checks that `e.RowIndex` and `e.ColumnIndex` are zero or greater, then puts a plain message on that cell, such as "The value cannot be converted or saved. Check the format." And it writes `e.Exception` to a log, for example with `System.Diagnostics.Debug.WriteLine`, so developers still see the real cause. The user gets something to fix, and the developer gets the real detail. Do not use exceptions for ordinary user mistakes in your own rules either; setting `ErrorText` and `e.Cancel` is the normal path.

New rows and bound rows without breaking editing

An editable grid usually shows an empty placeholder row at the bottom where the user can start a new contact. Until someone types in it, required-field rules would put errors on a row the user never touched. Skip the placeholder row, and skip header positions where an index is negative, until the row really contains user input. Then validate its required columns before it is committed.

In a bound grid the binding source and the model can also reject a value. Let `DataError` handle conversion and binding failures, and let the model validation from Module 5 own the business rules, so the grid does not copy every rule the form already has. Keep `CellValidating` fast: it runs on every cell exit, so a check that queries the database there makes the whole grid feel slow. Finally, before Save, loop over the rows, skip the placeholder, and refuse to save while any row still has an error. Module 7 folds that grid check into the complete Save pipeline.



Common pitfalls

Reading `cell.Value` instead of `e.FormattedValue` and missing the edit in progress. Setting `ErrorText` without `e.Cancel = true` when the edit must be rejected. Throwing exceptions for normal user mistakes. Showing raw conversion or database exception text. Validating against the whole database from inside `CellValidating`.



Validation layers habit

Name the layer before you write the check. Field validation protects editing and runs when the user leaves a control; model validation protects the data and runs the same rules from the form, the grid and any service; save validation protects the database and runs every layer again before anything is written. Show every error near its field and in a visible summary, in plain language that says what to do next, and keep user-correctable mistakes separate from system failures.



Learning objectives

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

- ✓ Handle `DataGridView.CellValidating` and `CellValidated`.
- ✓ Use `RowIndex`, `ColumnIndex` and `FormattedValue` safely.
- ✓ Set cell `ErrorText` and row `ErrorText` for grid feedback.
- ✓ Handle `DataError` for parsing and data-source exceptions.
- ✓ Validate new rows and bound rows without breaking edit flow.

Lab / Exam Guide

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



Hands-on lab

Make the ValidationClinic contacts grid refuse bad data. Handle `gridContacts.CellValidating`: read `e.FormattedValue` rather than the cell value, clear the cell's `ErrorText`, then reject a blank or @-less EmailColumn value and an AgeColumn value outside 0 to 120 with `e.Cancel = true` and a cell `ErrorText`. Add a `ValidateRow` method that sets row `ErrorText` and the `ClosedDateColumn` cell error when Status is Closed without a closed date, clear stale errors in `CellValidated`, and handle `DataError` with `ThrowException = false`, a friendly cell message and a debug log of the exception. Skip the new placeholder row until it holds user data, and block Save while any row still has an error.

Starting point

ValidationClinic after Module 5: the intake form validates its fields and its bound model, but the contacts page only has `gridContacts` with the columns `NameColumn`, `EmailColumn`, `AgeColumn`, `StatusColumn` and `ClosedDateColumn`, bound to the in-memory contact repository, and no grid validation at all. Today you can type "abc" as an age, leave an email empty and save a Closed contact with no closed date.

Suggested approach

- ✓ Before writing code, break the grid on purpose: enter an email without an @, type "abc" in Age, and set a row to Closed with an empty closed date. Write down what the user sees in each case, including any raw exception text.
- ✓ Wire `gridContacts_CellValidating`. Convert `e.FormattedValue` to trimmed text, clear the cell's `ErrorText`, look up the column name from `e.ColumnIndex`, and reject a blank or @-less email and an age outside 0 to 120 with `e.Cancel = true` and a plain-language cell message.
- ✓ Write `ValidateRow(int rowIndex)` for the Closed rule. Read `Cells["StatusColumn"].Value` and the closed-date cell, then set or clear both `row.ErrorText` and the `ClosedDateColumn` cell error. Call it once the row's edits are accepted, and clear stale cell errors in `gridContacts_CellValidated`.
- ✓ Wire `gridContacts_DataError`: set `e.ThrowException = false`, check that `e.RowIndex` and `e.ColumnIndex` are not negative before touching a cell, put a friendly message on that cell and log `e.Exception` with `Debug.WriteLine`.
- ✓ Guard `btnSave_Click`: run `ValidateRow` for every row, skip the empty placeholder row at the bottom, and stop with a summary message while any row or cell still has `ErrorText`. Then repeat your broken inputs and confirm each one is now caught and explained.

Required deliverables

- ✓ `CellValidating` handler that reads `e.FormattedValue` and validates the email and age columns
- ✓ `ValidateRow` method setting row `ErrorText` and the closed-date cell error for Closed contacts
- ✓ `CellValidated` handler that clears stale cell errors

- ✓ DataError handler with `ThrowException = false`, a friendly message and a logged exception
- ✓ Save guard that skips the placeholder new row and refuses to save while any row has an error

Student review questions

- ✓ A classmate's email rule reads `grid.Rows[e.RowIndex].Cells[e.ColumnIndex].Value` and "works in testing". Describe the exact edit that slips through, and why.
- ✓ Your age rule checks the range 0 to 120. What should it do when the text is not a number at all, and why is a `DataError` handler still needed even if the rule rejects "abc" itself?
- ✓ The Closed rule could be shown as a cell error, a row error, or both. Which did you choose, and what would a user who never hovers over an icon still see?



Validation layers habit

Name the layer before you write the check. Field validation protects editing and runs when the user leaves a control; model validation protects the data and runs the same rules from the form, the grid and any service; save validation protects the database and runs every layer again before anything is written. Show every error near its field and in a visible summary, in plain language that says what to do next, and keep user-correctable mistakes separate from system failures.