

1 Lecture 3: Multiple Forms, Navigation, and Reusable Components

Course: Scientific Visual Programming

Topic: Building Multi-Form Windows Applications

Recommended duration: 90 minutes

1.1 Learning objectives

Students should be able to create multiple forms, choose between modal and modeless windows, pass data between forms, use dialogs for focused tasks, and organize reusable user controls.

1.2 1. Why use multiple forms?

A small application may fit on one form, but a growing application needs structure. Separate forms can represent different tasks such as login, dashboard, data entry, reports, and settings. Each form should have a focused responsibility and a clear navigation path.

Splitting the interface does not automatically improve design. Too many windows can confuse users. Use a new form when the task has a distinct purpose, needs its own layout, or must be opened independently.

1.3 2. Opening a form

A form can be shown modally with `ShowDialog` or modelessly with `Show`. A modal form requires the user to complete or close it before returning to the owner form. This is appropriate for a settings dialog or a required data-entry step.

```
private void btnOpenSettings_Click(object sender, EventArgs e)
{
    using (var settingsForm = new SettingsForm())
    {
        if (settingsForm.ShowDialog(this) == DialogResult.OK)
        {
            ApplySettings(settingsForm.SelectedSettings);
        }
    }
}
```

A modeless form remains open while the user works with other windows. It is useful for a tool window or an independent report, but the application must manage duplicate instances and closing behavior.

1.4 3. Passing data through constructors

Passing required data through a constructor makes dependencies explicit:

```
public partial class StudentDetailsForm : Form
{
    private readonly int studentId;
```

```

public StudentDetailsForm(int studentId)
{
    InitializeComponent();
    this.studentId = studentId;
}
}

```

Avoid using global variables to pass every value between forms. Global state makes the application difficult to test and can create unexpected coupling.

1.5 4. Returning a result from a dialog

A dialog can expose selected data through a property and return `DialogResult.OK` when the user confirms. The owner form then decides what to do with that result. This keeps the dialog focused on collecting input.

```

public string SelectedDepartment => cmbDepartment.Text;

private void btnOk_Click(object sender, EventArgs e)
{
    if (string.IsNullOrEmpty(SelectedDepartment))
    {
        MessageBox.Show("Select a department.");
        return;
    }

    DialogResult = DialogResult.OK;
    Close();
}

```

1.6 5. Main form and navigation patterns

One pattern uses a main form with a menu and opens child forms as dialogs. Another uses a panel as a content area and loads user controls into it. The second approach can provide a dashboard-like experience and reduce the number of top-level windows.

When using a panel, remove or dispose the previous control before adding the new one. Keep navigation commands in one place so that the interface behaves consistently.

1.7 6. User controls

A `UserControl` is a reusable group of controls and behavior. For example, an address editor can combine street, city, postal code, and validation into one component. User controls promote reuse and make complex forms easier to maintain.

A user control should expose a small, meaningful public interface. It should not force every parent form to know how its internal text boxes and labels are arranged.

1.8 7. The MDI approach

Multiple Document Interface (MDI) allows a parent form to host child forms. It can be useful for applications that manage several documents, but it can also produce a crowded interface. Use MDI only when the document-oriented workflow benefits from it.

1.9 8. Example: student management application

A student management application may contain a login form, a main dashboard, a student list, a student editor, and a report form. The login form returns an authenticated user. The dashboard opens the student list. The student list opens the editor with a student identifier. The editor returns a saved result, and the list refreshes its data.

This flow creates explicit transitions and avoids keeping separate, conflicting copies of the same student object in many forms.

1.10 9. Resource and lifecycle management

Forms and dialogs should be disposed when they are no longer needed. The `using` statement is a convenient pattern for short-lived modal dialogs. Event subscriptions and timers should also be stopped or released when a form closes to avoid memory leaks or events targeting disposed controls.

1.11 Practice activity

Create a small library application with a main form, a book list form, and an add-book dialog. The dialog must return the title and author only after validation. The list form must refresh after a book is added. Add a reusable `SearchBoxControl` containing a text box and a clear button.

1.12 Summary

Multi-form applications are easier to understand when each form has a focused responsibility. Use constructors for required data, dialog results for controlled return values, and user controls for reusable interface components. Choose modal, modeless, panel-based, or MDI navigation according to the workflow.

1.13 References