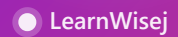


Built-in Validation Rules and the Validation Extender

Lesson Guide · Lab / Exam Guide



Lesson Guide

The Wisej.NET Validation extender: the ValidationRules extended property at design time, SetValidationRules at runtime, the Required, Email, Integer, Decimal, Currency, Telephone and Regex rules, rule order that stops at the first failure, and ErrorProvider or Label display targets. 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

The Wisej.NET Validation extender: the ValidationRules extended property at design time, SetValidationRules at runtime, the Required, Email, Integer, Decimal, Currency, Telephone and Regex rules, rule order that stops at the first failure, and ErrorProvider or Label display targets.

1. The production problem

At the end of Module 2 the ValidationClinic intake form works, but look at how much code it took: a helper for the name, a helper for the email, a helper for the phone, and each one repeats the same "is it blank, is it the right shape" checks. Add age, credit limit and a customer code and the form grows another handful of near-identical methods. The checks are not wrong; they are just typed out again on every field and every form. This module replaces that repetition with the `Validation` extender, where you describe what a valid field looks like and Wisej.NET runs the checks for you.

2. Key concepts

Concept	Why it matters
<code>Validation</code> extender	A non-visual component you add to a form; it attaches to the <code>Validating</code> and <code>Validated</code> events of every control that has rules, so you no longer write a handler per field.
extended property	A property that a component adds to other controls; once the extender is on the form, each compatible control shows a <code>ValidationRules</code> entry in the Properties window.
<code>SetValidationRules</code>	The code equivalent of the designer: <code>validation.SetValidationRules(txtEmail, new ValidationRule[] { ... })</code> gives one control its list of rules at runtime.
built-in rules	Ready-made checks for common input: <code>RequiredValidationRule</code> , <code>EmailValidationRule</code> , <code>IntegerValidationRule</code> , <code>DecimalValidationRule</code> , <code>CurrencyValidationRule</code> , <code>TelephoneValidationRule</code> and <code>RegexValidationRule</code> .
rule order	Rules run in the order you list them; the first one that fails stops the rest and cancels the control's <code>Validating</code> event.
display target	Where the failing rule's <code>InvalidMessage</code> appears: on the control itself, through an <code>ErrorProvider</code> icon, or as text in a Label, all reached through <code>validation.ErrorProvider</code> .
rule configuration helper	A small method that builds the rule arrays for common fields, so a second form gets the same checks and the same wording by calling it.

3. Lesson narrative

One component, two ways to configure it

The `Validation` extender is a component, like a timer or the `ErrorProvider` from Module 2: it has no visible surface of its own. Drop it from the Toolbox onto the intake form in the Visual Studio designer and it appears in the component tray under the form. From that moment every compatible control on the form gains a new entry in the Properties window called `ValidationRules`. Select `txtEmail`, open that collection editor, and add the rules the field needs. The designer writes them into the form's designer file, so anyone opening the form can see which fields are required without reading a single event handler.

The same thing can be done in code, which is what the lab uses. You create the component, point its `ContainerControl` at the form, and call `SetValidationRules` once per control, usually from a `ConfigureValidationRules` method called in the form's constructor. Both routes produce the same result: the extender hooks the control's validation events and runs the rules when the user leaves the field, exactly the moment Module 1 taught you to handle by hand.

```

errorProvider.ContainerControl = this;
validation.ContainerControl = this;
validation.ErrorProvider = errorProvider;

validation.SetValidationRules(txtEmail, new ValidationRule[]
{
    new RequiredValidationRule { InvalidMessage = "Email is required.", Trim = true },
    new EmailValidationRule { InvalidMessage = "Enter a valid email address." }
});
validation.SetValidationRules(txtAge, new ValidationRule[]
{
    new RequiredValidationRule { InvalidMessage = "Age is required." },
    new IntegerValidationRule { InvalidMessage = "Age must be a whole number." }
});

```

The seven built-in rules

Each rule answers one question about the text in a control. `RequiredValidationRule` asks "is there anything here?" and has three useful properties: `Required` switches the check on, `Trim` removes spaces first so a field containing only blanks still counts as empty, and `MinimumLength` asks for at least a certain number of characters. `EmailValidationRule` checks that the value looks like an email address.

`IntegerValidationRule` accepts whole numbers such as an age, while `DecimalValidationRule` accepts numbers with a fractional part. `CurrencyValidationRule` checks a money amount such as the credit limit and can format the value once it is accepted. `TelephoneValidationRule` checks a phone number against a mask, and `RegexValidationRule` compares the value with a regular expression, which is how ValidationClinic enforces a customer code made of the letters CUS, a dash and four digits.

Every rule carries an `InvalidMessage`. That text is what the user reads, so write it as product copy that says what to do next: "Age must be a whole number." rather than "Integer parse failed".

Rule order and the first failure

The rules on a control run from first to last. As soon as one fails, the remaining rules are skipped and the control's `Validating` event is canceled, which has the same effect as setting `e.Cancel = true` yourself in Module 1. Only one message is shown: the message of the rule that failed first.

That is why order is a design decision. Put `RequiredValidationRule` first on every required field. Leave the email box empty and the user reads "Email is required.", which is the real problem. If the Email rule came first, the same empty box would say "Enter a valid email address.", which is technically true and much less helpful. In the lab you will prove this with a blank `txtEmail`: Required fails, Email never runs, and focus behaves according to the `AutoValidate` mode you chose earlier. Fields that are optional, such as the credit limit, simply start with their format rule.

Choosing where the message appears

Without a display target, a failing rule writes its message back to the control itself as an invalid tooltip. Assign an `ErrorProvider` to `validation.ErrorProvider` and the extender calls `SetError` on it instead, giving you the icon and tooltip from Module 2 with no `SetError` calls in your own code. A Label can also be the target: it shows the message as plain text and hides itself when there is nothing to say, which suits a

short form or a field that needs visible text rather than an icon. Any class implementing `IErrorProvider` can play this role; Module 7 builds one that feeds a summary panel.

The extender also raises its own `Validating` and `Validated` events for every control that has rules. That gives you one central place to refresh `validationSummaryLabel`, so the visible summary from Module 2 stays in step with the icons even though you no longer write the checks yourself.

Reusable rule configurations

Email, age and phone appear on more than one ValidationClinic screen. Instead of copying the arrays, put them in a small helper, for example a static `ClinicRules` class with methods such as `RequiredEmail()` and `RequiredAge()` that each return a fresh `ValidationRule[]`. The intake form and any second form call the same methods, so the order and the wording of every message are decided once.

Keep the helper about field shape only. Rules that compare two fields, look something up in a database, or express a business policy do not belong in the extender, because they would only run when a control loses focus. Module 4 moves those into validation methods, a validator service and custom `ValidationRule` classes.



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:

- ✓ Use the Wisej.NET Validation extender at design time and runtime.
- ✓ Apply Required, Email, Integer, Decimal, Currency, Telephone and Regex rules.
- ✓ Explain rule order and why validation stops at the first failure.
- ✓ Assign an ErrorProvider or Label to display validation messages.
- ✓ Create reusable rule configurations for common forms.

Lab / Exam Guide

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



Hands-on lab

Replace the hand-written field handlers in ValidationClinic with the Validation extender. Add a Validation component and an ErrorProvider to the intake form, set ContainerControl and validation.ErrorProvider, then call SetValidationRules in a ConfigureValidationRules method: Required then Email on txtEmail, Required then Integer on txtAge, Required then Telephone on txtPhone, Currency on txtCreditLimit and a Regex rule for the customer code. Put Required first everywhere so the most helpful message wins, hook the extender's Validating event to refresh the summary, and move the rule arrays into a small reusable helper so another form can apply the same configuration.

Starting point

You open ValidationClinic as Module 2 left it: the intake form has `txtName`, `txtEmail`, `txtPhone` and `txtAge`, an `ErrorProvider`, the `ValidateName`, `ValidateEmail` and `ValidatePhone` helpers and a `validationSummaryLabel`. Add a `txtCreditLimit` and a `txtCustomerCode` TextBox for this lab if they are not on the form yet.

Suggested approach

- ✓ Add a `Validation` component to the intake form in the designer, select `txtEmail` and add a Required rule and an Email rule through the `ValidationRules` collection editor, then run the form once to see the extender work before writing any code.
- ✓ Remove those designer rules and write `ConfigureValidationRules`, called from the constructor: set `errorProvider.ContainerControl`, `validation.ContainerControl` and `validation.ErrorProvider`, then call `SetValidationRules` for email, age, phone, credit limit and customer code.
- ✓ Give every required field `RequiredValidationRule` as its first rule, with `Trim = true` where spaces should not count, and write each `InvalidMessage` as a sentence that tells the user what to type.
- ✓ Handle the extender's `Validating` and `Validated` events in one place and rebuild `validationSummaryLabel` from the current errors, hiding it when none remain.
- ✓ Move the rule arrays into a static `ClinicRules` helper whose methods return a fresh `ValidationRule[]`, call it from the intake form, then test a blank email, a malformed email, a valid email, a non-numeric age and a wrong customer code.

Required deliverables

- ✓ Validation component wired to the form and to an ErrorProvider display target
- ✓ ConfigureValidationRules method calling SetValidationRules for email, age, phone, credit limit and customer code
- ✓ Required rule placed first on every required field, with plain-language InvalidMessage text

- ✓ Extender Validating handler that keeps the validation summary in step with the rules
- ✓ Reusable rule configuration helper that a second form can call

Student review questions

- ✓ You swap the order so the Email rule runs before Required on `txtEmail`. Describe exactly what a user sees after leaving the field empty, and why that message is less useful.
- ✓ The Module 2 helpers and the extender both end with an icon beside the field. What did you stop writing by hand, and what did you still have to decide yourself?
- ✓ A colleague wants to add "the end date must be after the start date" to `ClinicRules`. Why is the extender the wrong home for that check, and when would it fail to run?



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.