Overview of the .NET Framework

The .NET Framework is a technology that supports building and running the next generation of apps and XML Web services. The .NET Framework is designed to fulfill the following objectives:

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of apps, such as Windows-based apps and Web-based apps.
- To build all communication on industry standards to ensure that code based on the .NET Framework integrates with any other code.

The .NET Framework consists of the common language runtime (CLR) and the .NET Framework class library. The common language runtime is the foundation of the .NET Framework. Think of the runtime as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict type safety and other forms of code accuracy that promote security and robustness. In fact, the concept of code management is a fundamental principle of the runtime. Code that targets the runtime is known as managed code, while code that doesn't target the runtime is known as unmanaged code. The class library is a comprehensive, object-oriented collection of reusable types that you use to develop apps ranging from traditional command-line or graphical user interface (GUI) apps to apps based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services.

The .NET Framework can be hosted by unmanaged components that load the common language runtime into their processes and initiate the execution of managed code, thereby creating a software environment that exploits both managed and unmanaged features. The .NET Framework not only provides several runtime hosts but also supports the development of third-party runtime hosts.

For example, ASP.NET hosts the runtime to provide a scalable, server-side environment for managed code. ASP.NET works directly with the runtime to enable ASP.NET apps and XML Web services, both of which are discussed later in this topic.

Internet Explorer is an example of an unmanaged app that hosts the runtime (in the form of a MIME type extension). Using Internet Explorer to host the runtime enables you to embed

managed components or Windows Forms controls in HTML documents. Hosting the runtime in this way makes managed mobile code possible, but with significant improvements that only managed code offers, such as semi-trusted execution and isolated file storage.

The following illustration shows the relationship of the common language runtime and the class library to your apps and to the overall system. The illustration also shows how managed code operates within a larger architecture.

Features of the common language runtime

The common language runtime manages memory, thread execution, code execution, code safety verification, compilation, and other system services. These features are intrinsic to the managed code that runs on the common language runtime.

Regarding security, managed components are awarded varying degrees of trust, depending on a number of factors that include their origin (such as the Internet, enterprise network, or local computer). This means that a managed component might or might not be able to perform file-access operations, registry-access operations, or other sensitive functions, even if it's used in the same active app.

The runtime also enforces code robustness by implementing a strict type-and-code-verification infrastructure called the common type system (CTS). The CTS ensures that all managed code is self-describing. The various Microsoft and third-party language compilers generate managed code that conforms to the CTS. This means that managed code can consume other managed types and instances, while strictly enforcing type fidelity and type safety.

In addition, the managed environment of the runtime eliminates many common software issues. For example, the runtime automatically handles object layout and manages references to objects, releasing them when they are no longer being used. This automatic memory management resolves the two most common app errors, memory leaks and invalid memory references.

The runtime also accelerates developer productivity. For example, programmers write apps in their development language of choice yet take full advantage of the runtime, the class library, and components written in other languages by other developers. Any compiler vendor who chooses to target the runtime can do so. Language compilers that target the .NET Framework make the features of the .NET Framework available to existing code written in that language, greatly easing the migration process for existing apps.

While the runtime is designed for the software of the future, it also supports software of today and yesterday. Interoperability between managed and unmanaged code enables developers to continue to use necessary COM components and DLLs.

The runtime is designed to enhance performance. Although the common language runtime provides many standard runtime services, managed code is never interpreted. A feature called just-in-time (JIT) compiling enables all managed code to run in the native machine language of the system on which it's executing. Meanwhile, the memory manager removes the possibilities of fragmented memory and increases memory locality-of-reference to further increase performance.

Finally, the runtime can be hosted by high-performance, server-side apps, such as Microsoft SQL Server and Internet Information Services (IIS). This infrastructure enables you to use managed code to write your business logic, while still enjoying the superior performance of the industry's best enterprise servers that support runtime hosting.

.NET Framework class library

The .NET Framework class library is a collection of reusable types that tightly integrate with the common language runtime. The class library is object oriented, providing types from which your own managed code derives functionality. This not only makes the .NET Framework types easy to use but also reduces the time associated with learning new features of the .NET Framework. In addition, third-party components integrate seamlessly with classes in the .NET Framework.

For example, the .NET Framework collection classes implement a set of interfaces for developing your own collection classes. Your collection classes blend seamlessly with the classes in the .NET Framework.

As you would expect from an object-oriented class library, the .NET Framework types enable you to accomplish a range of common programming tasks, including tasks such as string management, data collection, database connectivity, and file access. In addition to these common tasks, the class library includes types that support a variety of specialized development scenarios. Use the .NET Framework to develop the following types of apps and services:

1. Console apps. See Building Console Applications.
2. Windows GUI apps (Windows Forms). See Windows Forms.
3. Windows Presentation Foundation (WPF) apps. See Windows Presentation Foundation.
4. ASP.NET apps. See Web Applications with ASP.NET.
5. Windows services. See Introduction to Windows Service Applications.
6. Service-oriented apps using Windows Communication Foundation (WCF). See Service-Oriented Applications with WCF.
7. Workflow-enabled apps using Windows Workflow Foundation (WF). See Windows Workflow Foundation.

The Windows Forms classes are a comprehensive set of reusable types that vastly simplify Windows GUI development. If you write an ASP.NET Web Form app, you can use the Web

Forms classes.

WPF overview

Windows Presentation Foundation (WPF) lets you create desktop client applications for Windows with visually stunning user experiences.

The core of WPF is a resolution-independent and vector-based rendering engine that is built to take advantage of modern graphics hardware. WPF extends the core with a comprehensive set of application-development features that include Extensible Application Markup Language (XAML), controls, data binding, layout, 2D and 3D graphics, animation, styles, templates, documents, media, text, and typography. WPF is included in the .NET Framework, so you can build applications that incorporate other elements of the .NET Framework class library.

This overview is intended for newcomers and covers the key capabilities and concepts of WPF.

Program with WPF

WPF exists as a subset of .NET Framework types that are for the most part located in the System.Windows namespace. If you have previously built applications with .NET Framework using managed technologies like ASP.NET and Windows Forms, the fundamental WPF programming experience should be familiar; you instantiate classes, set properties, call methods, and handle events, all using your favorite .NET programming language, such as C# or Visual Basic.

WPF includes additional programming constructs that enhance properties and events: dependency properties and routed events.

Markup and code-behind

WPF lets you develop an application using both markup and code-behind, an experience with which ASP.NET developers should be familiar. You generally use XAML markup to implement the appearance of an application while using managed programming languages (code-behind) to implement its behavior. This separation of appearance and behavior has the following benefits:

- Development and maintenance costs are reduced because appearance-specific markup is not tightly coupled with behavior-specific code.
- Development is more efficient because designers can implement an application's appearance simultaneously with developers who are implementing the application's behavior.
- Globalization and localization for WPF applications is simplified.

Markup

XAML is an XML-based markup language that implements an application's appearance declaratively. You typically use it to create windows, dialog boxes, pages, and user controls, and to fill them with controls, shapes, and graphics.

The following example uses XAML to implement the appearance of a window that contains a single button.

XAML 复制

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    Title="Window with Button"
    Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button">Click Me!</Button>

</Window>
```

Specifically, this XAML defines a window and a button by using the Window and Button elements, respectively. Each element is configured with attributes, such as the Window element's Title attribute to specify the window's title-bar text. At run time, WPF converts the elements and attributes that are defined in markup to instances of WPF classes. For example, the Window element is converted to an instance of the Window class whose Title property is the value of the Title attribute.

Code-behind

The main behavior of an application is to implement the functionality that responds to user interactions, including handling events (for example, clicking a menu, tool bar, or button) and calling business logic and data access logic in response. In WPF, this behavior is implemented in code that is associated with markup. This type of code is known as code-behind. The following example shows the updated markup from the previous example and the code-behind.

XAML 复制

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.AWindow"
    Title="Window with Button"
    Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button" Click="button_Click">Click Me!</Button>

</Window>
```

In this example, the code-behind implements a class that derives from the Window class. The x:Class attribute is used to associate the markup with the code-behind class. InitializeComponent is called from the code-behind class's constructor to merge the UI that is defined in markup with the code-behind class. (InitializeComponent is generated for you

when your application is built, which is why you don't need to implement it manually.) The combination of x:Class and InitializeComponent ensure that your implementation is correctly initialized whenever it is created. The code-behind class also implements an event handler for the button's Click event. When the button is clicked, the event handler shows a message box by calling the System.Windows.MessageBox.Show method.

Controls

The user experiences that are delivered by the application model are constructed controls. In WPF, control is an umbrella term that applies to a category of WPF classes that are hosted in either a window or a page, have a user interface, and implement some behavior.

Input and commands

Controls most often detect and respond to user input. The WPF input system uses both direct and routed events to support text input, focus management, and mouse positioning.

Applications often have complex input requirements. WPF provides a command system that separates user-input actions from the code that responds to those actions.

Layout

When you create a user interface, you arrange your controls by location and size to form a layout. A key requirement of any layout is to adapt to changes in window size and display settings. Rather than forcing you to write the code to adapt a layout in these circumstances, WPF provides a first-class, extensible layout system for you.

The cornerstone of the layout system is relative positioning, which increases the ability to adapt to changing window and display conditions. In addition, the layout system manages the negotiation between controls to determine the layout. The negotiation is a two-step process: first, a control tells its parent what location and size it requires; second, the parent tells the control what space it can have.

The layout system is exposed to child controls through base WPF classes. For common layouts such as grids, stacking, and docking, WPF includes several layout controls:

- Canvas: Child controls provide their own layout.
- DockPanel: Child controls are aligned to the edges of the panel.
- Grid: Child controls are positioned by rows and columns.
- StackPanel: Child controls are stacked either vertically or horizontally.
- VirtualizingStackPanel: Child controls are virtualized and arranged on a single line that is either horizontally or vertically oriented.
- WrapPanel: Child controls are positioned in left-to-right order and wrapped to the next line when there are more controls on the current line than space allows.

Data binding

Most applications are created to provide users with the means to view and edit data. For WPF applications, the work of storing and accessing data is already provided for by technologies such as SQL Server and ADO .NET. After the data is accessed and loaded into an application's managed objects, the hard work for WPF applications begins. Essentially, this involves two things:

1. Copying the data from the managed objects into controls, where the data can be displayed and edited.
2. Ensuring that changes made to data by using controls are copied back to the managed objects.

To simplify application development, WPF provides a data binding engine to automatically perform these steps. The core unit of the data binding engine is the Binding class, whose job is to bind a control (the binding target) to a data object (the binding source). This relationship is illustrated by the following figure:

The WPF data binding engine provides additional support that includes validation, sorting, filtering, and grouping. Furthermore, data binding supports the use of data templates to create custom user interface for bound data when the user interface displayed by the standard WPF controls is not appropriate.

Graphics

WPF introduces an extensive, scalable, and flexible set of graphics features that have the following benefits:

- Resolution-independent and device-independent graphics. The basic unit of measurement in the WPF graphics system is the device-independent pixel, which is 1/96th of an inch, regardless of actual screen resolution, and provides the foundation for resolution-independent and device-independent rendering. Each device-independent pixel automatically scales to match the dots-per-inch (dpi) setting of the system it renders on.
- Improved precision. The WPF coordinate system is measured with double-precision floating-point numbers rather than single-precision. Transformations and opacity values are also expressed as double-precision. WPF also supports a wide color gamut (scRGB) and provides integrated support for managing inputs from different color spaces.
- Advanced graphics and animation support. WPF simplifies graphics programming by managing animation scenes for you; there is no need to worry about scene processing, rendering loops, and bilinear interpolation. Additionally, WPF provides hit-testing support and full alpha-compositing support.

- Hardware acceleration. The WPF graphics system takes advantage of graphics hardware to minimize CPU usage.

WPF Data Binding with LINQ to XML

Overview

This topic introduces the dynamic data binding features in the System.Xml.Linq namespace. These features can be used as a data source for user interface (UI) elements in Windows Presentation Foundation (WPF) apps. This scenario relies upon special dynamic properties of System.Xml.Linq.XAttribute and System.Xml.Linq.XElement.

XAML and LINQ to XML

The Extensible Application Markup Language (XAML) is an XML dialect created by Microsoft to support .NET Framework 3.0 technologies. It is used in WPF to represent user interface elements and related features, such as events and data binding. In Windows Workflow Foundation, XAML is used to represent program structure, such as program control (workflows). XAML enables the declarative aspects of a technology to be separated from the related procedural code that defines the more individualized behavior of a program.

There are two broad ways that XAML and LINQ to XML can interact:

- Because XAML files are well-formed XML, they can be queried and manipulated through XML technologies such as LINQ to XML.
- Because LINQ to XML queries represent a source of data, these queries can be used as a data source for data binding for WPF UI elements.

This documentation describes the second scenario.

Data Binding in the Windows Presentation Foundation

WPF data binding enables a UI element to associate one of its properties with a data source. A simple example of this is a Label whose text presents the value of a public property in a user-defined object. WPF data binding relies on the following components:

A dependency property is a concept specific to WPF that represent a dynamically computed property of a UI element. For example, dependency properties often have default values or values that are provided by a parent element. These special properties are backed by instances of the DependencyProperty class (and not fields as with standard properties). For more information, see Dependency Properties Overview.

Dynamic Data Binding in WPF

By default, data binding occurs only when the target UI element is initialized. This is called one-time binding. For most purposes, this is insufficient; typically, a data-binding solution requires that the changes be dynamically propagated at run time using one of the following:

- One-way binding causes the changes to one side to be propagated automatically. Most commonly, changes to the source are reflected in the target, but the reverse can sometimes be useful.
- In two-way binding, changes to the source are automatically propagated to the target, and changes to the target are automatically propagated to the source.

For one-way or two-way binding to occur, the source must implement a change notification mechanism, for example by implementing the INotifyPropertyChanged interface or by using a PropertyNameChanged pattern for each property supported.

Dynamic Properties in LINQ to XML Classes

Most LINQ to XML classes do not qualify as proper WPF dynamic data sources. Some of the most useful information is available only through methods, not properties, and properties in these classes do not implement change notifications. To support WPF data binding, LINQ to XML exposes a set of dynamic properties.

These dynamic properties are special run-time properties that duplicate the functionality of existing methods and properties in the XAttribute and XElement classes. They were added to these classes solely to enable them to act as dynamic data sources for WPF. To meet this need, all these dynamic properties implement change notifications.

Many of the standard public properties, found in the various classes in the System.Xml.Linq namespace, can be used for one-time data binding. However, remember that neither the source nor the target will be dynamically updated under this scheme.