.NET Windows Forms

Microsoft .NET Windows Forms

WinForms

- **Windows Forms** (WinForms) is a graphical class library included as a part of Microsoft .NET Framework or Mono Framework
  - providing a platform to write rich client applications for desktop, laptop, and tablet PCs.
  - Windows Forms is built on the existing Windows API and some controls merely wrap underlying Windows components

Microsoft .NET
Windows Forms

WinForms

- WinForms released on .NET framework 1.0 and was updated over the frameworks.
- In .NET Framework 2.0, Windows Forms gained richer layout controls, Office 2003 style controls, multithreading component, richer design-time and data binding support as well as ClickOnce for web-based deployment.
- WinForms has been feature-complete since .NET 2.0 (VS2005), and most of it’s develop team switched to WPF since .NET 3.0, Only critical operating system compatibility and security fixes have been released since .NET 2.0

WinForms compared to WPF

Topic	Windows Forms	WPF
Introduced in	2001 (.NET Framework 1.0)	2006 (.NET Framework 3.0)
System Requirements	>= Windows 98	>= Windows XP
Technology	Layer on top of standard Windows controls (Win32)	WPF is built from scratch with more than 80% managed code
Rendering	Pixel graphic UI based on GDI+	Vector graphic UI based on DirectX
Customization	•Limited to the behavior and look & feel of Windows Forms controls. •Extending the behavior needs often to use Win32 API calls. •3rd party libraries provide more flexible controls.	•Very flexible and powerful to adapt and restyle UI controls. •Adaption of control behavior can be done with .NET API.
Styling / Theming	Styling is quite limited with the standard controls.	Styling support is very powerful. But it's also complex.
UI Definition	Procedural: C#/VB code (generated by the GUI designer)	Declarative: XAML (XML dialect); More control over the code created by the designer

WinForms compared to WPF

Topic	Windows Forms	WPF
Supporting separation of concerns, databinding.	•Provides limited databinding	•Powerful databinding allows a cleaner separation of UI and domain logic •Command pattern •Model-View-ViewModel Pattern
Long term support	Yes. Windows Forms is part of the .NET Framework.	Yes. WPF is part of the .NET Framework.
Mature	Yes. This technology is in maintenance mode. Innovative improvements are not expected anymore.	Yes. This technology is in maintenance mode. Innovative improvements are not expected anymore.
Tool support	Visual Studio Windows Forms designer	Visual Studio XAML designer; Expression Blend
Known drawbacks	•Issues with high resolution displays because pixel graphic rendering does not scale well. •Generated code by VS Designer is hard to merge when merge conflicts exist.	•The learning curve is quite steep. •Slower than Windows Forms because of a more complex UI stack.

WinForms compared to WPF

WinForms pros:

- Excellent for rapid development of simple Windows forms
- Supported on a wider range of Windows operating systems when compared to WPF
- Mature product with large and active userbase

WinForms cons:

- No support for vector graphics or resolution independence
- Limited binding system
- Advanced visual effects are very difficult to achieve
- Pixel graphic UI based on GDI+

WPF pros:

- Vector scaling, resolution independence
- Advanced visual effects and animations are made simple
- Powerful data binding system makes keeping your UI in sync with your model simple
- Supports the MVVM pattern and powerful databinding.
- Hardware accelerated based on DirectX
- Follows the direction Microsoft are heading

WPF cons:

- Newer technology with a smaller userbase
- Implementation feels incomplete in some places
- Supported on fewer Windows operating systems
- Can be slower to develop simpler applications

Microsoft .NET Windows Forms

Event handler

- An event handler is a method that is bound to an event. When the event is raised, the code within the event handler is executed.
- Each event handler provides two parameters that allow you to handle the event properly.
- The following example shows an event handler for a Button control's Click event:

```
private void button1_Click(object sender, System.EventArgs e)
{


}
```

Event handler

- The first parameter (sender) provides a reference to the object that raised the event.
- The second parameter (e) in the example above, passes an object specific to the event that is being handled. By referencing the object's properties (and, sometimes, its methods), you can obtain information such as the location of the mouse for mouse events or data being transferred in drag-and-drop events.
- You can use the same event handler to handle more than one event (e.g: MouseUp and MouseDown events) and you can also use the same event handler for different controls (e.g: Radio buttons).

Invoke

Microsoft .NET Windows Forms

- When using event handlers like button click, the thread that runs the event is the main UI thread, means that until the event is done the UI is hanged.
- To prevent this problem we use different thread to handle using Task.Run() or other threading methods. But using this different thread we can not access and change elements directly in our form because different thread is managing them.
- To change elements in different thread we can use Invoke from a different thread safely.
- To check if invoke is required we use Contol.InvokeRequired that returns true if the control is blocked by another thread.

Microsoft .NET Windows Forms

Invoke

For example if we want to change button1 text when we click on it even before the event handler finished its block or by another thread we can use:

```
If (button1.InvokeRequired){
        Invoke(new Action(() =>
        {
                button1.Text = “Clicked”;
        }));
}
else button1.Text = “Clicked”;
```

User Custom controls

- Windows Forms controls are reusable components that encapsulate user interface functionality and are used in client-side Windows-based applications.
- Windows Forms provide many ready-to-use controls, but it also provides the infrastructure for developing your own controls.
- You can combine existing controls, extend existing controls, or author your own custom controls.
- To create custom control using visual studio, on the solution explorer choose add item, go to Windows Forms item list and choose Custom Control. This will create a file with partial Class that extends Control. You can change it to extend an existing control and just change some properties or create a new control and define all the properties.

Microsoft .NET Windows Forms

Best practices with WinForms

1. avoid putting too many controls onto a single form.
   - Try to minimize the amount of controls on the same form, use views and new windows to break your main form to logical areas.
   - Doing this not only keeps your classes from getting unmanageably large, but it also makes tasks like setting up resizing and tab order much more straightforward.
   - It also allows you to easily disable whole sections of your user interface in one go where necessary.
2. Keep non UI code out of code behind
   - Do not inset code for accessing database , network , file system or any “non UI” code into your form’s code behind. Separate those functions to different classes with single responsibility.

Microsoft .NET Windows Forms

Best practices with WinForms

3. Create passive views with interfaces and use presenters to control the views (Model-View-Presenter pattern):
   - Create all your views passive without UI business logic.
   - Create a Presenter classes that will be called from the passive views, using event handlers. The presenters will do the UI business logic and contacting the Models.
   - Never use the Views code behind to manipulate the Models.
   - By ensuring that your view implementations are as simple as possible, you will maximize your chances of being able to migrate to an alternative UI framework.

Microsoft .NET Windows Forms

Best practices with WinForms

4. Use the Command pattern:
   - If your application contains a toolbar with a large number of buttons for the user to click, the Command pattern may be a very good fit. The command pattern dictates that you create a class for each command.
5. Use the Event Aggregator pattern
   - This is a pattern where the raiser of an event and the handler of an event do not need to be coupled to each other at all. When an "event" happens in your code that needs to be handled elsewhere, simply post a message to the event aggregator.
   - An example might be that you send a "help requested" message with details of where the user currently is in the UI. Another service then handles that message and ensures the correct page in the help documentation is launched in a web browser.