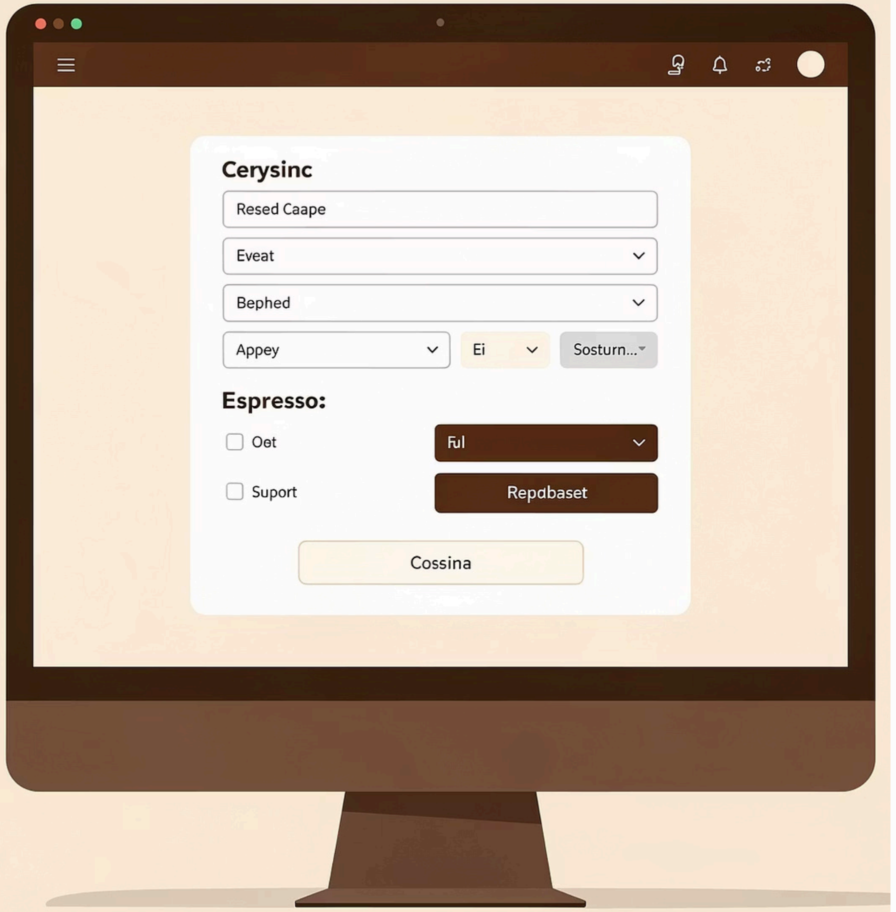


Windows Forms and Controls

Pravin Nikam

Building Desktop Applications with Windows Forms

In this unit, you'll learn how to create professional desktop applications using the Windows Forms framework in .NET. We'll cover everything from basic form design to advanced controls, menus, and dialogs. By the end, you'll be able to build complete GUI-based desktop applications that respond to user interactions through event-driven programming.

Windows Forms provides a powerful way to create Windows desktop applications with rich user interfaces. Think of it as your toolkit for building programs that users can interact with through buttons, text boxes, menus, and other visual elements—just like the desktop applications you use every day.

What You'll Learn

Windows Forms Basics

Understand the Windows Forms model, architecture, and how to create your first GUI application

Properties & Events

Master form properties and event handling to create interactive, responsive applications

Controls & UI Elements

Work with buttons, text boxes, labels, checkboxes, and other essential controls

Advanced Features

Implement menus, dialogs, tooltips, and other professional UI elements

FOUNDATION

What is Windows Forms?

Windows Forms is a GUI (Graphical User Interface) framework that comes built into .NET. It provides a complete set of tools and components for building desktop applications that run on Windows operating systems.

Think of it like this: If HTML/CSS is for building web pages, Windows Forms is for building desktop programs. It gives you all the visual building blocks—windows, buttons, text boxes—that users expect from Windows applications.

Key Characteristics

- Part of the .NET Framework
- Designed for Windows desktop apps
- Built using C# or VB.NET
- Created in Visual Studio IDE
- Event-driven programming model

The Windows Forms Model

Windows Forms follows a simple but powerful architecture. At its core, a **Form** is a window—the container where you place all your interactive elements. **Controls** are the individual UI components you add to the form, like buttons and text boxes.

The magic happens through **event-driven programming**. When a user clicks a button or types in a text box, an event fires. You write code (called an event handler) that responds to these events. This is fundamentally different from traditional programming where code runs top-to-bottom—here, your code waits and responds to user actions.

The **code-behind model** means your visual design (the form) is separated from your logic (the C# code). The designer generates the UI code automatically, and you focus on writing the behavior.

The Four Core Concepts

01

Form = Window

A Form is simply a window in your application. It's the canvas where everything else lives.

02

Controls = UI Elements

Controls are the interactive pieces—buttons, text boxes, labels—that you place on the form.

03

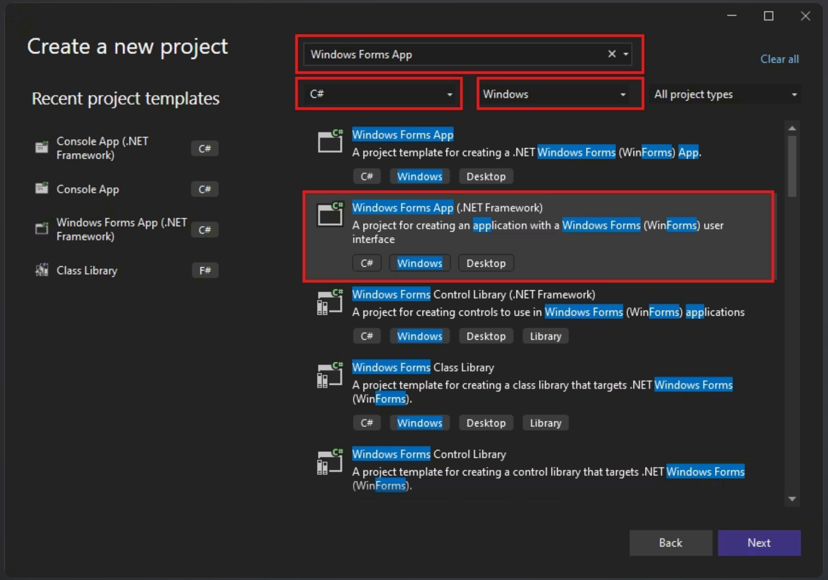
Events = User Actions

Events are triggered when users interact with your app—clicks, key presses, mouse movements.

04

Code-Behind = Your Logic

Your C# code responds to events and implements the application's behavior and business logic.


Create a new project

Windows Forms App

Clear all

C# | Windows | All project types

Recent project templates

- Console App (.NET Framework) C#
- Console App C#
- Windows Forms App (.NET Framework) C#
- Class Library F#

Windows Forms App
A project template for creating a .NET Windows Forms (WinForms) App.
C# Windows Desktop

Windows Forms App (.NET Framework)
A project for creating an application with a Windows Forms (WinForms) user interface
C# Windows Desktop

Windows Forms Control Library (.NET Framework)
A project for creating controls to use in Windows Forms (WinForms) applications
C# Windows Desktop Library

Windows Forms Class Library
A project template for creating a class library that targets .NET Windows Forms (WinForms).
C# Windows Desktop Library

Windows Forms Control Library
A project template for creating a control library that targets .NET Windows Forms (WinForms).

Back Next

Creating Your First Windows Forms Application

Starting a Windows Forms project in Visual Studio is straightforward. Open Visual Studio, select "Create a new project," and choose "Windows Forms App (.NET Framework)" or "Windows Forms App" for .NET Core/.NET 6+.

Visual Studio creates a default form (Form1) with a design surface where you can drag and drop controls. The Solution Explorer shows your project structure, and the Properties window lets you configure every aspect of your form and controls. You'll switch between Design view (visual editor) and Code view (C# code) as you build your application.

Essential Form Properties

Why Properties Matter

Properties control how your form looks and behaves. They're like settings that you can adjust to customize your application's appearance and functionality.

Text

The title displayed in the window's title bar

Size

Width and height of the form window (in pixels)

StartPosition

Where the form appears when launched (center, manual, etc.)

BackColor

Background color of the form

Font

Default font for text on the form

Setting Properties: Two Ways

Method 1: Properties Window

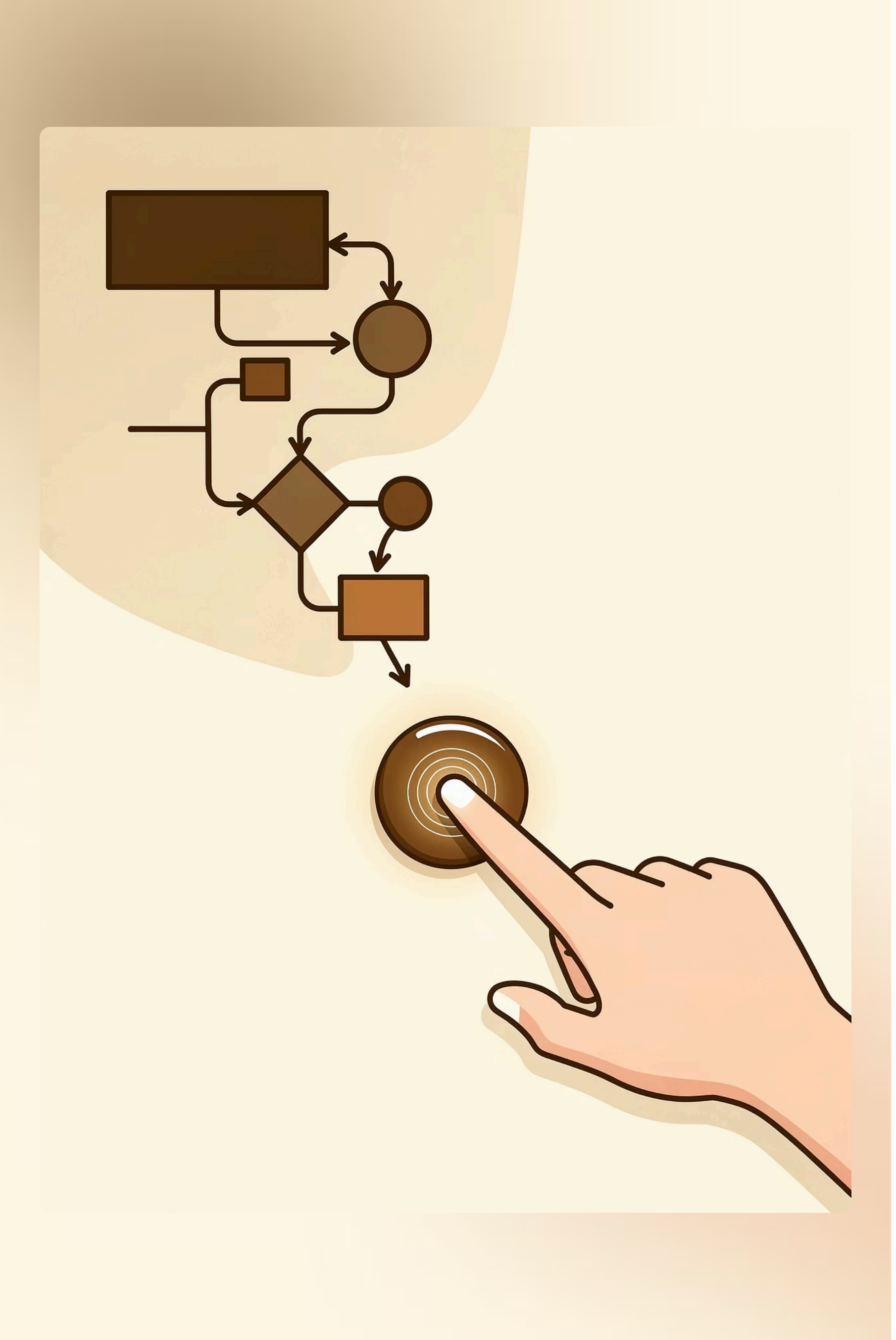
The easiest way for beginners. Select your form in the designer, and the Properties window shows all available properties. Simply find the property you want to change and type the new value.

Example: To change the form title, find "Text" in the Properties window and type "My First App".

Method 2: Write Code

You can also set properties programmatically in your C# code. This is useful when you need to change properties dynamically at runtime based on user actions or program logic.

```
this.Text = "My First App";
this.Size = new Size(800, 600);
this.BackColor = Color.White;
```



Understanding Event-Driven Programming

Windows Forms applications don't run line-by-line from start to finish like a simple console program. Instead, they wait for events to happen and then respond. This is called **event-driven programming**.

Real-world analogy: Think of a vending machine. It doesn't do anything until you interact with it—press a button, insert money, etc. Each action triggers a specific response. Your Windows Forms app works the same way: it waits quietly until the user clicks a button, types in a text box, or performs another action.

Events are the bridge between user actions and your code. When an event fires, the corresponding event handler (a method you write) executes automatically.

Common Events You'll Use

Button Click

Fires when user clicks a button—the most common event you'll handle

TextChanged

Fires whenever text in a TextBox is modified by the user

Form Load

Fires when the form first loads—perfect for initialization code

KeyPress

Fires when a key is pressed while a control has focus

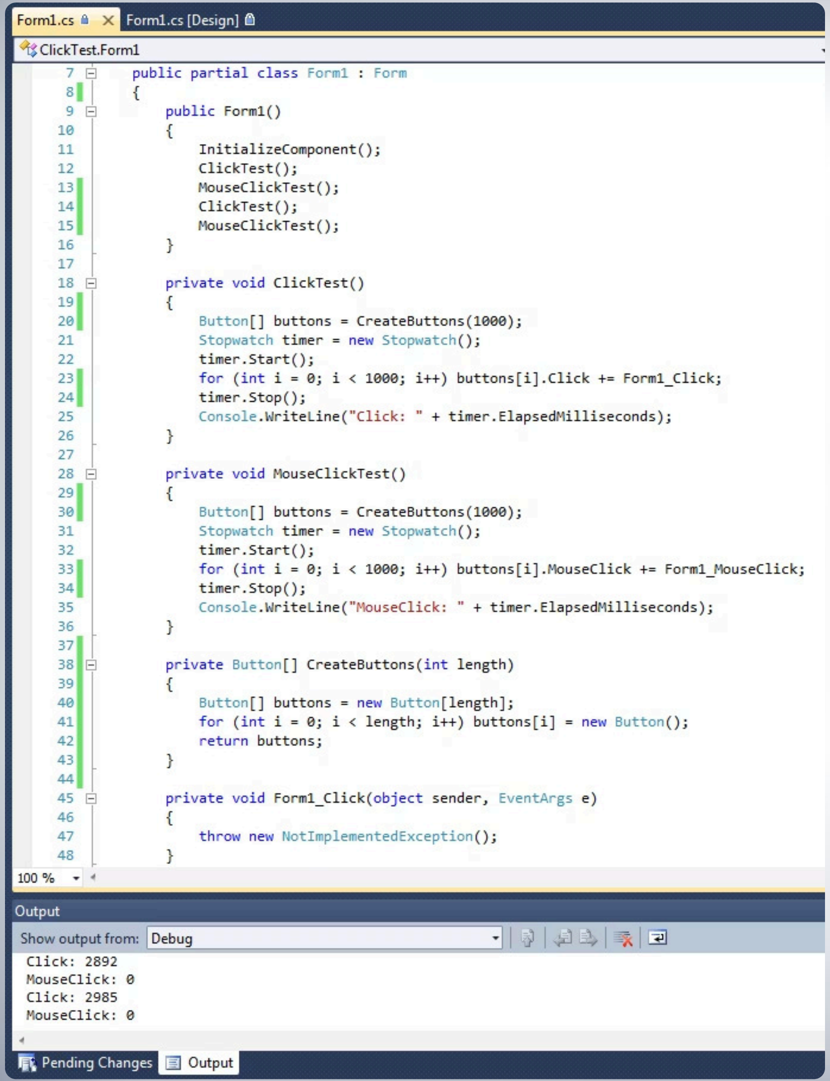
Writing an Event Handler

Creating an event handler is simple. Double-click any control in the designer, and Visual Studio automatically creates an event handler method and wires it up. The method signature follows a standard pattern:

```
private void button1_Click(object sender, EventArgs e)
{
    // Your code here runs when button is clicked
    MessageBox.Show("Hello, Windows Forms!");
}
```

The `sender` parameter tells you which control triggered the event, and `EventArgs e` provides additional information about the event (though for simple events like Click, it's often not used).

Pro tip: Use descriptive names for your controls (like "btnSubmit" instead of "button1") so your event handler names make sense (btnSubmit_Click is clearer than button1_Click).



```
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            ClickTest();
            MouseClickTest();
            ClickTest();
            MouseClickTest();
        }

        private void ClickTest()
        {
            Button[] buttons = CreateButtons(1000);
            Stopwatch timer = new Stopwatch();
            timer.Start();
            for (int i = 0; i < 1000; i++) buttons[i].Click += Form1_Click;
            timer.Stop();
            Console.WriteLine("Click: " + timer.ElapsedMilliseconds);
        }

        private void MouseClickTest()
        {
            Button[] buttons = CreateButtons(1000);
            Stopwatch timer = new Stopwatch();
            timer.Start();
            for (int i = 0; i < 1000; i++) buttons[i].MouseClick += Form1_MouseClick;
            timer.Stop();
            Console.WriteLine("MouseClick: " + timer.ElapsedMilliseconds);
        }

        private Button[] CreateButtons(int length)
        {
            Button[] buttons = new Button[length];
            for (int i = 0; i < length; i++) buttons[i] = new Button();
            return buttons;
        }

        private void Form1_Click(object sender, EventArgs e)
        {
            throw new NotImplementedException();
        }
```

```
Click: 2892
MouseClick: 0
Click: 2985
MouseClick: 0
```

Let's Build: Simple Login Form

Time to put everything together! We'll create a basic login form with username and password fields. This exercise demonstrates form design, control placement, property configuration, and event handling—all the fundamentals in one practical example.

What you'll create: A form with two text boxes (username and password), a login button, and a label to show login status. When the button is clicked, we'll validate the inputs and display a message.

Login Form: Step-by-Step

01

Add Controls

Drag two Labels (for "Username:" and "Password:"), two TextBoxes, and one Button onto your form

02

Set Properties

Name your controls: txtUsername, txtPassword, btnLogin. Set the Password TextBox's PasswordChar to "*"

03

Arrange Layout

Position controls neatly. Align the labels and text boxes. Set form's Text property to "Login"

04

Add Event Handler

Double-click the Login button to create a Click event handler and write your validation code

Login Form: The Code

```
private void btnLogin_Click(object sender, EventArgs e)
{
    string username = txtUsername.Text;
    string password = txtPassword.Text;

    if (string.IsNullOrEmpty(username) || string.IsNullOrEmpty(password))
    {
        MessageBox.Show("Please enter both username and password",
                        "Validation Error");
    }
    else if (username == "admin" && password == "1234")
    {
        MessageBox.Show("Login successful!", "Success");
    }
    else
    {
        MessageBox.Show("Invalid credentials", "Error");
    }
}
```

This code reads the text from both text boxes, validates that they're not empty, and checks against hardcoded credentials. In a real application, you'd check against a database.

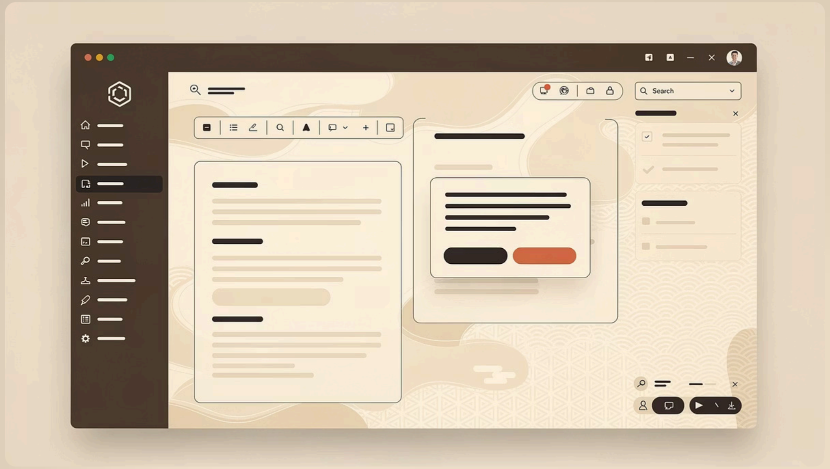
Enable/Disable Controls Dynamically

Often you'll want to enable or disable controls based on application state. For example, disable a Submit button until a user agrees to terms, or disable text boxes during processing.

Every control has an `Enabled` property. Setting it to `false` makes the control visible but grayed out and non-interactive. Setting it to `true` makes it active again.

Example Code

```
// Disable a TextBox
txtUsername.Enabled = false;

// Enable a Button
btnSubmit.Enabled = true;


// Toggle based on CheckBox
if (chkAgree.Checked)
{
 btnSubmit.Enabled = true;
}
else
{
 btnSubmit.Enabled = false;
}
```



Advanced Controls and Professional UI

Now that you understand the basics, let's explore the controls that make desktop applications professional and user-friendly. We'll cover essential controls like checkboxes and combo boxes, then move into menus, dialogs, and tooltips—the features that users expect from polished desktop software.

Essential Windows Form Controls

Button

Triggers actions when clicked—the most common interactive control

Label

Displays static text—perfect for field labels and instructions

TextBox

Accepts user text input—single-line or multi-line

CheckBox

True/false selections—users can check multiple options

RadioButton

Mutually exclusive choices—only one can be selected in a group

ComboBox

Dropdown list of options—combines text input with selection

ListBox

Scrollable list where users can select one or multiple items

PictureBox

Displays images—supports various formats and sizing modes

CheckBox vs. RadioButton

CheckBox: Multiple Choices

Use checkboxes when users can select zero, one, or multiple options. Each checkbox is independent.

Example: "Select your interests: ☑️ Sports ☑️ Music ⍰ Reading"

```
if (chkSports.Checked)
{
   interests.Add("Sports");
}
if (chkMusic.Checked)
{
   interests.Add("Music");
}
```

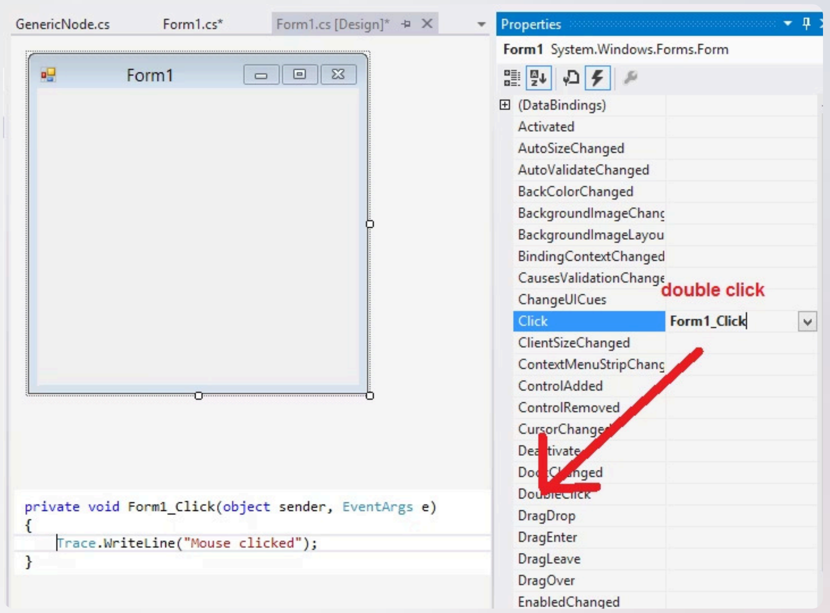
RadioButton: One Choice

Use radio buttons when users must select exactly one option from a group. Selecting one automatically deselects others in the same container.

Example: "Select size: ◉ Small ○ Medium ○ Large"

```
if (rbSmall.Checked)
{
 size = "Small";
}
else if (rbMedium.Checked)
{
 size = "Medium";
}
```

ComboBox: Dropdown Selections



```
private void Form1_Click(object sender, EventArgs e)
{
    Trace.WriteLine("Mouse clicked");
}
```

A ComboBox provides a dropdown list of options, saving space on your form. Users click to expand the list and select an item. ComboBoxes are ideal when you have many options but limited screen space.

Adding items: You can add items at design time through the Properties window (Items collection) or programmatically in code:

```
// Add items in code
comboBox1.Items.Add("Option 1");
comboBox1.Items.Add("Option 2");
comboBox1.Items.Add("Option 3");

// Get selected item
string selected = comboBox1.SelectedItem.ToString();

// Set default selection
comboBox1.SelectedIndex = 0; // Select first item
```

ListBox: Displaying Multiple Items

A ListBox displays a scrollable list of items. Unlike ComboBox, all items are visible (within the box's height). ListBoxes support single or multiple selection modes.

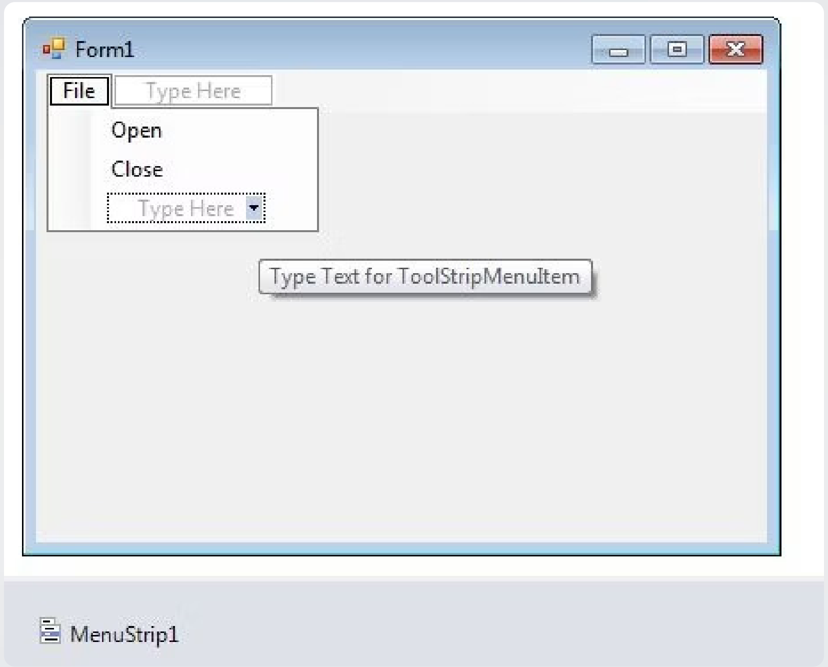
Adding Items

```
// Add items
listBox1.Items.Add("Item 1");
listBox1.Items.Add("Item 2");

// Add from array
string[] items = {"A", "B", "C"};
listBox1.Items.AddRange(items);
```

Getting Selected Items

```
// Single selection
string selected =
  listBox1.SelectedItem.ToString();

// Multiple selections
foreach (var item in
         listBox1.SelectedItems)
{
    Console.WriteLine(item);
}
```

Set `SelectionMode` property to `MultiSimple` or `MultiExtended` to allow multiple selections.



MENUS

Adding Professional Menus with MenuStrip

The MenuStrip control adds a professional menu bar to the top of your form—just like File, Edit, View menus in most applications. It's the standard way to organize commands and options in desktop applications.

Drag a MenuStrip from the Toolbox onto your form. It appears at the top, and you can type directly into it to create menu items. Click "Type Here" to add top-level menus (File, Edit, etc.), then click inside to add submenu items.

Building Your Menu Structure

Top-Level Menus

File, Edit, View, Help—the main categories visible in the menu bar

Menu Items

New, Open, Save—the actual commands under each menu category

Separators

Type "-" to add a horizontal line that groups related items visually

Submenus

Items can have their own submenus—click the arrow to expand more options

Each menu item is a ToolStripMenuItem. Double-click any item to create its Click event handler, just like a button.

Menu Features: Shortcuts and Icons

Keyboard Shortcuts

Professional apps provide keyboard shortcuts for common actions (Ctrl+S for Save, Ctrl+O for Open). Set the ShortcutKeys property on any menu item to assign a keyboard combination.

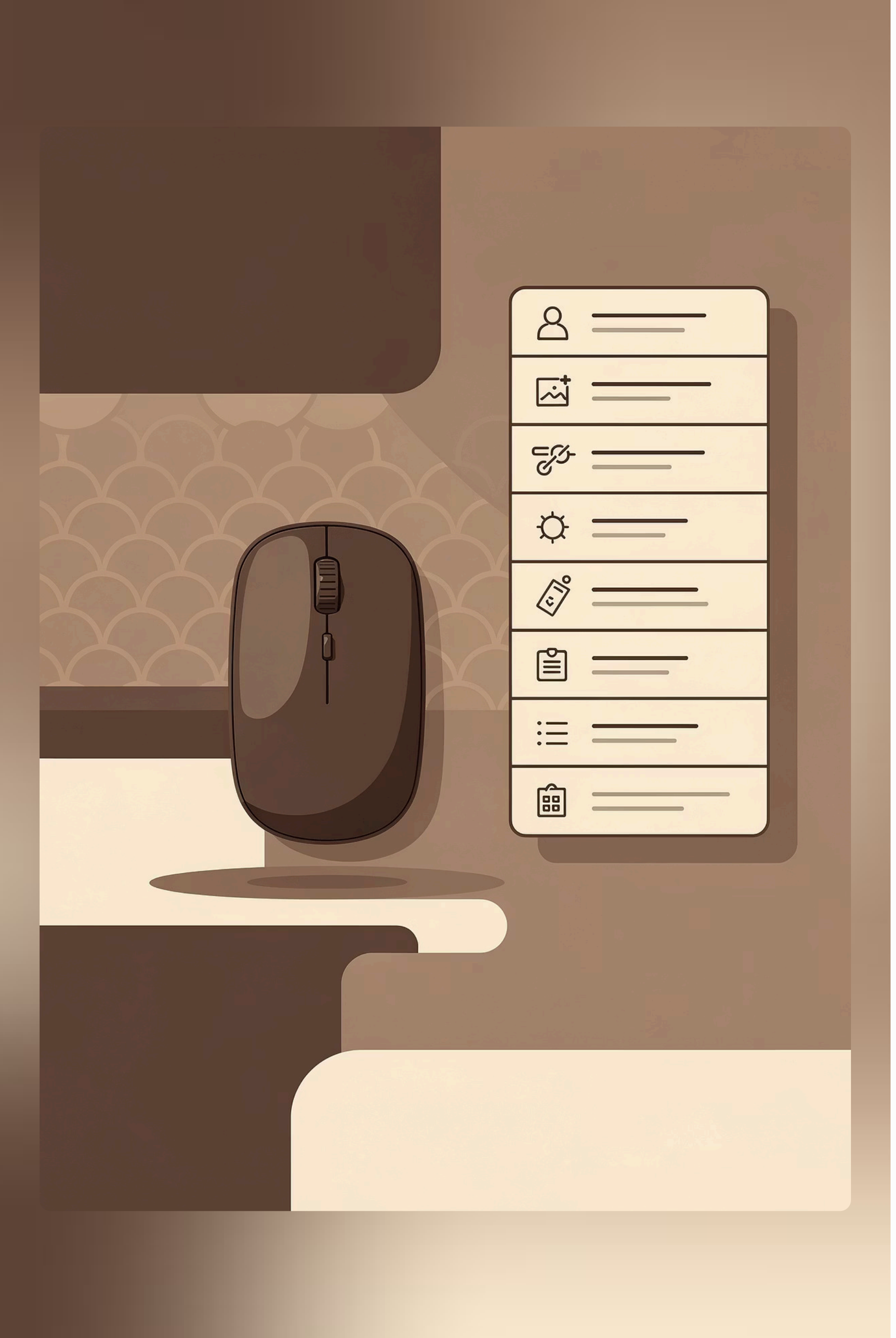
Users can then trigger the command without opening the menu—faster and more convenient.

Adding Icons

Menu items can display small icons (16×16 pixels) for visual recognition. Set the Image property to add an icon from your resources or a file.

Icons make menus more scannable and help users find commands quickly.

```
// Example: Exit menu item
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
 Application.Exit(); // Close the application
}
```



Context Menus: Right-Click Functionality

A ContextMenuStrip is a popup menu that appears when users right-click a control. It's perfect for providing quick access to relevant actions—like "Copy," "Paste," "Delete" when right-clicking text.

Add a ContextMenuStrip control to your form, design its menu items, then assign it to any control's ContextMenuStrip property. Now when users right-click that control, your custom menu appears.

```
// Example: Copy text to clipboard
private void copyToolStripMenuItem_Click(object sender, EventArgs e)
{
 Clipboard.SetText(textBox1.SelectedText);
}
```

DIALOGS

Dialog Boxes: Interacting Beyond Your Form

Dialog boxes are special windows that appear temporarily to gather information or display messages. Windows Forms includes several built-in dialogs for common tasks—opening files, choosing colors, selecting fonts, and showing messages.

Why dialogs matter: They provide a consistent, familiar experience. Users already know how to use a File Open dialog because every Windows application uses the same one. You don't have to recreate these interfaces from scratch.

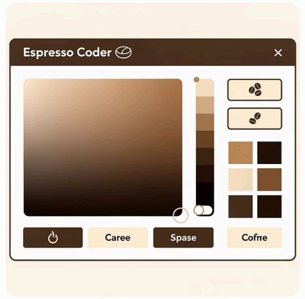
Common Dialog Controls

OpenFileDialog

Let users browse and select files to open—returns the file path

SaveFileDialog

Let users choose where to save a file and specify its name

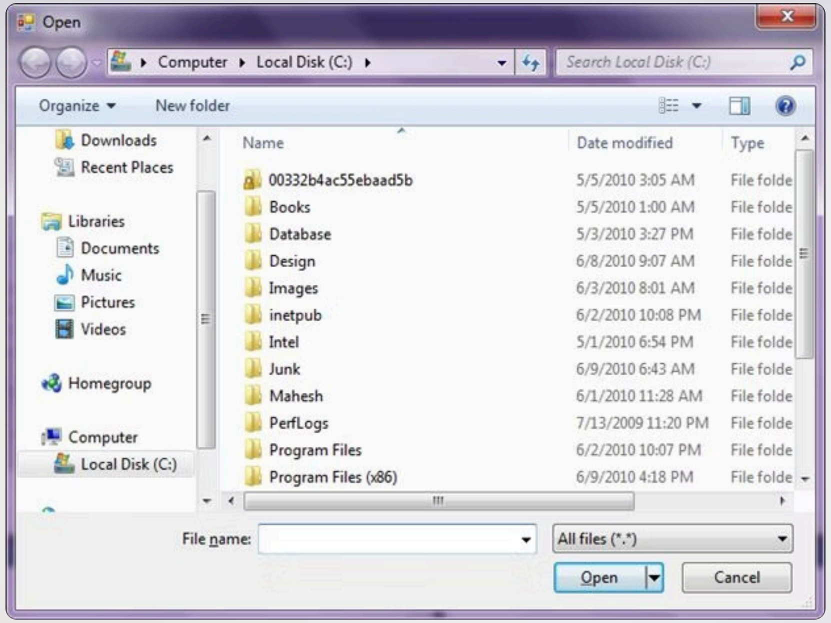


ColorDialog

Display a color picker—users select a color and you get the RGB value

FontDialog

Let users choose font family, size, and style—perfect for text editors



Name	Date modified	Type
00332b4ac55ebaad5b	5/5/2010 3:05 AM	File folde
Books	5/5/2010 1:00 AM	File folde
Database	5/3/2010 3:27 PM	File folde
Design	6/8/2010 9:07 AM	File folde
Images	6/3/2010 8:01 AM	File folde
inetpub	6/2/2010 10:08 PM	File folde
Intel	5/1/2010 6:54 PM	File folde
Junk	6/9/2010 6:43 AM	File folde
Mahesh	6/1/2010 11:28 AM	File folde
PerfLogs	7/13/2009 11:20 PM	File folde
Program Files	6/2/2010 10:07 PM	File folde
Program Files (x86)	6/9/2010 4:18 PM	File folde

File name: All files (*.*)

Using OpenFileDialog

The OpenFileDialog lets users browse their computer to select a file. It returns the full file path, which you can then use to read the file. This is essential for any app that works with user files—text editors, image viewers, data processors, etc.

```
private void btnOpen_Click(object sender, EventArgs e)
{
    OpenFileDialog openDialog = new OpenFileDialog();
    openDialog.Filter = "Text Files|*.txt|All Files|*.*";
    openDialog.Title = "Select a File";

    if (openDialog.ShowDialog() == DialogResult.OK)
    {
        string filePath = openDialog.FileName;
        string content = File.ReadAllText(filePath);
        textBox1.Text = content;
    }
}
```

The Filter property controls which file types appear. The format is "Description|Pattern" (e.g., "Text Files|*.txt").

MessageBox: Displaying Messages

MessageBox is the simplest way to show information, warnings, errors, or ask questions. It's a static method, so you don't need to create an instance—just call MessageBox.Show().

The return value tells you which button the user clicked (OK, Cancel, Yes, No, etc.), allowing you to branch your code accordingly.

Basic Usage

```
// Simple message
MessageBox.Show("File saved!");

// With title
MessageBox.Show("Success!",
 "File Operation");

// With buttons
DialogResult result =
 MessageBox.Show(
 "Delete this file?",
 "Confirm",
 MessageBoxButtons.YesNo);

if (result == DialogResult.Yes)
{
 // Delete the file
}
```

FontDialog: Let Users Choose Fonts

The FontDialog provides a standard interface for selecting font properties. When users make their selection, you can apply the chosen font to your controls. This is commonly used in text editors and word processors.

```
private void btnChangeFont_Click(object sender, EventArgs e)
{
    FontDialog fontDialog = new FontDialog();
    fontDialog.Font = textBox1.Font; // Set current font as default

    if (fontDialog.ShowDialog() == DialogResult.OK)
    {
        textBox1.Font = fontDialog.Font; // Apply selected font
    }
}
```

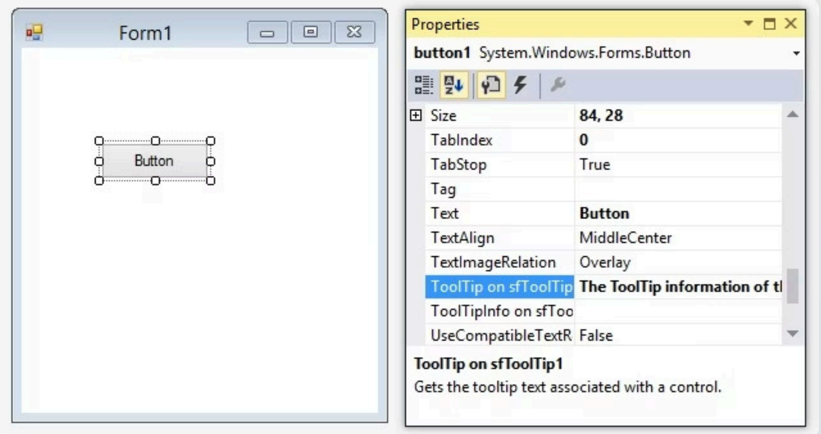
Real-world use case: This is perfect for Practical #2 in your syllabus—implementing font size increase/decrease functionality. You can use FontDialog or programmatically adjust the font size with buttons.

USER EXPERIENCE

ToolTips: Helpful Hints for Users

A ToolTip is a small popup hint that appears when users hover over a control. It provides helpful information without cluttering your interface. ToolTips are excellent for explaining button functions, providing input hints, or clarifying abbreviations.

Add a ToolTip component to your form (it appears in the component tray below the designer). Then set the "ToolTip on toolTip1" property for any control you want to add a hint to. That's it—hover over the control and your hint appears!



button1 System.Windows.Forms.Button

Size	84, 28
TabIndex	0
TabStop	True
Tag	
Text	Button
TextAlign	MiddleCenter
TextImageRelation	Overlay
ToolTip on sfToolTip	The ToolTip information of tl
ToolTipInfo on sfToo	
UseCompatibleTextR	False

ToolTip on sfToolTip1
Gets the tooltip text associated with a control.

Adding ToolTips: Two Approaches

Design-Time Method

Drag a ToolTip component onto your form. Select any control, find its "ToolTip on toolTip1" property in the Properties window, and type your hint text.

This is the easiest approach for static tooltips that don't change at runtime.

Code Method

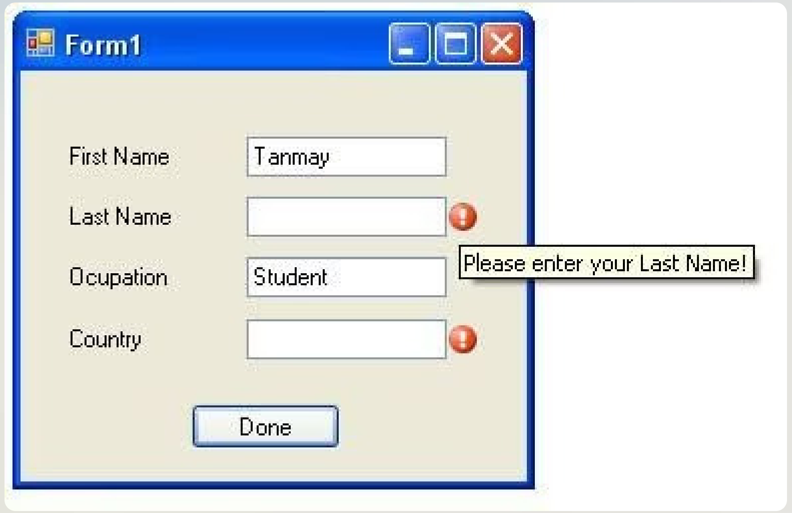
Set tooltips programmatically when you need dynamic hints that change based on application state.

```
// Create ToolTip
ToolTip toolTip1 = new ToolTip();

// Set tooltips
toolTip1.SetToolTip(button1,
 "Click to submit form");

toolTip1.SetToolTip(textBox1,
 "Enter your username");

// Change tooltip at runtime
if (isLoggedIn)
{
 toolTip1.SetToolTip(button1,
 "Click to logout");
}
```

PRACTICAL APPLICATION

Putting It All Together

Now you have all the tools to build professional desktop applications. Let's review how the concepts connect to your practical assignments and real-world development scenarios.



Tanmay

Please enter your Last Name!

Student

Practical 1: Enable/Disable Controls

Use the Enabled property and event handlers to toggle TextBox states based on CheckBox selections or button clicks

Practical 2: Font Size Control

Implement buttons that increase/decrease font size, or use FontDialog for a complete font selection interface

Practical 4: Name Input Form

Create a form with TextBox for name input, Button for submission, and Label to display greeting message

Practical 9: Temperature Converter

Build a GUI with TextBoxes for input/output, RadioButtons for unit selection, and calculation logic in button Click event

Key Takeaways

Windows Forms = Desktop Development

Windows Forms provides everything you need to build professional Windows desktop applications with rich graphical interfaces

Event-Driven Architecture

Applications respond to user actions through events—clicks, key presses, text changes. Your code waits and reacts rather than running sequentially

Properties Control Appearance

Every form and control has properties that determine how it looks and behaves. Set them at design time or programmatically at runtime

Rich Control Library

From basic buttons and text boxes to menus, dialogs, and tooltips—Windows Forms provides all the UI elements users expect from desktop software

Professional Features Are Built-In

Leverage standard dialogs (File Open, Save, Color, Font) for consistent user experience. Add menus and tooltips to enhance usability

What's Next

Coming Up: Visual Inheritance

In Unit 5, you'll learn how to create reusable form templates through visual inheritance. Build a base form with common elements, then inherit from it to create consistent, maintainable applications with less code duplication.

This powerful technique lets you define layouts, controls, and behavior once and reuse them across multiple forms—essential for large applications.

Practice Makes Perfect

Before moving forward, make sure you can:

- Create a new Windows Forms application
- Add and configure controls
- Handle basic events (Click, Load, TextChanged)
- Implement menus and dialogs
- Build at least one complete mini-application

Experiment with different layouts and controls. The more you practice, the more intuitive Windows Forms development becomes.