**Курс:**
**Основы технологии программирования .NET**

# C# .NET

## WINDOWS PRESENTATION FOUNDATION

# Windows Presentation Foundation

- Windows Presentation Foundation (WPF) — графическая система отображения для Windows.

WPF спроектирована для .NET с использования аппаратного ускорения.

Она также представляет собой наиболее радикальное изменение в пользовательском интерфейсе Windows.

# Windows Presentation Foundation

Стандартное приложение Windows для создания пользовательского интерфейса полагается на две основополагающие части операционной системы Windows:

- *User32 обеспечивает знакомый внешний вид и поведение таких элементов, как* окна, кнопки, текстовые поля и т.п.;
- *GDI/GDI+ предоставляет поддержку рисования фигур, текста и изображений за* счет дополнительного усложнения.

# Windows Presentation Foundation

С годами обе технологии совершенствовались, и API-интерфейсы, используемые разработчиками для взаимодействия с ними, значительно изменились.

Но как бы вы ни разрабатывали приложение — с помощью .NET и Windows Forms “за кулисами” работают одни и те же части операционной системы Windows. Новые каркасы просто представляют лучшие оболочки для взаимодействия с User32 и GDI/GDI+

# DirectX: графический механизм

- DirectX – это набор компонентов в ОС Windows, который позволяет программному обеспечению, в первую очередь компьютерным играм, напрямую взаимодействовать с видео- и аудиооборудованием. Игры, использующие DirectX, могут более эффективно использовать встроенные в ваше оборудование функции акселерации мультимедиа, благодаря чему повышается производительность выполнения мультимедийных задач.
- Инструмент DxDiag представляет подробную информацию о компонентах и драйверах DirectX, которые установлены в вашей системе и доступны к использованию.

# DirectX: графический механизм

- Лежащая в основе WPF графическая технология — это DirectX.

Приложения WPF используют DirectX независимо от создаваемого типа пользовательского интерфейса. Это значит, что создаете ли вы сложную трехмерную графику, либо просто рисуете кнопки и простой текст — вся работа по рисованию проходит через конвейер DirectX.

# Аппаратное ускорение и WPF

Видеокарты различаются между собой в их поддержке специализированных средств визуализации и оптимизации. При программировании с DirectX это является существенной проблемой.

С применением WPF она не так сильно проявляется, поскольку WPF обладает способностью выполнять всю работу с использованием программных вычислений вместо того, чтобы полагаться на встроенную поддержку видеокарты.

# Аппаратное ускорение и WPF

Наличие мощной видеокарты не дает абсолютной гарантии, что вы получите максимальную, с аппаратной поддержкой производительность на WPF.

WPF не может обеспечить аппаратного ускорения на видеокартах, если используются устаревшие драйверы.

WPF достаточно интеллектуален, чтобы по возможности использовать аппаратную оптимизацию, но в случае неудачи все будет обработано программно. Поэтому если вы запустите приложение WPF на компьютере с унаследованной видеокартой, интерфейс будет выглядеть так, как вы его разработали

# Аппаратное ускорение и WPF

## Уровни WPF

Видеокарты значительно различаются между собой. Когда WPF обращается к видеокарте, то учитывает много факторов, на основе всех этих деталей определяется значение *уровня визуализации WPF:*

- **Уровень визуализации 0.** Видеокарта не предоставляет никакого аппаратного ускорения. Это соответствует версии DirectX уровня ниже 7.0.
- **Уровень визуализации 1.** Видеокарта обеспечивает частичное аппаратное ускорение. Это соответствует уровню версии DirectX выше 7.0, но ниже 9.0.
- •**Уровень визуализации 2.** Все средства, которые могут быть ускорены аппаратно, будут ускорены. Это отвечает уровню версии DirectX от 9.0 и выше.

# WPF: высокоуровневый API

Наиболее существенные изменения, которые принес с собой WPF в мир программирования Windows:

- *Web-подобная модель компоновки.* пользовательский интерфейс, который может быть адаптирован для отображения высокодинамичного содержимого или разных языков
- *Богатая модель рисования.* Вместо рисования пикселей в WPF вы имеете дело с *примитивами — базовыми фигурами, блоками текста и прочими графическими* ингредиентами.
- *Богатая текстовая модель.* WPF предоставляет приложениям Windows возможность отображения богатого стилизованного текста в любом месте пользовательского интерфейса.

# WPF: высокоуровневый API

Наиболее существенные изменения, которые принес с собой WPF в мир программирования Windows:

- *Анимация как первоклассная программная концепция.* WPF анимация — неотъемлемая часть программного каркаса. Вы определяете анимацию декларативными дескрипторами, и WPF запускает ее в действие автоматически.
- *Поддержка аудио и видео.* WPF включает поддержку воспроизведения любого аудио- или видеофайла, поддерживаемого Windows Media Player.
- *Стили и шаблоны.* Стили позволяют стандартизовать форматирование и повторно использовать его по всему приложению. Шаблоны позволяют изменить способ отображения элементов, даже таких основополагающих, как кнопки.

# WPF: высокоуровневый API

Наиболее существенные изменения, которые принес с собой WPF в мир программирования Windows:

- *Команды.* вы можете определять прикладные команды в одном месте и привязывать их к множеству элементов управления.
- *Декларативный пользовательский интерфейс.* Содержимое каждого окна сериализуется в виде XML-дескрипторов в документе XAML.
- *Приложения на основе страниц.* Используя WPF, вы можете строить браузер-подобные приложения, которые позволяют перемещаться по коллекции страниц, оснащенной кнопками навигации вперед и назад. WPF автоматически обрабатывает все сложные детали, такие как хронология посещения страниц. Вы можетедаже развернуть ваш проект в виде браузерного приложения.

# *Единицы WPF*

Окно WPF и все элементы внутри него измеряются в *независимых от устройства единицах.*

*Одна такая единица определяется как 1/96 дюйма.*

WPF автоматически заботится о том, чтобы все имело правильный размер.

При проектировании пользовательского интерфейса WPF даже самые мелкие пиктограммы обычно реализованы в векторной графике. Векторная графика определена как набор фигур, каждая из которых может быть легко масштабирована до любого размера.

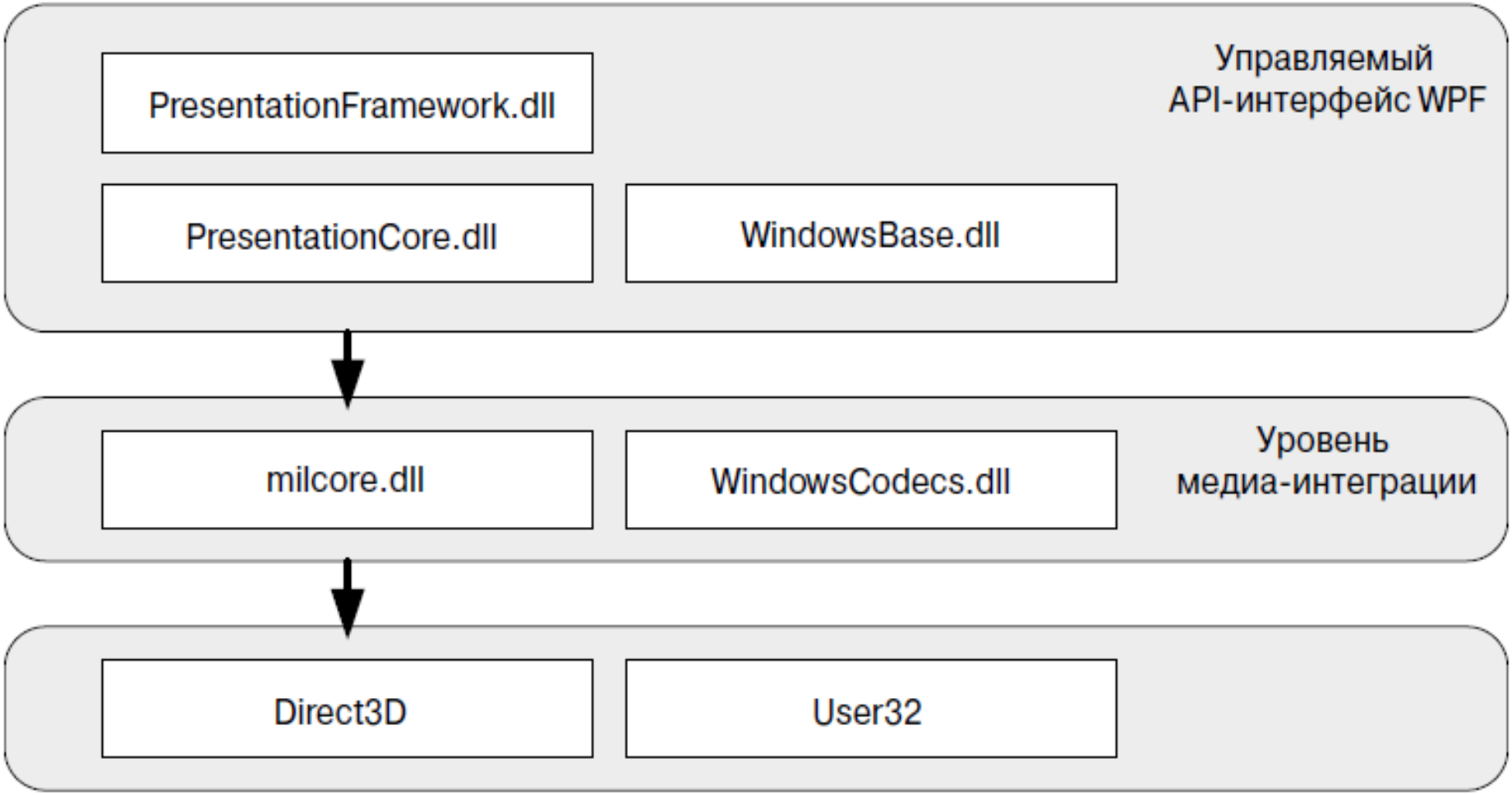
# Архитектура WPF

WPF использует многослойную архитектуру. На вершине находятся ваше приложение, взаимодействующее с высокоуровневым набором служб, полностью состоящих из управляемого кода C#.

Работа по трансляции объектов .NET в текстуры Direct3D и треугольники происходит “за кулисами”, с использованием низкоуровневого неуправляемого компонента по имени milcore.dll.

Компонент milcore.dll реализован в неуправляемом коде потому, что ему требуется тесная интеграция с Direct3D, и потому, что для него чрезвычайно важна производительность.

# Архитектура WPF



# Архитектура WPF

- *PresentationFramework.dll* содержит типы WPF верхнего уровня, включая те, что представляют окна, панели и прочие виды элементов управления. Также он реализует высокоуровневые программные абстракции, такие как стили. Большинство классов, которые используются, находятся непосредственно в этой сборке.
- *PresentationCore.dll* содержит базовые типы, такие как UIElement и Visual, от которых унаследованы все фигуры и элементы управления.

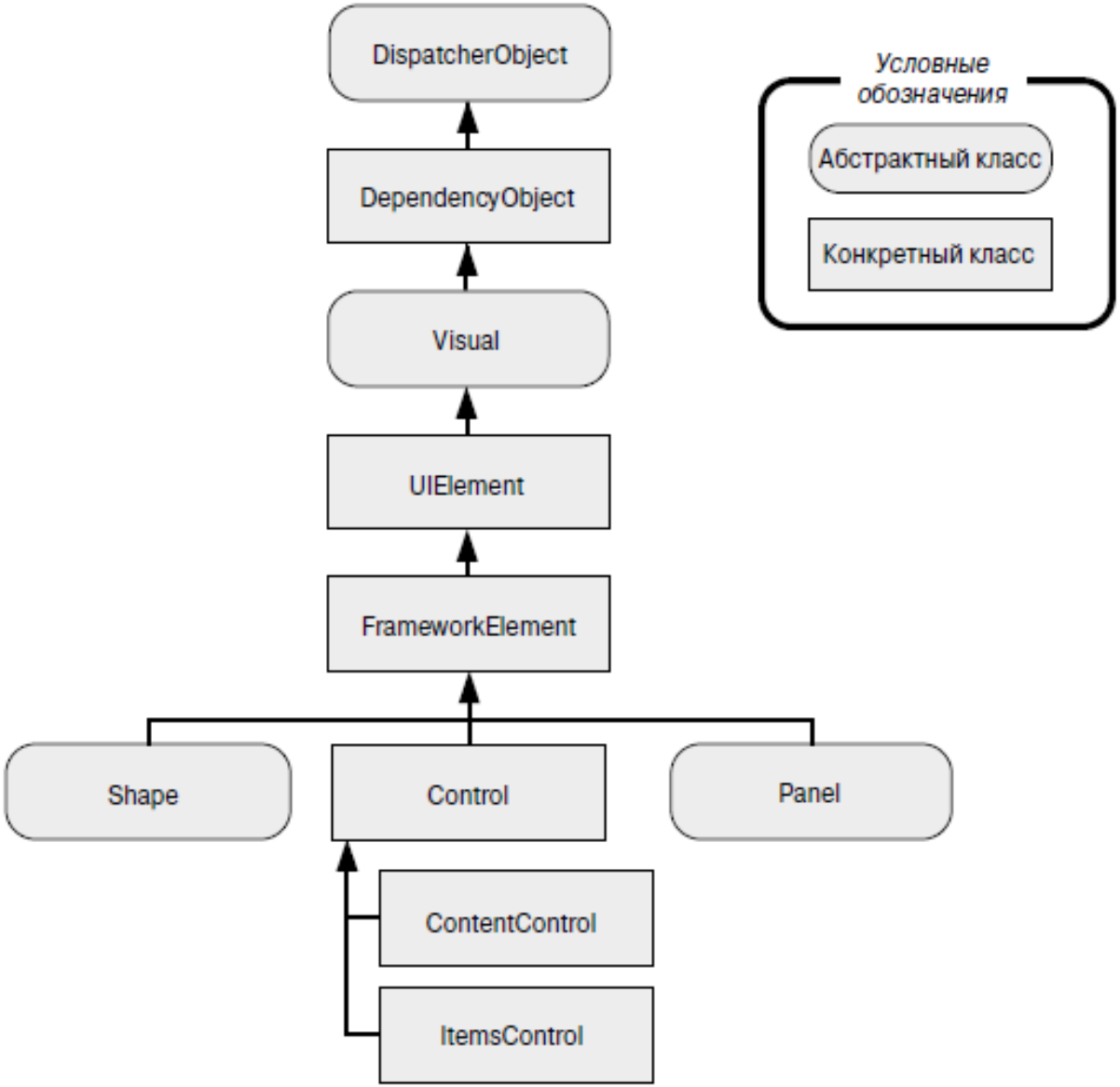
# Архитектура WPF

- *WindowsBase.dll* содержит еще более базовые ингредиенты, которые потенциально могут использоваться вне WPF, такие как DispatcherObject и DependencyObject, поддерживающие механизм свойств зависимости.
- *milcore.dll* — ядро системы визуализации WPF и фундамент уровня медиа-интеграции (Media Integration Layer — MIL). Его составной механизм транслирует визуальные элементы в треугольники и текстуры, которых ожидает Direct3D.

# Архитектура WPF

- *WindowsCodecs.dll — низкоуровневый API, обеспечивающий поддержку изображений* (например, обработку, отображение и масштабирование битовых карт и JPEG).
- *Direct3D — низкоуровневый API, через который визуализируется вся графика в* WPF.
- *User32* используется для определения того, какое место на экране к какой программе относится. В результате он по-прежнему вовлечен в WPF, но не участвует в визуализации распространенных элементов управления.

# Иерархия классов



# Иерархия классов

- ***System.Threading. DispatcherObject***

Приложения WPF используют однопоточную модель (single-thread affinity — STA), а это означает, что весь пользовательский интерфейс принадлежит единственному потоку. Взаимодействовать с элементами пользовательского интерфейса из других потоков небезопасно. Чтобы содействовать работе этой модели, каждое приложение WPF управляется *диспетчером, координирующим сообщения.*

Будучи унаследованным от DispatcherObject, каждый элемент вашего пользовательского интерфейса может удостовериться, выполняется ли код в правильном потоке, и обратиться к диспетчеру, чтобы направить код в поток пользовательского интерфейса.

# Иерархия классов

- ***System.Windows.DependencyObject***

В WPF центральный путь взаимодействия с экранными элементами пролегает через свойства. На ранней стадии цикла проектирования архитекторы WPF решили создать более мощную модель свойств, которая положена в основу таких средств, как уведомления об изменениях, наследуемые значения по умолчанию и более экономичное хранилище свойств. Конечным результатом стало средство *свойств зависимости (dependency* property).

Наследуясь от DependencyObject, классы WPF получают поддержку свойств зависимости.

# Иерархия классов

- ***System.Windows.Media.Visual***

Каждый элемент, появляющийся в WPF, в основе своей является Visual. Вы можете воспринимать класс Visual как единственный объект рисования, инкапсулирующий в себе инструкции рисования, дополнительные подробности рисования (наподобие отсечения, прозрачности и настроек трансформации) и базовую функциональность (такую как проверка попадания). Класс Visual также обеспечивает связь между управляемыми библиотеками WPF и milcore.dll, который визуализирует ваш дисплей. Любой класс, унаследованный от Visual, обладает способностью отображаться в окне

# Иерархия классов

- ***System.Windows.UIElement***

UIElement добавляет поддержку таких сущностей WPF, как компоновка (layout), ввод (input), фокус (focus) и события (events) — все, что команда разработчиков WPF называет аббревиатурой *LIFE.* Здесь определен двухшаговый процесс измерения и организации компоновки, здесь щелчки кнопками мыши и нажатия клавиш трансформируются в более удобные события, такие как MouseEnter. Так же WPF реализует расширенную систему передачи событий, именуемую *маршрутизируемыми событиями (routed events).* UIElement добавляет поддержку команд.

# Иерархия классов

- ***System.Windows.FrameworkElement***

FrameworkElement — конечный пункт в центральном дереве наследования WPF. Он реализует некоторые члены, которые просто определены в UIElement. Например, UIElement устанавливает фундамент для системы компоновки WPF, но FrameworkElement включает ключевые свойства (вроде HorizontalAlignment и Margin), которые поддерживают его. UIElement также добавляет поддержку привязки данных, анимации и стилей — все это центральные средства.

# Иерархия классов

- ***System.Windows.Shapes. Shape***

От этого класса наследуются базовые фигуры, такие как Rectangle, Polygon, Ellipse, Line и Path. Эти фигуры могут быть использованы наряду с более традиционными визуальными элементами Windows вроде кнопок и текстовых полей.

# Иерархия классов

***System.Windows.Controls.Control***

*Элемент управления (control) — это элемент, который может взаимодействовать* с пользователем. К нему, очевидно, относятся такие классы, как TextBox, Button и ListBox. Класс Control добавляет дополнительные свойства для установки шрифта, а также цветов переднего плана и фона. Но наиболее интересная деталь, которую он предоставляет — это поддержка шаблонов, которая позволяет заменять стандартный внешний вид элемента управления вашим собственным рисованием.

# Иерархия классов

- ***System.Windows.Controls.ContentControl***

Это базовый класс для всех элементов управления, которые имеют отдельный кусок содержимого. Сюда относится все — от скромной метки Label до окна Window.

Наиболее впечатляющая часть этой модели заключается в том, что единственный кусок содержимого может быть чем угодно — от обычной строки до панели компоновки, содержащей комбинацию других фигур и элементов управления.

# Иерархия классов

- ***System.Windows.Controls.ItemsControl***

Это базовый класс для всех элементов управления, которые отображают коллекцию каких-то единиц информации, вроде ListBox и TreeView. Списочный элемент управления замечательно гибок; например, используя встроенные средства класса ItemsControl, вы можете трансформировать обычный ListBox в список переключателей, список флажков, ряд картинок или комбинацию совершенно разных элементов по своему выбору. Фактически в WPF все меню, панели инструментов и линейки состояния на самом деле являются специализированными списками, и классы, реализующие их, наследуются от ItemsControl.

# Иерархия классов

- ***System.Windows.Controls.Panel***

Это базовый класс для всех контейнеров компоновки — элементов, которые содержат в себе один или более дочерних элементов и упорядочивают их в соответствии с определенными правилами компоновки. Эти контейнеры образуют фундамент системы компоновки WPF, и их использование — ключ к упорядочиванию вашего содержимого наиболее привлекательным и гибким способом.

# XAML

XAML (сокращение от Extensible Application Markup Language — расширяемый язык разметки приложений) представляет собой язык разметки, используемый для создания экземпляров объектов .NET.

Язык XAML — это технология, которая может быть применима ко многим различным предметным областям, его главное назначение — конструирование пользовательских интерфейсов WPF.

Легче всего разрабатывать сложные, графически насыщенные приложения, если отделить графическую часть от лежащего в основе кода. Таким образом, художники могут заниматься графикой, а разработчики — кодом. Обе части могут проектироваться и совершенствоваться по отдельности, без проблем с версиями.

# XAML

При проектировании приложения WPF в Visual Studio создаваемое вами окно не транслируется в код. Вместо этого оно сериализуется в набор дескрипторов XAML. Когда вы запускаете приложение, эти дескрипторы используются для генерации объектов, составляющих пользовательский интерфейс.

XAML открывает возможности для кооперации, поскольку другие инструменты проектирования понимают формат XAML.

# XAML

Существует несколько подмножеств XAML:

- *WPF XAML включает элементы, описывающие содержимое WPF вроде векторной графики, элементов управления и документов.*
- *XPS XAML — часть WPF XAML, определяющая XML-представление форматированных электронных документов.*
- *Silverlight XAML — подмножество WPF XAML, предназначенное для Silverlight-приложений.*
- *WF XAML включает элементы, описывающие содержимое Windows WorkflowFoundation (WF).*

# Компиляция XAML

BAML (Binary Application Markup Language — двоичный язык разметки приложений) - это не что иное, как двоичное представление XAML. Когда вы компилируете приложение WPF в Visual Studio, все ваши файлы XAML преобразуются в код BAML, и этот код BAML затем встраивается в виде ресурса в финальную сборку DLL или EXE.

BAML поддерживает *лексемы, а это* значит, что длинные куски XAML заменены короткими лексемами. И код BAML не только существенно меньше, но он также оптимизирован таким образом, что он быстрее интерпретируется во время выполнения.

# Основы XAML

Основополагающие правила XAML.

- Каждый элемент в документе XAML отображается на экземпляр класса .NET. Имя элемента соответствует имени класса *в точности. Например, элемент* <*Button*> сообщает WPF, что должен быть создан объект Button.
- Вы можете устанавливать свойства каждого класса через атрибуты. Однако в некоторых ситуациях атрибуты не достаточно мощны, чтобы справиться с этой работой. В этих случаях вам понадобятся вложенные дескрипторы со специальным синтаксисом.

# Основы XAML

Основополагающие правила XAML.

- Как и любой документ XML, код XAML допускает вложение одного элемента внутрь другого. XAML дает каждому классу гибкость в принятии решения относительно того, как справиться с ситуацией. Однако вложение обычно является способом выразить включение (containment).

Другими словами, если вы видите элемент Button внутри элемента Grid, ваш пользовательский интерфейс, вероятно, включает Grid, содержащий внутри себя Button.

# Основы XAML

Простейший документ XAML, представляющий новое пустое окно:

```
1 <Window x:Class="WindowsApplication1.Window1"
2 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4 Title="Window1" Height="300" Width="300">
5
6 <Grid>
7 </Grid>
8 </Window>
```

Этот документ включает всего два элемента — элемент верхнего уровня Window, который представляет все окно, и Grid, куда вы можете поместить свои элементы управления.

# Класс отделенного кода

- XAML позволяет конструировать пользовательский интерфейс, но для того, чтобы создать функционирующее приложение, вам нужен способ подключения обработчиков событий, содержащих код вашего приложения. XAML позволяет легко это сделать с помощью атрибута Class, показанного ниже:

```
1 <Window x:Class="WindowsApplication1.Window1"
```

# *Метод InitializeComponent()*

Метод InitializeComponent() генерируется автоматически при компиляции вашего приложения. По существу все, что делает InitializeComponent() — это вызов метода LoadComponent() класса System.Windows.Application.

Метод LoadComponent() извлекает код BAML (компилированный XAML) из вашей сборки и использует его для построения вашего пользовательского интерфейса. При разборе BAML он создает объекты каждого элемента управления, устанавливает их свойства и прикрепляет все обработчики событий.

# *Именование элементов*

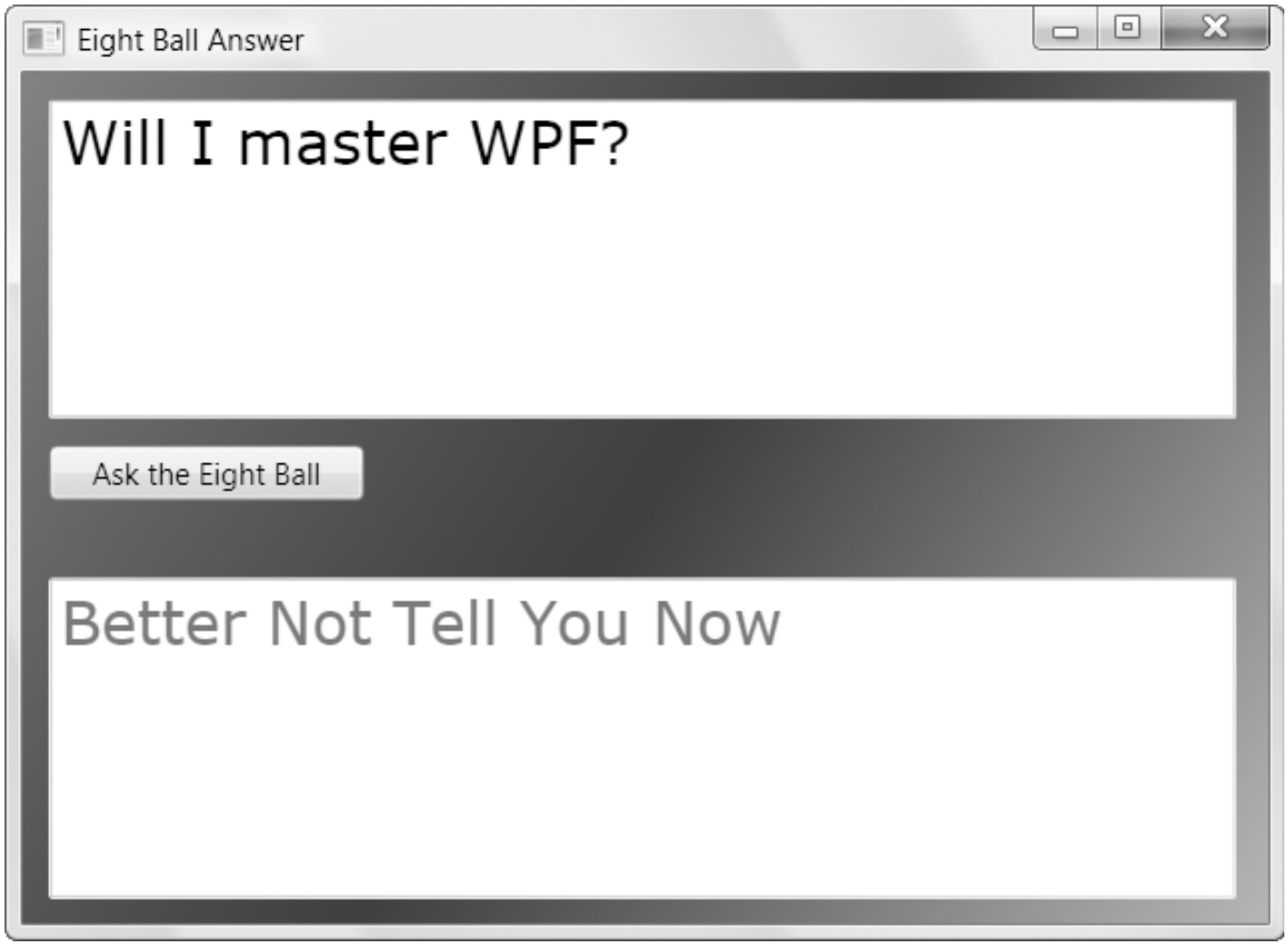
Иногда необходимо программно манипулировать элементами управления, чтобы обеспечить такую возможность, элемент управления должен включать XAML-атрибут Name.

Например, вы можете читать либо изменять свойства, прикреплять или откреплять обработчики событий на лету.

```
<Grid x:Name="grid1">
 </Grid>
```

# Свойства и события в XAML

Пример с автоответчиком на вопросы пользователя.



# Свойства и события в XAML

```
<Window x:Class="EightBall.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Eight Ball Answer" Height="328" Width="412">
  <Grid Name="grid1">
    <Grid.Background>
      ...
    </Grid.Background>
    <Grid.RowDefinitions>
      ...
    </Grid.RowDefinitions>
    <TextBox Name="txtQuestion" ... >
      ...
    </TextBox>
    <Button Name="cmdAnswer" ... >
      ...
    </Button>
    <TextBox Name="txtAnswer" ... >
      ...
    </TextBox>
  </Grid>
</Window>
```

# Свойства и события в XAML

Атрибуты элемента устанавливают свойства соответствующего объекта.

Например, текстовые поля в примере автоответчика конфигурируют выравнивание, поля и шрифт:

```
<TextBox Name="txtQuestion"
VerticalAlignment="Stretch"
  HorizontalAlignment="Stretch"
FontFamily="Verdana" FontSize="24"
  Foreground="Green" ... >
```

# Свойства и события в XAML

Значение в атрибуте XAML всегда представлено простой строкой. Однако свойства объекта могут быть любого типа .NET.

Чтобы преодолеть зазор между строковыми значениями и не строковыми свойствами, анализатору XAML необходимо выполнить преобразование. Это преобразование осуществляется конвертерами типов — базовой частью инфраструктуры .NET.

Конвертер типов играет в своей жизни только одну роль — он предоставляет служебные методы, которые могут преобразовывать определенный тип данных .NET в любой другой тип .NET.

# Свойства и события в XAML

Анализатор XAML выполняет следующие два шага, чтобы найти нужный конвертер типа.

1. Проверяет объявление свойства в поисках атрибута TypeConverter.

2. Если в объявлении свойства отсутствует атрибут TypeConverter, то анализатор XAML проверяет объявление класса соответствующего типа данных.

Если в объявлении свойства или объявлении класса не оказывается ассоциированного конвертера типа, то анализатор XAML генерирует ошибку.

# Сложные свойства

Как бы ни были удобны конвертеры типов, они подходят не для всех сценариев.

XAML предусматривает другой выбор: синтаксис “свойство-элемент”. С помощью этого синтаксиса вы можете добавлять дочерний элемент с именем в форме Родитель.Имя

Ключевая деталь, которая заставляет это работать — это точка (.) в имени элемента. Это отличает свойства от других типов и сложенного содержимого. Свойства.

# Класс **Application**

В процессе выполнения каждое приложение WPF представлено экземпляром класса System.Windows.Application. Этот класс отслеживает все открытые окна в вашем приложении, решает, когда ваше приложение должно быть остановлено, и инициирует события приложения, которые вы можете обрабатывать для выполнения инициализации и очистки.

# Создание объекта Application

- Простейший способ использования класса *Application* заключается в его создании вручную.
- Запущенное приложение будет продолжать работу до тех пор, пока главное окно *и все его прочие окна не будут закрыты.*

# Создание объекта Application

1 Способ: точка входа в приложение (метод Main()), которая создает окно по имени Window1 и запускает новое приложение

```
using System;
using System.Windows;
public class Startup
{
        [STAThread()]
        static void Main()
{
// Создание приложения.
Application app = new Application();
// Создание главного окна.
Window1 win = new Window1();
// Запуск приложения и отображение главного окна.
app.Run(win);
        }
}
```

# Создание объекта Application

2 Способ:

```
using System;
using System.Windows;
public class Startup
{
        [STAThread()]
        static void Main()
{
          // Создать приложение.
          Application app = new Application();
          // Создать, присвоить и показать главное окно.
          Window1 win = new Window1();
          app.MainWindow = win;
          win.Show();
          // "Оживить" приложение.
          app.Run();
    }
    }
```

# Наследование специального класса приложения

- Visual Studio наследует специальный класс от класса Application. В простом приложении этот подход не дает существенного эффекта. Однако если вы планируете обрабатывать события приложения, он предоставляет более изящную модель, потому что вы можете поместить ваш код обработки событий в класс, производный от Application.

```
<Application x:Class="TestApplication.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri="Window1.xaml"
>
</Application>
```

# Наследование специального класса приложения

- Дескриптор Application не только создает специальный класс приложения, но также устанавливает свойство StartupUri для идентификации документа XAML, представляющего главное окно. В результате вам не нужно явно создавать экземпляр этого окна в коде — анализатор XAML сделает это за вас.
- Как и с окном, класс приложения определен в двух отдельных частях, которые сплавляются вместе во время компиляции. Автоматически сгенерированная часть невидима в вашем проекте, но она содержит точку входа Main() и код для запуска приложения.

# Наследование специального класса приложения

```
using System;
using System.Windows;
public partial class App : Application
{
[STAThread()]
public static void Main()
{
TestApplication.App app = new TestApplication.App();
app.InitializeComponent();
app.Run();
}
public void InitializeComponent()
{
this.StartupUri = new Uri("Window1.xaml",
System.UriKind.Relative);
}
}
```

# Наследование специального класса приложения

- Единственное отличие между автоматически сгенерированным кодом, показанным здесь, и специальным классом приложения, который вы можете создать самостоятельно, состоит в том, что автоматически сгенерированный класс использует свойство StartupUri вместо установки свойства MainWindow или передачи главного окна в качестве параметра методу Run().
- Вы можете создать специальный класс приложения, использующий этот подход, до тех пор, пока применяете тот же формат URI. Вам нужно создать относительный объект Uri, который именует документ XAML, находящийся в вашем проекте.

# Останов приложения

- Обычно класс Application оставляет ваше приложение активным до тех пор, пока хотя бы одно окно остается открытым. Если вам не нужно такое поведение, вы можете изменить Application.ShutdownMode.
- Если вы создаете объект Application вручную, вам следует установить свойство ShutdownMode перед запуском Run(). Если вы используете файл App.xaml, то можете просто установить свойство ShutdownMode в коде разметки XAML.

# Останов приложения

**Таблица Значения перечисления `ShutdownMode`**

| Имя | Описание |
|---|---|
| `OnLastWindowClose` | Поведение по умолчанию — ваше приложение выполняется до тех пор, пока существует хотя бы одно открытое им окно. Если вы закрываете главное окно, то свойство `Application.MainWindow` все еще ссылается на объект, представляющий закрытое окно. (Дополнительно вы можете использовать код для переназначения свойства `MainWindow`, чтобы оно указывало на другое окно.) |
| `OnMainWindowClose` | Это традиционный подход — приложение остается живым только до тех пор, пока открыто главное окно. |
| `OnExplicitShutdown` | Приложение никогда не завершается (даже если все окна закрыты), пока вы не вызовете `Application.Shutdown()`. Этот подход может быть оправдан, если ваше приложение является интерфейсом для долго выполняющейся задачи, или если вы хотите использовать более сложную логику, чтобы решить, когда ваше приложение должно закрыться (и тогда вы вызываете метод `Application.Shutdown()`). |

# Останов приложения

- Независимо от того, какой способ останова вы используете, вы всегда можете вызвать метод Application.Shutdown() для немедленного завершения вашего приложения. (Разумеется, когда вы вызываете метод Shutdown(), ваше приложение необязательно сразу завершится. Вызов Application.Shutdown() заставляет метод Application.Run() немедленно вернуть управление, но может существовать дополнительный код, который выполняется в методе Main() или реагирует на событие Application.Exit.)

# События класса Application

- Изначально файл App.xaml.cs не содержит никакого кода. Хотя какой-либо код не обязателен, вы можете добавить код обработки событий приложения. Класс Application предоставляет небольшой набор полезных событий. В таблице перечислены наиболее важные из них.

**Таблица События класса `Application`**

| Имя | Описание |
|---|---|
| `Startup` | Происходит после вызова метода `Application.Run()` и непосредственно перед показом главного окна (если вы передаете главное окно методу `Run()`). |
| `Exit` | Происходит, когда приложение останавливается по любой причине, непосредственно перед возвратом из метода `Run()`. Вы не можете в этот момент отменить останов, хотя код вашего метода `Main()` может повторно запустить приложение. Вы можете использовать событие `Exit` для установки целочисленного кода выхода, возвращаемого методом `Run()`. |

# События класса Application

**Таблица** **События класса `Application`**

| Имя | Описание |
|---|---|
| `SessionEnding` | Происходит по завершении сеанса Windows, например, когда пользователь выходит из системы или останавливает компьютер. (Вы можете определить, что именно произошло, проверив свойство `SessionEndingCancelEventArgs.ReasonSessionEnding`). Также вы можете отменить останов, присвоив `SessionEndingCancelEventArgs.Cancel` значение `true`. Если этого не делать, то WPF вызовет метод `Application.Shutdown()` по завершении обработчика события. |
| `Activated` | Происходит, когда активизируется одно из окон приложения. Это случается, когда вы переключаетесь с другой программы Windows на это приложение. Также случается при первом показе окна. |
| `Deactivated` | Происходит при деактивизации окна приложения. Случается, когда вы переключаетесь на другую программу Windows. |

# События класса Application

**Таблица** **События класса** `Application`

| Имя | Описание |
|---|---|
| `DispatcherUnhandledException` | Происходит, когда возникает необработанное исключение в любом месте вашего приложения (в главном потоке приложения). (Эти исключения перехватывает диспетчер приложения.) Реагируя на это событие, вы можете протоколировать критичные ошибки – можно даже нейтрализовать исключение и продолжить работу приложения, установив свойство `DispatcherUnhandledExceptionEventArgs.Handled` в `true`. Вы должны предпринять этот шаг только в том случае, если уверены, что ваше приложение пребывает в корректном состоянии и его работа может быть продолжена. |

# События класса Application

- При обработке событий у вас есть два выбора: вы можете прикрепить обработчик событий или же переопределить соответствующий защищенный метод. Если вы предпочитаете обрабатывать события приложения, вам не нужно использовать код делегата для привязки его в качестве обработчика. Вместо этого вы можете прикрепить его, используя атрибут в файле App.xml.
- Для инициализации каждого события приложения (из перечисленных в табл.) вызывается соответствующий метод. Имя метода совпадает с именем события, но с добавлением префикса *On, так что Startup становится OnStartup(), Exit — OnExit() и т.д.*

# Обработка аргументов командной строки

- Чтобы обработать аргументы командной строки, необходимо реагировать на событие Application.Startup. Аргументы передаются в виде массива строк через свойство StartupEventArgs.Args.
- Например, предположим, что вы хотите загрузить документ, когда его имя передается в виде аргумента командной строки. В этом случае имеет смысл прочесть аргументы командной строки и выполнить необходимую дополнительную инициализацию.

# Доступ к текущему приложению

- Вы можете получить экземпляр текущего приложения из любой точки его кода, обратившись к свойству Application.Current. Это обеспечивает возможность простейшего взаимодействия между окнами, поскольку любое окно может получить доступ к текущему объекту Application и через него — к ссылке на главное окно:

```
Window main = Application.Current.MainWindow;
MessageBox.Show("The main window is " + main.Title);
```

# Доступ к текущему приложению

Если необходимо получить доступ к любым методам, свойствам или событиям, которые вы добавили к своему специальному классу главного окна, вам придется выполнить приведение объекта окна к соответствующему типу. Если главное окно представляет собой экземпляр специального класса MainWindow, вы можете использовать следующий код:

```
MainWindow main = (MainWindow)Application.Current.MainWindow;
main.DoSomething();
```

# Доступ к текущему приложению

- На практике большинство приложений предпочитают использовать более структурированную форму взаимодействия между окнами. Если у вас есть несколько долго выполняющихся окон, которые открыты одновременно, и которым нужно как-то взаимодействовать между собой, имеет больше смысла хранить ссылки на эти окна в специальном классе приложения. Таким образом, вы всегда сможете найти именно то окно, которое вам нужно.

# Взаимодействие между окнами

- Класс Application позволяет хранить ссылки на важные окна, так чтобы одно окно могло обращаться к другому.

Эта техника имеет смысл, когда у вас есть немодальное окно, существующее в течение длительного времени и доступное нескольким разным классам (а не только классу, создавшему его).

# Взаимодействие между окнами

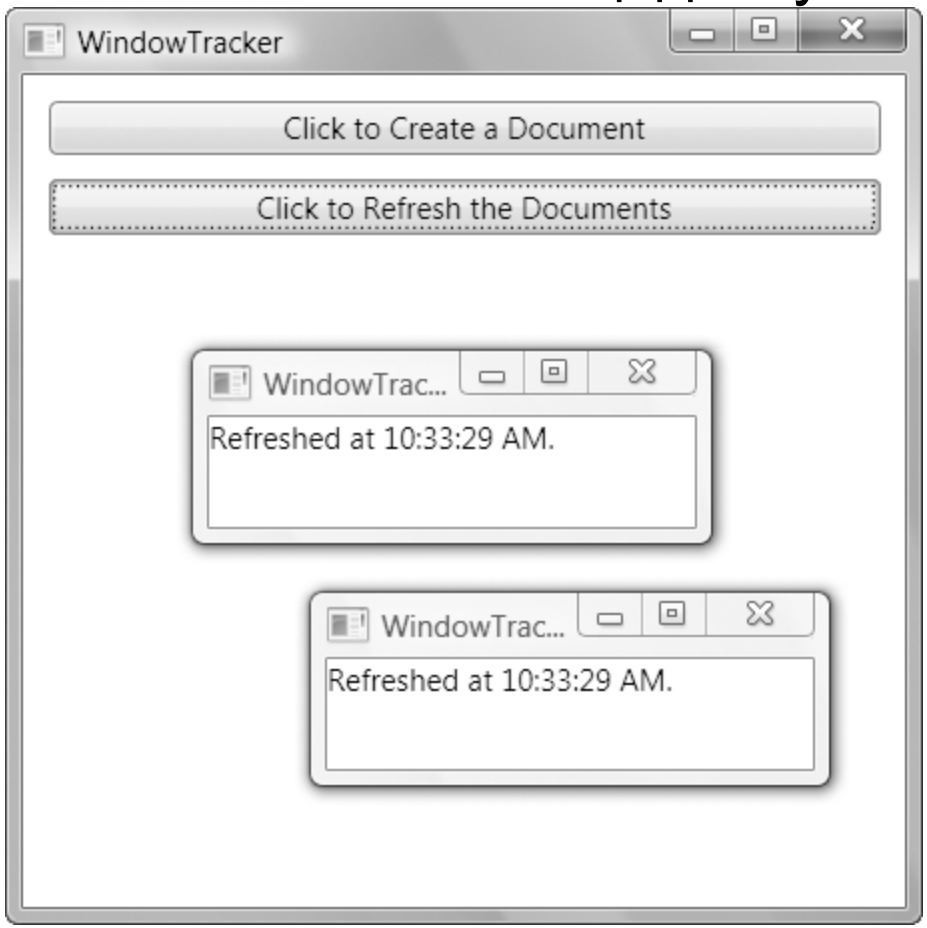
```
public partial class App : Application
{
private List<Document> documents = new List<Document>();
public List<Document> Documents
{
get { return documents; }
set { documents = value; }
}
}
```

Теперь, когда вы создаете новый документ, то должны просто не забыть добавить его к коллекции Documents. Вот обработчик события, который реагирует на щелчок кнопки и выполняет эту задачу:

```
private void cmdCreate_Click(object sender, RoutedEventArgs e)
{
Document doc = new Document();
doc.Owner = this;
doc.Show();
((App)Application.Current).Documents.Add(doc);
}
```

# Взаимодействие между окнами

- Теперь вы можете использовать эту коллекцию в любом месте вашего кода, чтобы проходить циклом по всем документам и использовать их общедоступные члены.



# Приложение одного экземпляра

- Обычно вы можете запустить столько копий приложения WPF, сколько вы хотите. В некоторых сценариях этот дизайн совершенно оправдан. Однако в других случаях это может стать проблемой, особенно при построении приложений, основанных на документах.
- Например, работа Microsoft Word. Независимо от того, сколько документов вы открываете (или как вы открываете их), только единственный экземпляр winword.exe

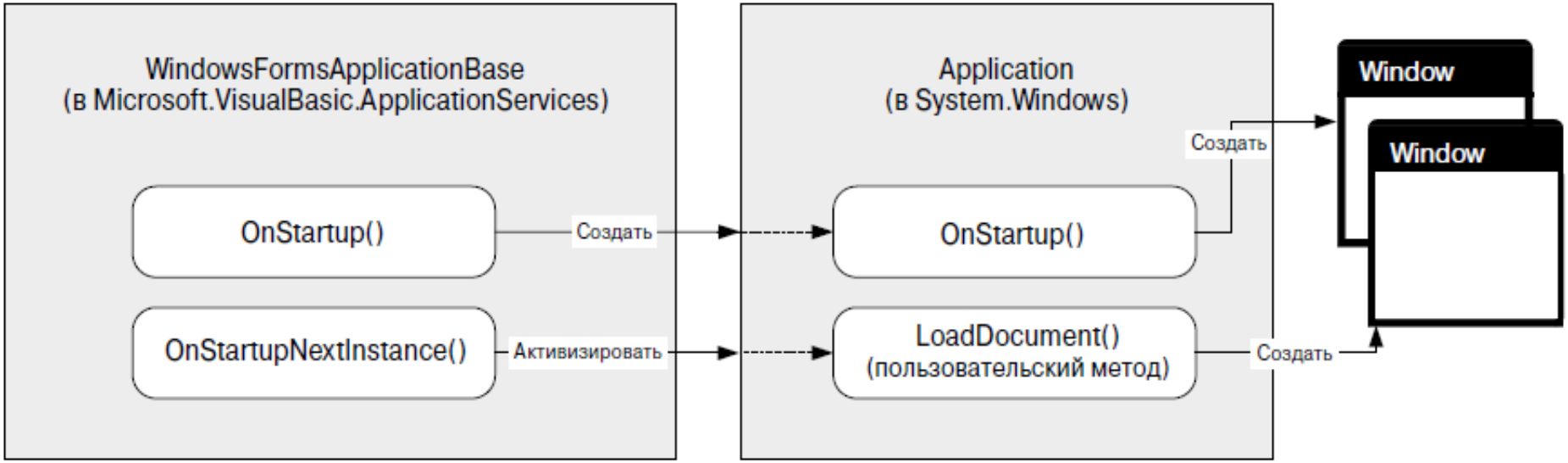
# Приложение одного экземпляра

- В WPF не предусмотрено “родного” решения для приложений одного экземпляра, но вы можете использовать несколько обходных маневров. Базовая техника заключается в проверке существования другого запущенного экземпляра вашего приложения при возникновении события Application.Startup.
- Простейший путь сделать это состоит в использовании системного мьютекса (объекта синхронизации, предоставляемого операционной системой, позволяющей межпроцессное взаимодействие).

# Приложение одного экземпляра

- Но простейший подход, рекомендованный командой WPF, заключается в использовании встроенной поддержки, которая предоставляется в Windows Forms, и изначально предназначалась для приложений Visual Basic. Этот подход обрабатывает все запутанные детали “за кулисами”.
- По сути, класс приложения старого стиля служит оболочкой для вашего класса приложения WPF. Когда запускается ваше приложение, вы создаете класс приложения старого стиля, который затем создаст класс приложения WPF. Класс приложения старого стиля осуществляет управление экземплярами, в то время как класс приложения WPF обслуживает реальное приложение.

# Приложение одного экземпляра




Упаковка приложения WPF в оболочку класса `WindowsFormsApplicationBase`

Первый шаг к применению этого подхода заключается в добавлении ссылки на сборку Microsoft.VisualBasic.dll и наследовании специального класса от класса Microsoft.VisualBasic.ApplicationServices.WindowsFormsApplicationBase.

# Приложение одного экземпляра

- Этот класс обеспечивает три важных члена, которые вы используете для управления экземплярами:
- *Свойство IsSingleInstance*
- *Метод OnStartup()*
- *Метод OnStartupNextInstance()*

- Свойство IsSingleInstance позволяет создать приложение одного экземпляра. Вы устанавливаете это свойство в true в конструкторе.

# Приложение одного экземпляра

- Метод OnStartup() инициируется при старте приложения. Вы переопределяете этот метод и создаете в этой точке объект приложения WPF.
- Метод OnStartupNextInstance() инициируется, когда запускается другой экземпляр приложения. Этот метод обеспечивает доступ к аргументам командной строки. В этой точке вы, вероятно, вызовете метод класса вашего приложения WPF, чтобы отобразить новое окно, не создавая другого объекта приложения.

# Приложение одного экземпляра

- Когда запускается приложение, этот класс создает экземпляр WpfApp, представляющего собой специальный класс приложения WPF (класс, унаследованный от System.Windows.Application).
- Класс WpfApp включает некоторую стартовую логику, которая отображает главное окно, наряду со специальным методом ShowDocument(), который загружает окно документа для заданного файла. Каждый раз, когда имя файла передается SingleInstanceApplicationWrapper через командную строку, SingleInstanceApplicationWrapper вызывает WpfApp.ShowDocument().

# Приложение одного экземпляра