# **ЛАБА 6-7**

# **Перечислите преимущества и недостатки WPF?**

**Преимущества:**

1. способность сделать очень богатые UIs относительно легко.
2. проще анимация и специальные эффекты
3. масштабируемости
4. легко создать собственный внешний вид
5. поддерживает Windows Forms
6. в возможность повторного использования существующего кода

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Использование традиционных языков .NET-платформы - C#, F# и VB.NET для создания логики приложения
2. Возможность декларативного определения графического интерфейса с помощью специального языка разметки XAML, основанном на xml и представляющем альтернативу программному созданию графики и элементов управления, а также возможность комбинировать XAML и C#/VB.NET
3. Независимость от разрешения экрана: поскольку в WPF все элементы измеряются в независимых от устройства единицах, приложения на WPF легко масштабируются под разные экраны с разным разрешением.
4. создание трехмерных моделей, привязка данных, использование таких элементов, как стили, шаблоны, темы и др.
5. Хорошее взаимодействие с WinForms, благодаря чему, например, в приложениях WPF можно использовать традиционные элементы управления из WinForms.
6. Богатые возможности по созданию различных приложений: это и мультимедиа, и двухмерная и трехмерная графика, и богатый набор встроенных элементов управления, а также возможность самим создавать новые элементы, создание анимаций, привязка данных, стили, шаблоны, темы и многое другое
7. Аппаратное ускорение графики - вне зависимости от того, работаете ли вы с 2D или 3D, графикой или текстом, все компоненты приложения транслируются в объекты, понятные Direct3D, и затем визуализируются с помощью процессора на видеокарте, что повышает производительность, делает графику более плавной.
8. Создание приложений под множество ОС семейства Windows

**Недостатки:**

1. набор управления в коробке WPF намного более ограничен, чем у Приложения WinForms.
2. существует большая поддержка в пространстве управления 3rd-party для WinForms.
3. создайте свой собственный внешний вид в приложении-это большая работа.
4. C# работает медленнее. Это несколько занято уход при использовании WPF, хотя в настоящее время запуск приложения WPF все еще немного медленный. Однако после запуска программы анимационные эффекты становятся очень плавными.
5. CозданиT приложений с большим количеством трехмерных изображений, прежде всего игр, лучше использовать другие средства - DirectX или специальные фреймворки, такие как Monogame или Unity.
6. объем программ на WPF и потребление ими памяти в процессе работы в среднем несколько выше чем в Windows Forms.

# **Зачем нужен язык XAML? Каким образом он используется в WPF?**

XAML (eXtensible Application Markup Language) - язык разметки, используемый для инициализации объектов в технологиях на платформе .NET.

Применительно к WPF данный язык используется прежде всего для создания пользовательского интерфейса декларативным путем. Мы будем говорить о нем чаще всего именно как о языке разметки, который позволяет создавать декларативным путем интерфейс, наподобие HTML в веб-программировании.

# **Какие бывают контейнеры компоновки?**

Компоновка (layout) представляет собой процесс размещения элементов внутри контейнера.

Благодаря компоновке мы можем удобным нам образом настроить элементы интерфейса, позиционировать их определенным образом. Например, элементы компоновки в WPF позволяют при ресайзе - сжатии или растяжении масштабировать элементы, что очень удобно, а визуально не создает всяких шероховатостей типа незаполненных пустот на форме.

В WPF компоновка осуществляется при помощи специальных контейнеров. Фреймворк предоставляет нам следующие контейнеры:

**Grid (**Это наиболее мощный и часто используемый контейнер, напоминающий обычную таблицу. Он содержит столбцы и строки, количество которых задает разработчик. Для определения строк используется свойство RowDefinitions, а для определения столбцов - свойство ColumnDefinitions**)**,

**UniformGrid(**Аналогичен контейнеру Grid контейнер UniformGrid, только в этом случае все столбцы и строки одинакового размера и используется упрощенный синтаксис для их определения:**)**,

**StackPanel(**Это более простой элемент компоновки. Он располагает все элементы в ряд либо по горизонтали, либо по вертикали в зависимости от ориентации.**)**,

**WrapPanel(**Эта панель, подобно StackPanel, располагает все элементы в одной строке или колонке в зависимости от того, какое значение имеет свойство Orientation - Horizontal или Vertical. Главное отличие от StackPanel - если элементы не помещаются в строке или столбце, создаются новые столбец или строка для не поместившихся элементов.**)**,

**DockPanel(**Этот контейнер прижимает свое содержимое к определенной стороне внешнего контейнера. Для этого у вложенных элементов надо установить сторону, к которой они будут прижиматься с помощью свойства DockPanel.Dock**)**

**Canvas(**Контейнер Canvas является наиболее простым контейнером. Для размещения на нем необходимо указать для элементов точные координаты относительно сторон Canvas. Для установки координат элементов используются свойства Canvas.Left, Canvas.Right, Canvas.Bottom, Canvas.Top. Например, свойство Canvas.Left указывает, на сколько единиц от левой стороны контейнера будет находиться элемент, а свойство Canvas.Top - насколько единиц ниже верхней границы контейнера находится элемент;

в качестве единиц используются не пиксели, а независимые от устройства единицы, которые помогают эффективно управлять масштабированием элементов. Каждая такая единица равна 1 /96 дюйма, и при стандартной установке в 96 dpi эта независимая от устройства единица будет равна физическому пикселю;

Если элемент не использует свойства Canvas.Top и другие, то по умолчанию свойства Canvas.Left и Canvas.Top будут равны нулю, то есть он будет находиться в верхнем левом углу**)**.

# **Перечислите основные группы элементов управления. Расскажите про объектную модель WPF.**

Все элементы управления(для взаимодействия с пользователем) могут быть условно разделены на несколько подгрупп:

* **Элементы управления содержимым**, например кнопки (Button), метки (Label)
* **Специальные контейнеры**, которые содержат другие элементы, но в отличие от элементов Grid или Canvas не являются контейнерами компоновки - ScrollViewer,GroupBox
* **Декораторы**, чье предназначение создание определенного фона вокруг вложенных элементов, например, Border или Viewbox.
* **Элементы управления списками**, например, ListBox, ComboBox.
* **Текстовые элементы управления**, например, TextBox, RichTextBox.
* **Элементы, основанные на диапазонах значений**, например, ProgressBar, Slider.
* **Элементы для работ с датами**, например, DatePicker и Calendar.
* **Остальные элементы управления**, которые не вошли в предыдущие подгруппы, например, Image.

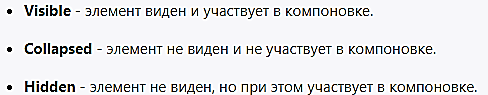
Все элементы управления наследуются от общего класса **System.Window.Controls.Control** и имеют ряд общих свойств. А общую иерархию элементов управления можно представить следующим образом:

Рассмотрим некоторые из основных свойств, которые наследуются элементами управления:

### **Name**(По установленному имени впоследствии можно будет обращаться к элементу, как в коде, так и в xaml разметке.)

### **FieldModidier**(Свойство FieldModifier задает модификатор доступа к объекту: private, protected, internal, protected internal и public)

### **Visibility**(Это свойство устанавливает параметры видимости элемента и может принимать одно из трех значений:)

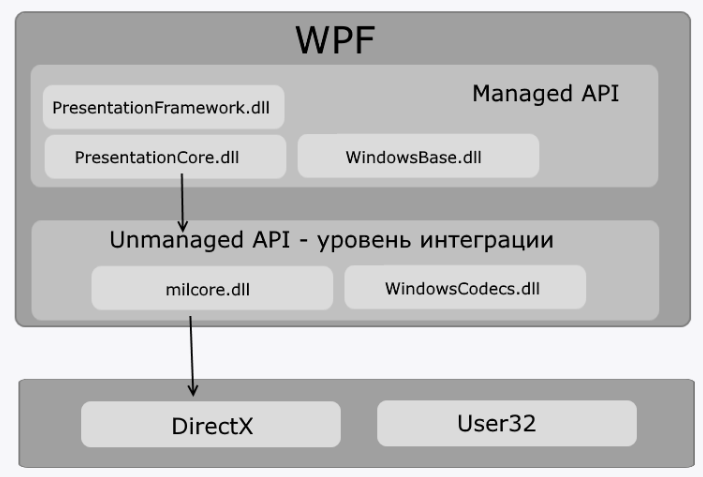


### **Cursor**(Это свойство позволяет нам получить или установить курсор для элемента управления в одно из значений, например, **Hand**, **Arrow**, **Wait** и др)

### **FlowDirection**(Данное свойство задает направление текста. Если оно равно RightToLeft, то текст начинается с правого края, если - LeftToRight, то с левого.)

Технология WPF (Windows Presentation Foundation) является часть экосистемы платформы .NET и представляет собой подсистему для построения графических интерфейсов. Одной из важных особенностей является использование языка декларативной разметки интерфейса XAML, основанного на XML: вы можете создавать насыщенный графический интерфейс, используя или декларативное объявление интерфейса, или код на управляемых языках C#, VB.NET и F#, либо совмещать и то, и другое.

АРХИТЕКТУРА WPF



Как видно на схеме, WPF разбивается на два уровня: managed API и unmanaged API (уровень интеграции с DirectX). Managed API (управляемый API-интерфейс) содержит код, исполняемый под управлением общеязыковой среды выполнения .NET - Common Language Runtime. Этот API описывает основной функционал платформы WPF и состоит из следующих компонентов:

* **PresentationFramework.dll**: содержит все основные реализации компонентов и элементов управления, которые можно использовать при построении графического интерфейса
* **PresentationCore.dll**: содержит все базовые типы для большинства классов из PresentationFramework.dll
* **WindowsBase.dll**: содержит ряд вспомогательных классов, которые применяются в WPF, но могут также использоваться и вне данной платформы

Unmanaged API используется для интеграции вышележащего уровня с DirectX:

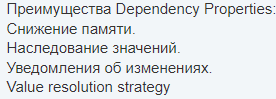
* **milcore.dll**: собственно обеспечивает интеграцию компонентов WPF с DirectX. Данный компонент написан на неуправляемом коде (С/С++) для взаимодействия с DirectX.
* **WindowsCodecs.dll**: библиотека, которая предоставляет низкоуровневую поддержку для изображений в WPF

# **Объясните назначение класса DependencyProperty. Зачем нужны свойства зависимостей? Как создать новой свойство зависимости и в каких случаях это необходимо?**

Dependency Property (Свойство зависимости) — “набор” свойств с дополнительными возможностями, позволяющими работать с анимацией, привязкой данных, стилями. Также они эффективно потребляют память и поддерживают уведомления об изменениях.

Одной из основной черт Dependecy Property является приоретизация. Т.е. наивысший приоритет будет иметь анимация, самый низкий – значения по умолчанию

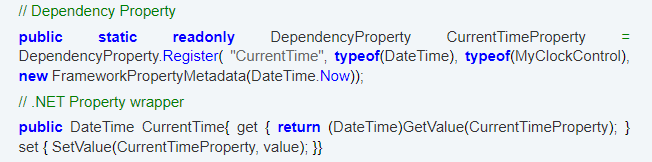
Для того чтобы в классе можно было реализовывать Dependency Property необходимо, чтобы класс наследовался от DependencyObject либо его наследников, таких например как UIElement, Geometry, FrameworkTemplate, Style и других.



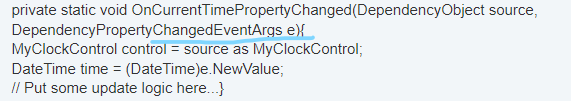
Зачем нужны? Свойства зависимостей и система свойств WPF расширяют функциональные возможности свойств, предоставляя тип, который обеспечивает свойство, в качестве альтернативы стандартному шаблону резервного копирования свойства с закрытым полем. Имя этого типа — [DependencyProperty](https://docs.microsoft.com/ru-ru/dotnet/api/system.windows.dependencyproperty). Другой важный тип, определяющий систему свойств WPF, — это [DependencyObject](https://docs.microsoft.com/ru-ru/dotnet/api/system.windows.dependencyobject) , определяющий базовый класс, который может регистрировать и владеть свойством зависимостей.

Как создать? Чтобы создать DependencyProperty, надо добавить статическое поле типа DepdencyProperty и вызвать DependencyProperty.Register () для создания экземпляра свойства зависимостей. Название DependendyProperty должны всегда заканчиваться на "....Property". Это соглашение об именовании в WPF.

**Важно: Не добавляйте логику для этих свойств, поскольку они вызываются только при установке свойства из кода. Если вы установите свойство от XAML, метод SetValue () вызывется напрямую.**



Изменение уведомления является статическим методом, который вызывается каждый раз, когда изменяется значение TimeProperty. Новое значение передается в EventArgs, объект, на котором значение изменилось передается в качестве источника.



Или так [C# и WPF | Создание свойств зависимостей (metanit.com)](https://metanit.com/sharp/wpf/13.3.php?ysclid=l3gcropsgy)

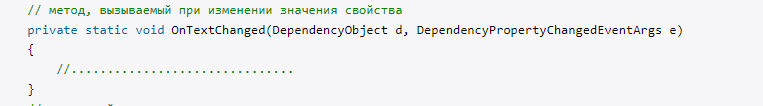
Как создать свойство зависимости?



Статическое свойство TextProperty является свойством зависимостей, представляя объект **System.Windows.DependencyProperty**. По соглашениям по именованию все свойства зависимостей представляют статические публичные поля (public static) с суффиксом *Property*.

Затем в статическом конструкторе класса происходит регистрация свойства с помощью метода **DependencyProperty.Register()**, в который передается ряд параметров:

* **имя свойства** (в данном случае "Text"). Как правило, соответствует названию свойства зависимостей без суффикса *Property*
* **тип свойства** (в данном случае string)
* **тип, который владеет свойством** - собственно тот тип, в котором свойство определено или в данном случае тип TextBlock
* Необязательный параметр **FrameworkPropertyMetadata** устанавливает дополнительные настройки свойства
* В качестве пятого необязательного параметра может использоваться ссылка на метод, который производит валидацию свойства. В данном случае этот параметр опущен.

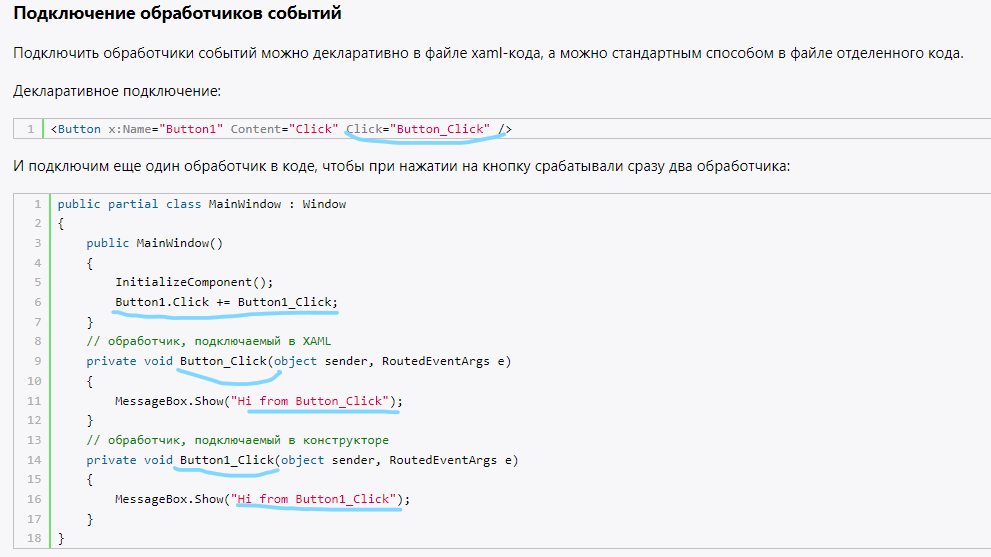


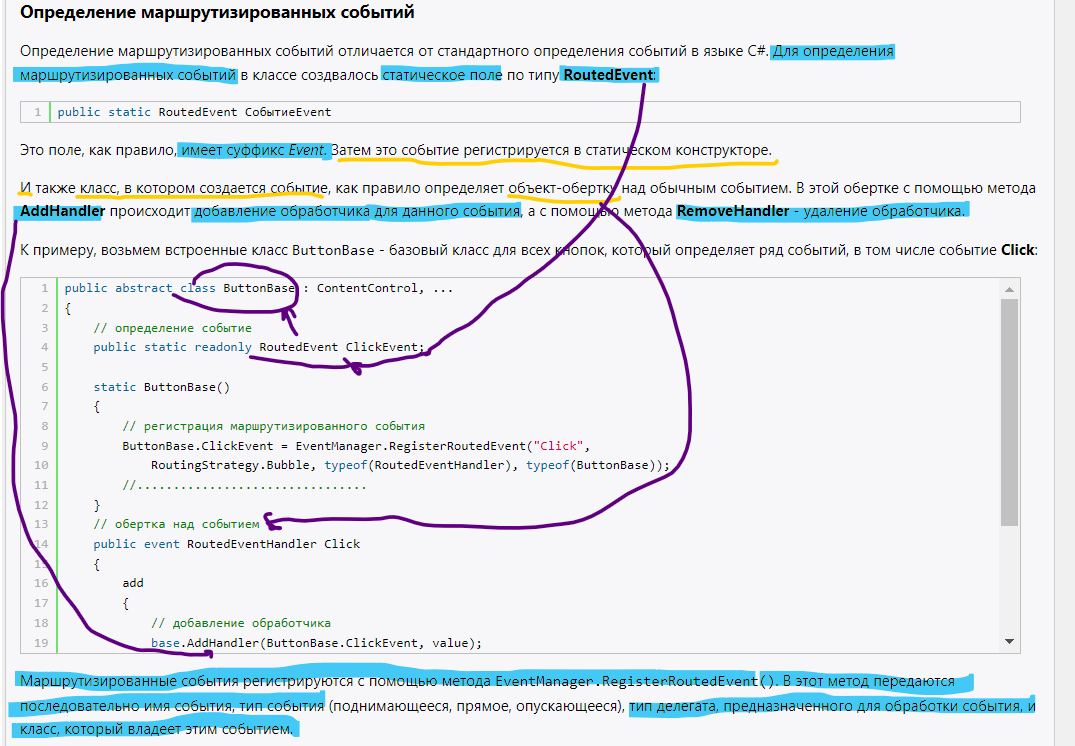
# **Каким образом осуществляется обработка событий в WPF? Что означает концепция маршрутизированных событий? Перечислите основные группы событий.**

WPF предлагает новую концепцию событий - **маршрутизированные события** (**routed events**).

Для элементов управления в WPF определено большое количество событий, которые условно можно разделить на несколько групп:

* События клавиатуры
* События мыши
* События стилуса
* События сенсорного экрана/мультитач
* События жизненного цикла





### **Маршрутизация событий**

События, возникнув на одном элементе, могут обрабатываться на другом. События могут подниматься и опускаться по дереву элементов.

Так, маршрутизируемые события делятся на три вида:

* **Прямые** (direct events) - они возникают и отрабытывают на одном элементе и никуда дальше не передаются. Действуют как обычные события.
* **Поднимающиеся** (bubbling events) - возникают на одном элементе, а потом передаются дальше к родителю - элементу-контейнеру и далее, пока не достигнет наивысшего родителя в дереве элементов.
* **Опускающиеся, туннельные** (tunneling events) - начинает отрабатывать в корневом элементе окна приложения и идет далее по вложенным элементам, пока не достигнет элемента, вызвавшего это событие.
* **Прикрепляемые события** (Attached events) - Если у нас есть несколько элементов одного и того же типа и мы хотим привязать их к одному событию, то мы можем воспользоваться прикрепляемыми событиями.

Все маршрутизируемые события используют класс **RoutedEventArgs** (или его наследников), который представляет доступ к следующим свойствам:

* **Source**: элемент логического дерева, являющийся источником события.
* **OriginalSource**: элемент визуального дерева, являющийся источником события. Обычно то же самое, что и Source
* **RoutedEvent**: представляет имя события
* **Handled**: если это свойство установлено в True, событие не будет подниматься и опускаться, а ограничится непосредственным источником.

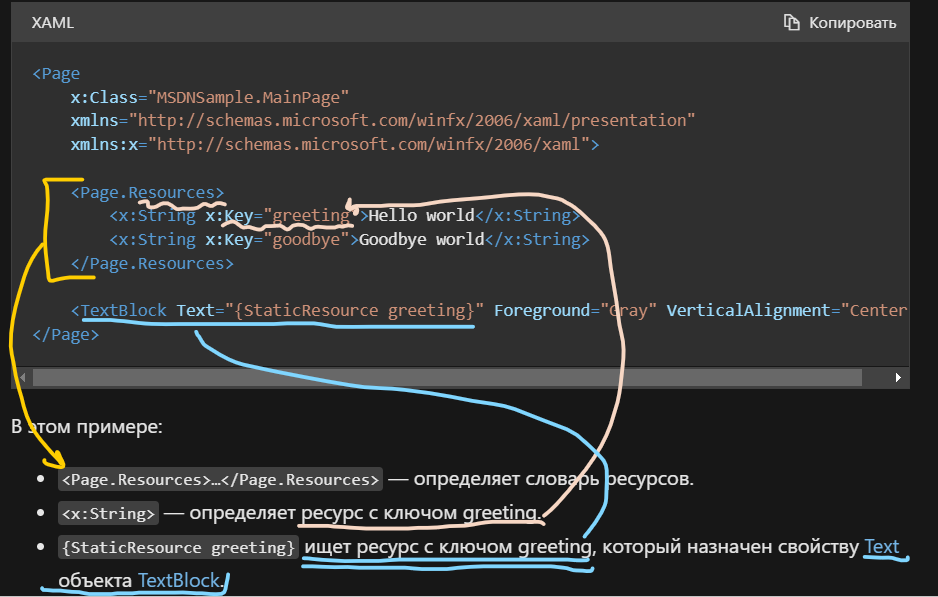
# **Что такое Resource Dictionary?**

Чтобы разделить ресурсы между множеством проектов можно создать словарь ресурсов. Словарь ресурсов представляет собой просто XAML-документ, который всего лишь хранит необходимые ресурсы.

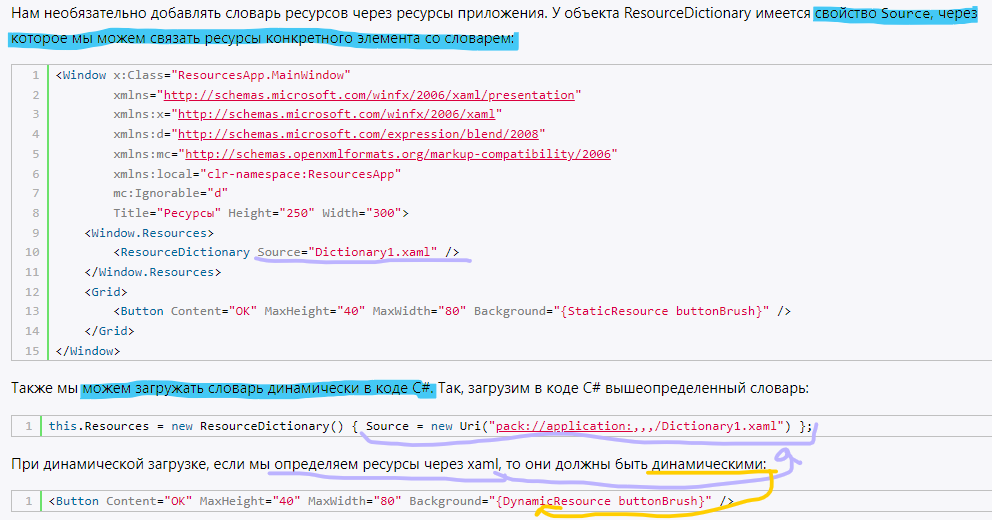
Вы можете определить пользовательский интерфейс или ресурсы для вашего приложения с помощью XAML. Обычно ресурсы представляют собой определения некоторого объекта, который предполагается использовать несколько раз. Чтобы к ресурсу XAML можно было обратиться позже, вам необходимо указать ключ, который выступает в качестве его имени. На ресурс можно ссылаться из любого места приложения или с любой используемой им страницы XAML. Вы можете определить свои ресурсы с помощью элемента [ResourceDictionary](https://docs.microsoft.com/ru-ru/uwp/api/Windows.UI.Xaml.ResourceDictionary) из среды выполнения Windows XAML. Затем вы можете ссылаться на ресурсы с помощью расширения разметки [StaticResource](https://docs.microsoft.com/ru-ru/windows/uwp/xaml-platform/staticresource-markup-extension) или [ThemeResource](https://docs.microsoft.com/ru-ru/windows/uwp/xaml-platform/themeresource-markup-extension).

Ресурсы XAML — это объекты, ссылка на которые из разметки осуществляется несколько раз. Ресурсы определены в [ResourceDictionary](https://docs.microsoft.com/ru-ru/uwp/api/Windows.UI.Xaml.ResourceDictionary) обычно в отдельном файле или в верхней части страницы разметки

**ОПРЕДЕЛЕНИЕ И ИСПОЛЬЗОВАНИЕ РЕСУРСОВ XAML**

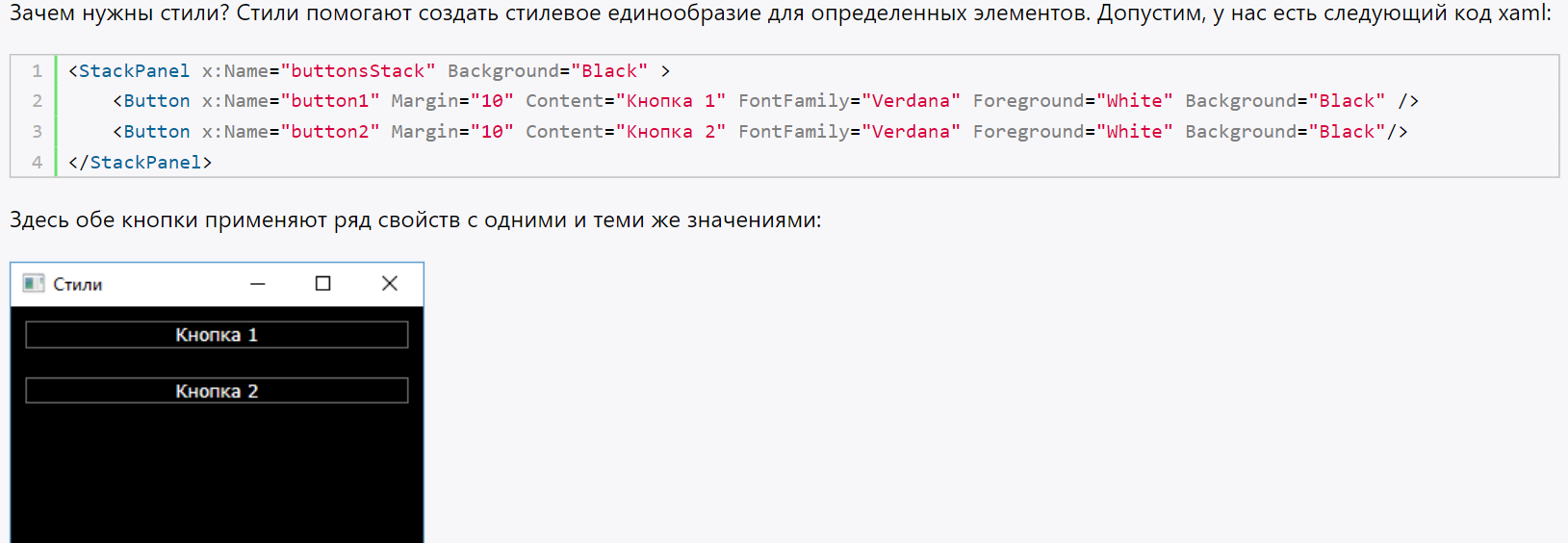


**ЗАГРУЗКА СЛОВАРЯ РЕСУРСОВ**

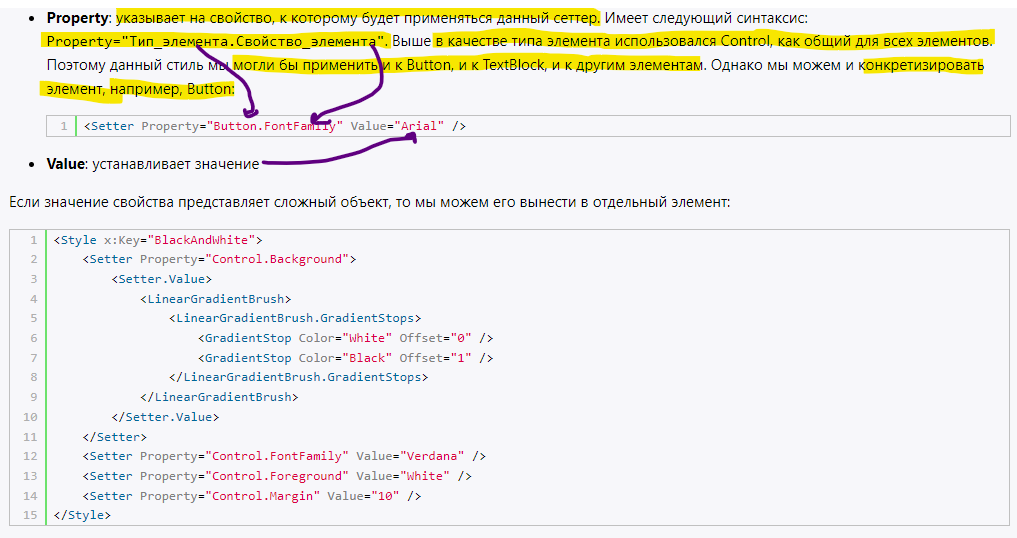


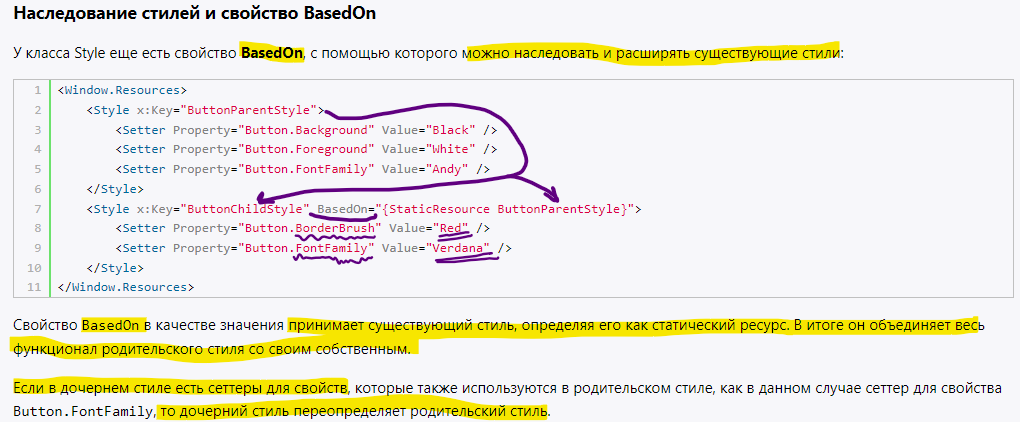
# **Что такое стиль и как его создать? В чем преимущество использования стиля?**

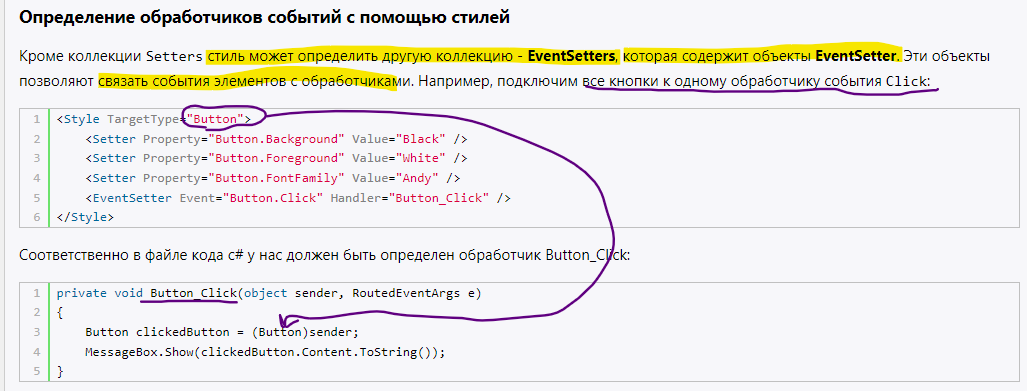
Стили позволяют определить набор некоторых свойств и их значений, которые потом могут применяться к элементам в xaml. Стили хранятся в ресурсах и отделяют значения свойств элементов от пользовательского интерфейса. Также стили могут задавать некоторые аспекты поведения элементов с помощью триггеров.



Стиль создается как ресурс с помощью объекта **Style**, который представляет класс **System.Windows.Style**. И как любой другой ресурс, он обязательно должен иметь ключ. С помощью коллекции **Setters** определяется группа свойств, входящих в стиль. В нее входят объекты **Setter**, которые имеют следующие свойства:







# **Что такое Command? Расскажите паттерн Command. Как в WPF используется Command и для чего?**

**Команда** — это поведенческий паттерн, позволяющий заворачивать запросы или простые операции в отдельные объекты.

Это позволяет откладывать выполнение команд, выстраивать их в очереди, а также хранить историю и делать отмену.

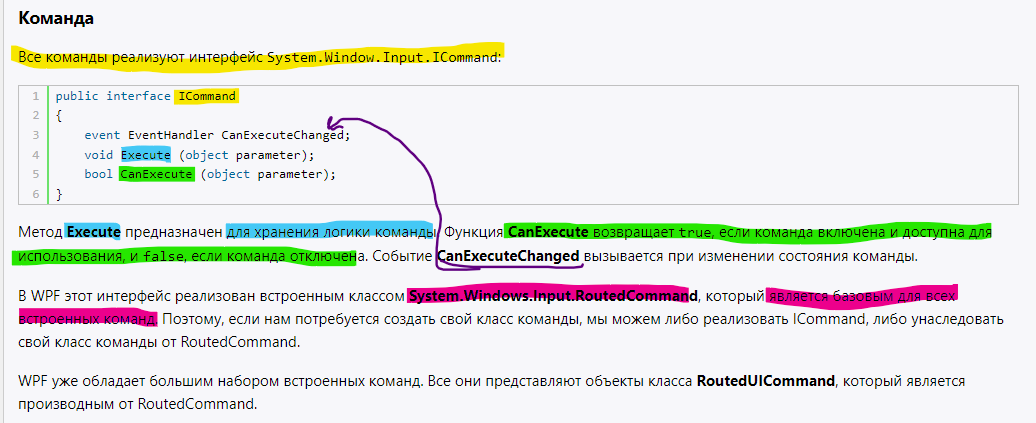
В Windows Presentation Foundation предлагается поддержка независимых от элементов управления событий через управляющие команды (control commands). Обычное событие .NET определяется внутри некоторого базового класса и может быть использовано этим классом и его наследниками. Таким образом, нормальные события .NET очень тесно связаны с классом, в котором они определены.

В WPF кроме обработки событий приложение может взаимодействовать с пользователем с помощью команд. **Команды** представляют механизм выполнения какой-нибудь задачи, например, копирования текста - когда мы нажимаем Ctrl+C, то мы копируем текст в буффер. В процессе копирования выполняется ряд действий, и все вместе эти действия объединяются в одну команду. Использование команд помогает нам сократить объем кода и использовать одну и ту же команду для нескольких элементов управления в различных местах программы. Таким образом, команды позволяют абстрагировать набор действий от конкретных событий конкретных элементов.

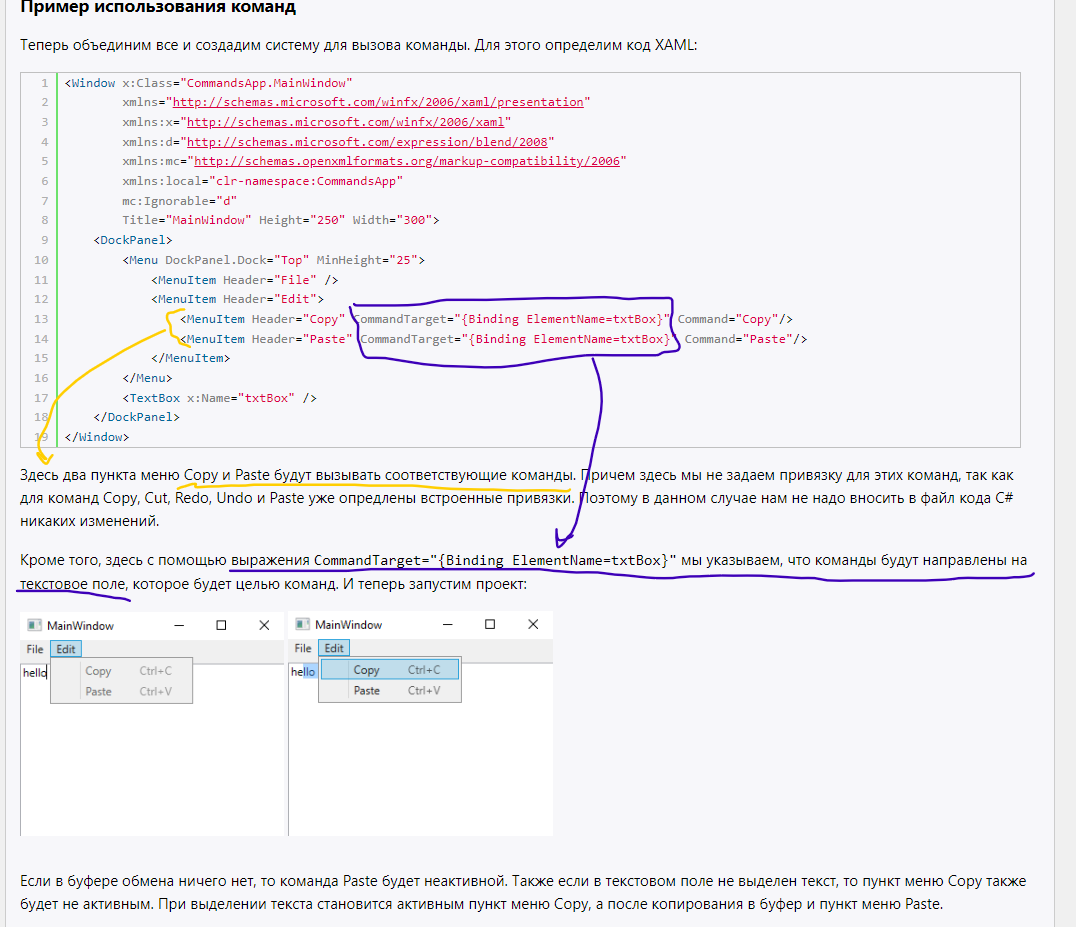
В некотором роде команды в WPF являются реализацией общераспространенного паттерна [Команда](http://metanit.com/sharp/patterns/3.3.php).

Модель команд в WPF состоит из четырех аспектов:

* Сама **команда**, которая представляем выполняемую задачу
* **Привязка команд**, которая связывает команду с определенной логикой приложения
* **Источник команды** - элемент пользовательского интерфейса, который запускает команду (например, кнопка, по нажатию который выполняется команда)
* **Цель команды** - элемент интерфейса, на котором выполняется команда

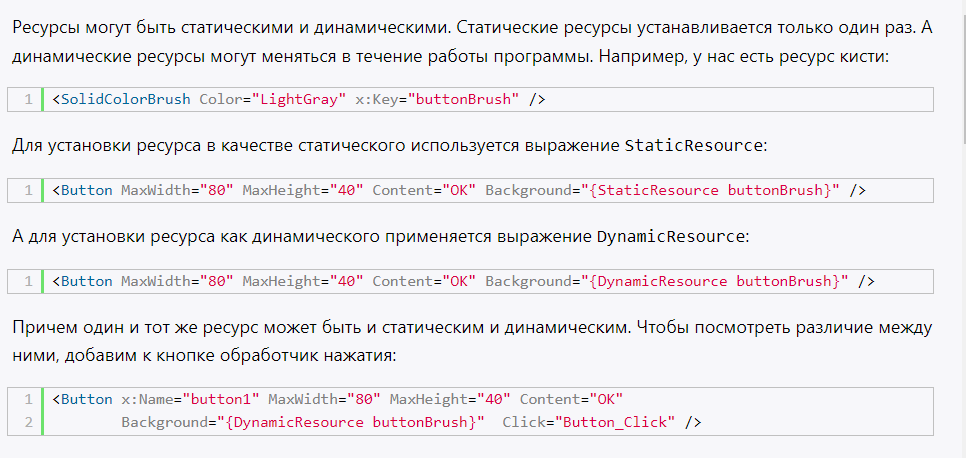






ЛАБА 8

**Лабораторная №8 Ресурсы, стили, триггеры, шаблоны**



1. **Для чего в WPF используются ресурсы? Каким образом можно определить новый ресурс и управлять им? Опишите назначение класса ResourceDictionary. Зачем каждый элемент имеет собственную коллекцию ресурсов?**

под ресурсами подразумеваются не дополнительные файлы (или **физические ресурсы**), как, например, аудиофайлы, файлы с изображениями, которые добавляются в проект. Здесь речь идет о **логических ресурсах**, которые могут представлять различные объекты - элементы управления, кисти, коллекции объектов и т.д. Логические ресурсы можно установить в коде XAML или в коде C# с помощью свойства Resources. Данное свойство опредлено в базовом классе **FrameworkElement**, поэтому его имеют большинство классов WPF.

В чем смысл использования ресурсов? Они повышают эффективность: мы можем определить один раз какой-либо ресурс и затем многократно использовать его в различных местах приложения. В связи с этим улучшается поддержка - если возникнет необходимость изменить ресурс, достаточно это сделать в одном месте, и изменения произойдут глобально в приложении.

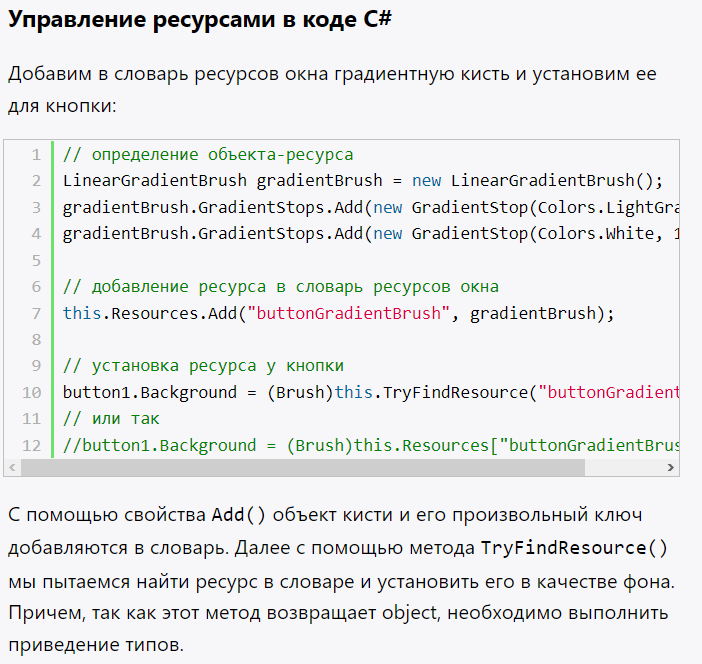
Свойство **Resources** представляет объект **ResourceDictionary** или словарь ресурсов, где каждый хранящийся ресурс имеет определенный ключ.

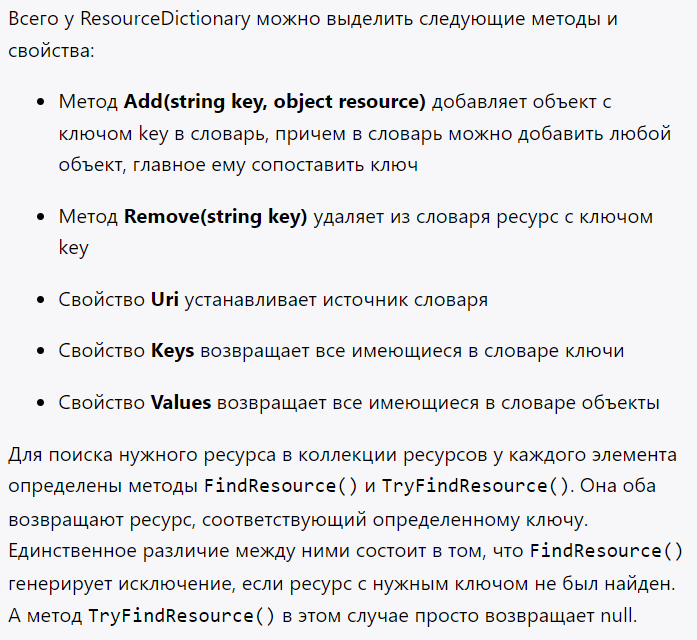
**Определение ресурсов:**

****

Здесь у окна определяются два ресурса: redStyle, который представляет объект SolidColorBrush, и gradientStyle, который представляет кисть с линейным градиентом. У кнопки определен один ресурс darkStyle, представляющий кисть SolidColorBrush. Причем каждый ресурс обязательно имеет свойство **x:Key**, которое и определяе ключ в словаре.

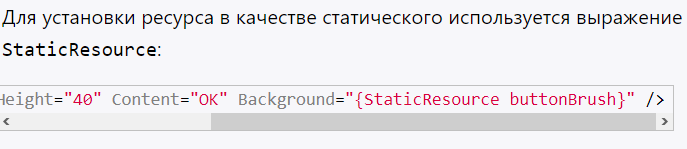
А в свойствах Background соответственно у грида и кнопки мы можем применить эти ресурсы: Background="{StaticResource gradientStyle}" - здесь после выражения StaticResource идет ключ применяемого ресурса.

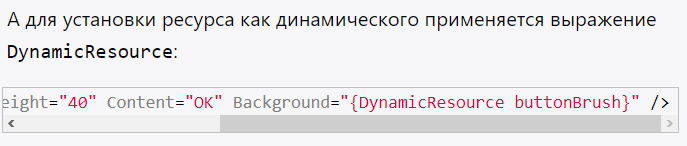
****

****

1. **Какая разница между статическими и динамическими ресурсами?**

**Статические** ресурсы устанавливается только один раз. А **динамические** ресурсы могут меняться в течение работы программы. Причем один и тот же ресурс может быть и статическим и динамическим.

****

****

В то же время надо отметить, что мы все равно может изменить статический ресурс - для этого нужно менять не сам объект по ключу, а его отдельные свойства.

1. **Что такое триггеры ? Для чего в WPF используются триггеры? Назовите основные типы триггеров.**

**Триггеры** позволяют декларативно задать некоторые действия, которые выполняются при изменении свойств стиля. Существует три вида триггеров:

* **Триггеры свойств**: вызываются в ответ на изменения свойствами зависимостей своего значения (Простые триггеры свойств задаются с помощью объекта **Trigger**. Они следят за значением свойств и в случае их изменения с помощью объекта **Setter** устанавливают значение других свойств.)
* **Триггеры данных**: вызываются в ответ на изменения значений любых свойств (они необязательно должны быть свойствами зависимостей) (Для соединения с отслеживаемыми свойства триггеры данных используют выражения привязки)



С помощью свойства Binding триггер данных устанавливает привязку к отслеживаемому свойству. Свойство **Value** задает значение отлеживаемого свойства, при котором сработает триггер

* **Триггеры событий**: вызываются в ответ на генерацию событий (Если простой триггер наблюдает за изменением свойства, то EventTrigger реагирует на определенные события совсем как обработчики событий. Правда, триггеры событий более ограничены в своих возможностях.)
* **Мультитриггеры**: вызываются при выполнении ряда условий (При необходимости отслеживания не одного, а сразу нескольких свойств используют. Он содержит коллекцию элементов **Condition**, каждый из которых, как и обычный триггер, определяет отслеживаемое свойство и его значение)

1. **Что такое локализация и как ее обеспечить.**

Локализация позволяет элементам управления изменяться в соответствии с текущими параметрами культуры в Windows. При локализации в WPF используются интерфейсы API в пространстве имен [System.Windows.Markup.Localizer](https://docs.microsoft.com/ru-RU/dotnet/api/system.windows.markup.localizer). Эти интерфейсы API поддерживают инструмент командной строки [LocBaml Tool Sample](https://github.com/microsoft/WPF-Samples/tree/master/Tools/LocBaml)

Существует несколько вариантов локализации приложения WPF. Например, можно привязать локализованные ресурсы в приложении к файлу XML, хранить локализуемый текст в resx-таблицах или же указать локализатору, использовать файлы XAML

В этом разделе описывается рабочий процесс локализации, использующий BAML-форму XAML, который предоставляет несколько преимуществ.

* Локализация может осуществляться после сборки.
* Можно выполнить обновление до более новой версии BAML-формы XAML с локализациями из старой версии BAML-формы XAML, чтобы обеспечить возможность локализации в процессе разработки.
* Можно проверять элементы и семантику исходного источника во время компиляции, так как BAML-форма XAML является скомпилированной формой XAML.

Процесс локализации начинается после сборки нелокализованного файла MyDialog.resources.dll. Элементы и свойства пользовательского интерфейса в исходном коде XAML извлекаются из BAML-формы XAML в пары "ключ — значение" с помощью интерфейсов API в [System.Windows.Markup.Localizer](https://docs.microsoft.com/ru-RU/dotnet/api/system.windows.markup.localizer). Локализаторы используют пары "ключ —значение" для локализации приложения. После завершения локализации можно создать файл .resource.dll на основе новых значений.

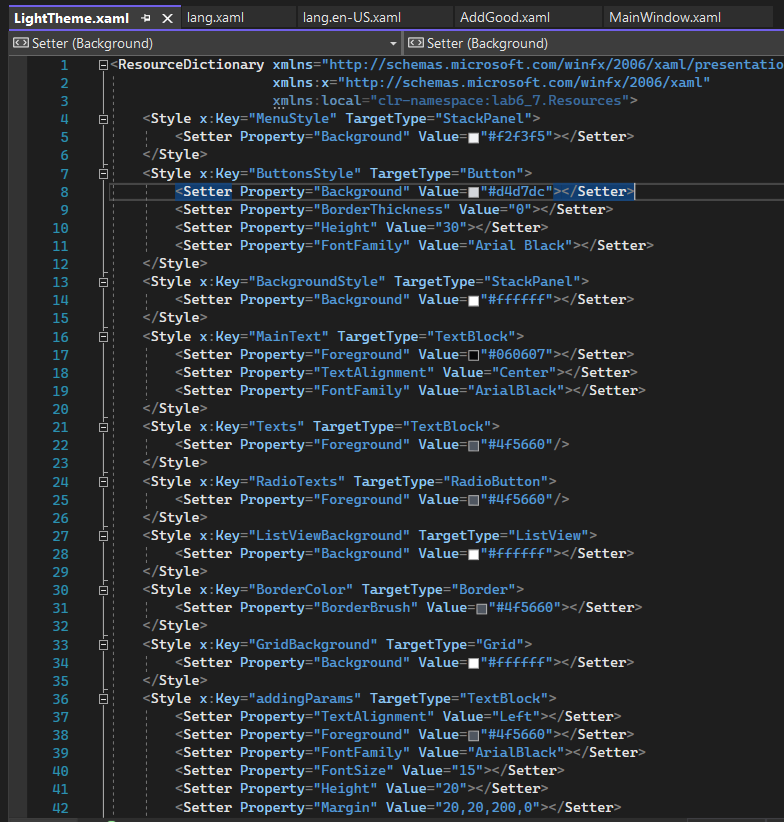
Ключи пар "ключ—значение" — это значения x:Uid, которые разработчик помещает в исходный пользовательский интерфейс после начала локализации локализатором. Изменения разработки можно объединить с уже завершенной работой локализации, чтобы минимальная работа по переводу была потеряна

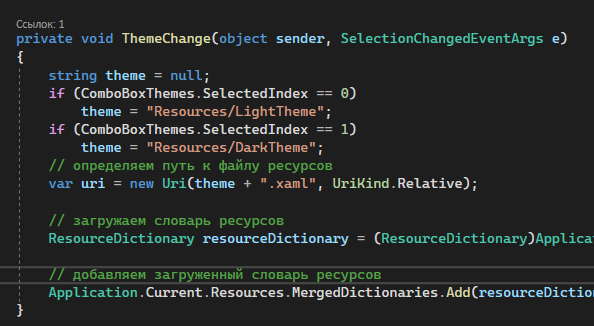
**https://professorweb.ru/my/WPF/UI\_WPF/level7/7\_11.php**

1. **Что такое тема? Опишите процесс создания темы на основе ресурсов и стилей.**

Стили позволяют задать стилевые особенности для определенного элемента или элементов одного типа. Но иногда возникает необходимость применить ко всем элементам какое-то общее стилевое единообразие. И в этом случае мы можем объединять стили элементов в темы.

Для этого добавляется в проект новый файл словаря ресурсов. Здесь указаны все те стили, которые применяются элементами окна. И добавляется еще один файл темы, для того, чтобы менять темы с помощью этих двух файлов (1 типо установлен по-умолчанию)

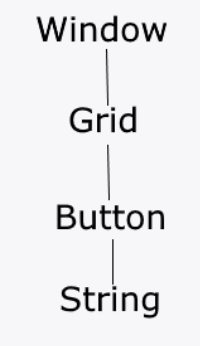
****

****

1. **Что такое шаблон и как его создать?**

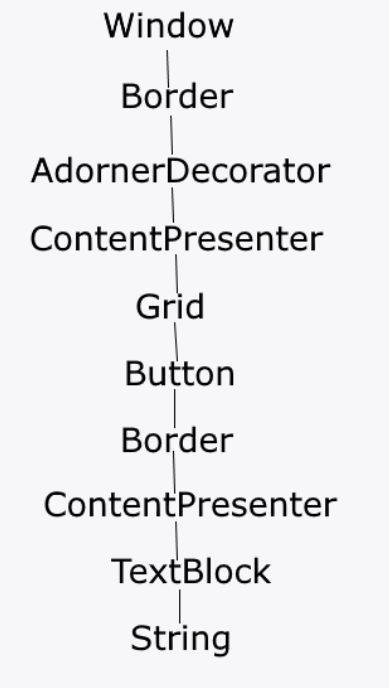
Несмотря на то, что стили существенно облегчают манипулирование внешним видом элементов управления, гораздо более сильным средством в плане визуализации являются **шаблоны**. В отличие от стилей **шаблоны** помогают полностью менять модель визуализации элемента

Визуализация в WPF тесно связана с такими понятиями как логическое и визуальное дерево. Эти деревья являются своего рода каркасом приложения.

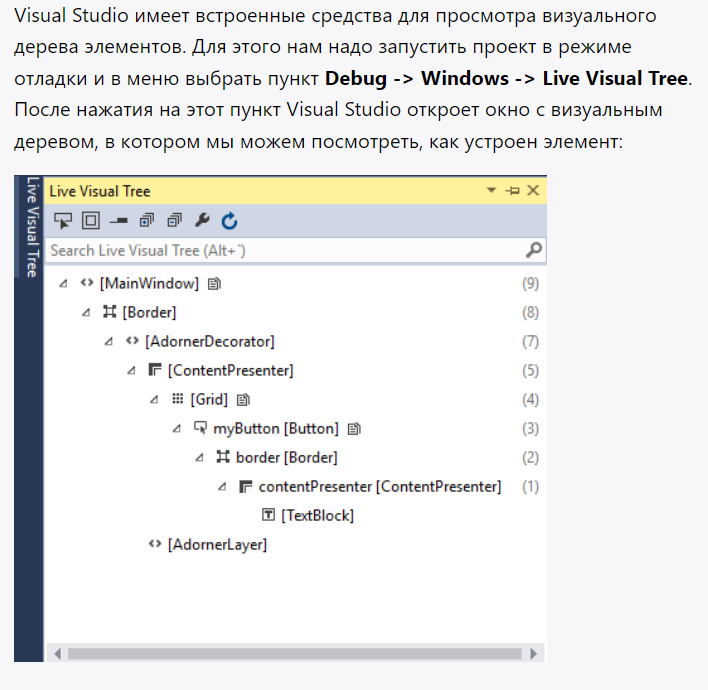
****

То есть в Window есть Grid, в Gride - элемент Button, в кнопке в качестве содержимого установлен некоторый текст в виде объекта String. В итоге получается некое дерево элементов, которое называется **логическим**. В WPF оно представлено классом **System.Windows.LogicalTreeHelper**. Логическое дерево имеет дело с визуализацией как таковой, оно образует модель доступа к дочерним элементам.

От него отличается **визуальное дерево**, представленное классом **System.Windows.Media.VisualTreeHelper**. Так, визуальное дерево для вышеприведенной разметки xaml будет выглядеть следующим образом

****

Визуальное дерево получается гораздо сложнее, оно показывает, как с визуальной точки зрения устроен элемент, из каких частей он состоит.

****

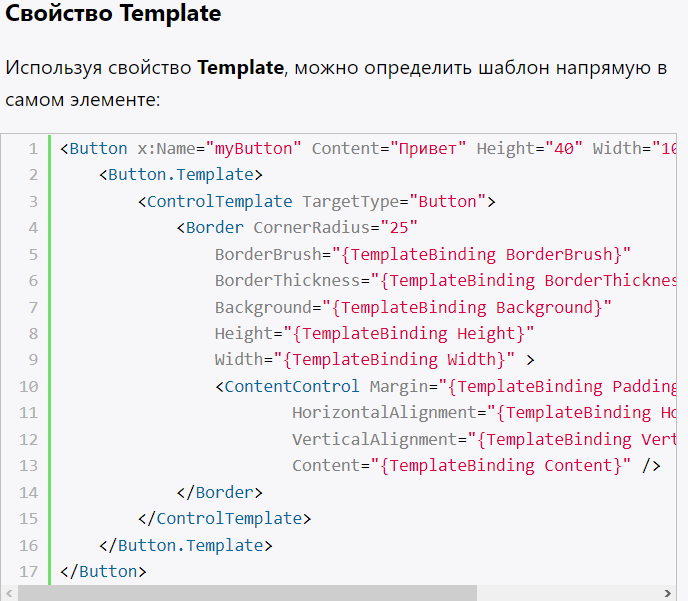
Все визуальные элементы в WPF уже имеют встроенные шаблоны, которые определяют визуальное дерево, структуру и даже поведение элементов. Однако мощь шаблонов состоит в том, что мы можем их переопределить по своему вкусу. Например, сделать круглое окно, а не квадратное, или кнопку в виде морской звезды.

Мы можем определять шаблоны с через стили, а можем в виде отдельных ресурсов.

****

Так, в данном случае с помощью элемента **ControlTemplate** определяется ресурс с ключом "btTemplate".

В ControlTemplate вложены элементы Border и ContentControl, которые через свои свойства определяют, как будет выглядеть кнопка.

****

1. **Зачем нужны пользовательские элементы управления? Как создать собственный элемент. Опишите члены класса UserControl.**

Хотя пользовательский элемент можно построить в любом проекте WPF, обычно такие элементы размещаются в специально выделенной сборке — библиотеке классов (DLL). Это позволяет разделять работу с множеством приложений WPF.

Чтобы гарантировать наличие всех необходимых ссылок на сборки и импорт всех нужных пространств имен, при создании приложения в Visual Studio в качестве типа проекта следует выбрать Custom Control Library (WPF) (Библиотека пользовательских элементов управления (WPF)). Внутри библиотеки классов можно создавать сколько угодно элементов управления.

Первый шаг в создании пользовательского элемента управления — это выбор корректного базового класса для наследования.

* **FrameworkElement**
* **Control**
* **ContentControl**
* **UserControl**
* **ItemsControl и Selector**
* **Panel**
* **Decorator**
* **Специфический класс элемента**

**UserControl**

Это элемент управления с содержимым, который может быть сконфигурирован с применением поверхности времени проектирования. Хотя такой пользовательский элемент управления не настолько отличается от обычного элемента управления с содержимым, обычно он используется тогда, когда необходимо быстро повторно применить неизменный блок пользовательского интерфейса в более чем одном окне (вместо создания действительно отдельного элемента управления, который может быть перенесен из одного приложения в другое)

1. **Что такое привязка данных? В чем разница между следующими режимами привязки: OneWay, TwoWay и OneTime.**

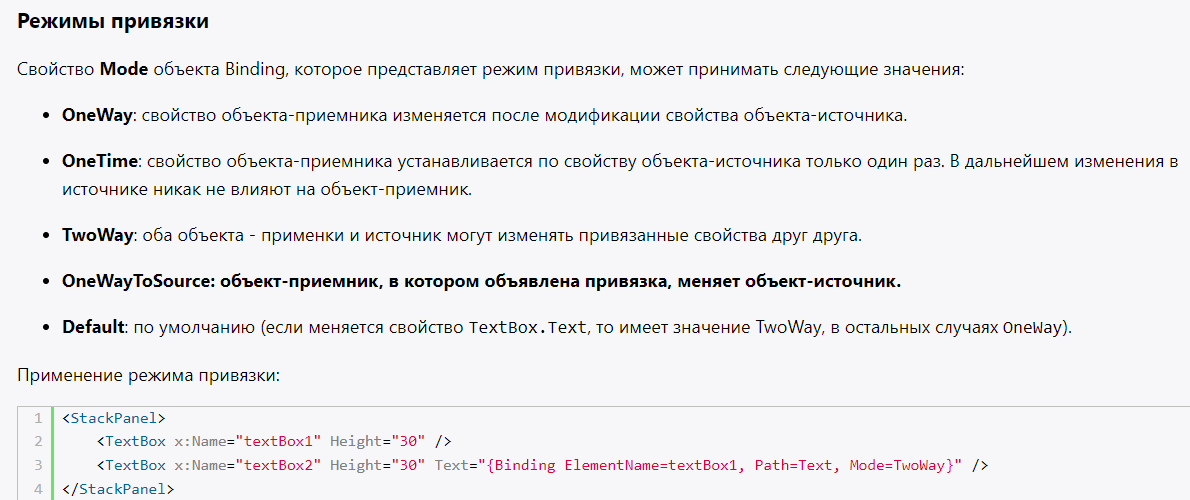
Большую роль при работе с данными играет механизм привязки/

Для создания привязки применяется элемент **Binding** и его свойства:

* **ElementName**: имя элемента, к которому идет привязка. Если мы говорим о привязке данных, то данное свойство задействуется редко за исключением тех случаев, когда данные определены в виде свойства в определенном элементе управления
* **Path**: ссылка на свойство объекта, к которому идет привязка
* **Source**: ссылка на источник данных, который не является элементом управления

Свойства элемента Binding помогают установить источник привязки. Для установки источника или контекста данных в элементах управления WPF предусмотрено свойство **DataContext**.





1. **Объясните назначение интерфейса INotifyPropertyChanged**

**Интерфейс** **INotifyPropertyChanged** используется для уведомления клиентов, как правило, привязки клиентов о том, что значение свойства изменилось

ЛАБА 9

1. Что такое Свойства зависимости? Для чего они нужны?

**свойства зависимостей** или **dependency property.**

Без свойств зависимостей были бы невозможны многие ключевые особенности WPF, как привязка данных, стили, анимация и т.д.

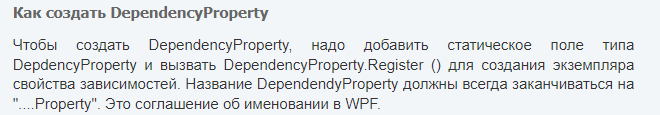
Главное отличие в том, что значения обычных .NET свойств можно читать прямо из закрытых членов вашего класса, в то время как значение DependencyProperties может динамически вызываться при вызове GetValue(), которые наследуется от DependencyObject.

Когда вы устанавливаете значение DepebdencyProperties, оно не хранится в поле вашего объекта, но хранится в словаре ваших ключей и значений, предоставленных DependencyObject базового класса. Ключ - это мя свойства, а значение - значение, которые вы хотите установить.

**Преимущества Dependency Properties:**  
Снижение памяти.  
Наследование значений.  
Уведомления об изменениях.  
Value resolution strategy

1. Как создать DependencyProperty? <https://metanit.com/sharp/wpf/13.php>

https://karpov-k.me/izuchaem-wpf/288-dependency-properties-v-wpf





WPF предоставляет два способа валидации значения свойства: **ValidateValueCallback** и **CoerceValueCallback**

3.Для чего и как используют делегат ValidateValueCallback?

**ValidateValueCallback**: делегат, который возвращает true, если значение проходит валидацию, и false - если не проходит

4.Для чего и как используют делегат CoerceValueCallback?

**CoerceValueCallback**: делегат, который может подкорректировать уже существующее значение свойства, если оно вдруг не попадает в диапазон допустимых значений

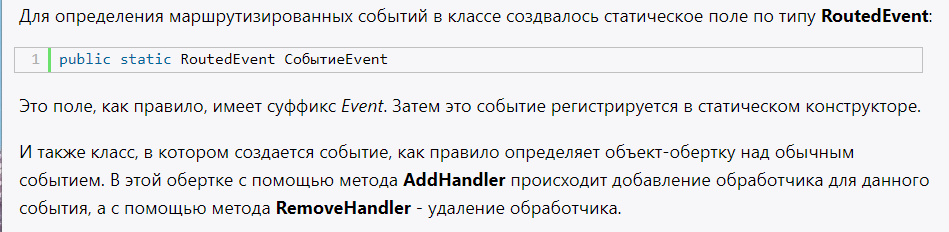
5.Какие типы маршрутизируемых событий есть в WPF (поясните каждый)?

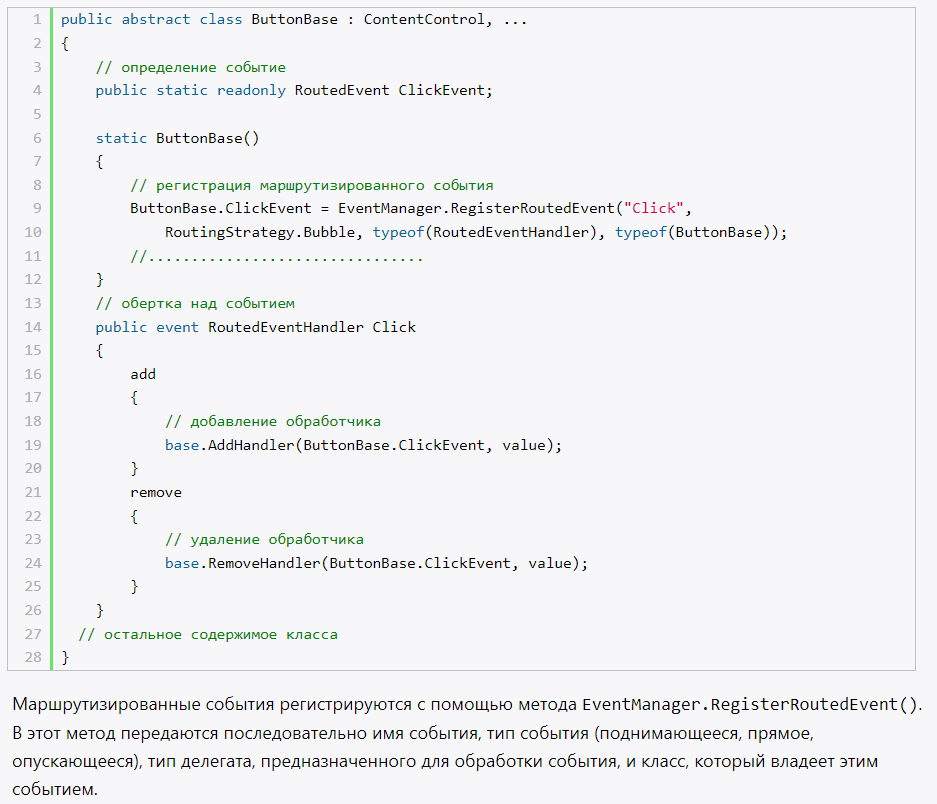
https://metanit.com/sharp/wpf/6.php

**Маршрутизируемые события делятся на три вида:**

* **Прямые** (direct events) - они возникают и отрабытывают на одном элементе и никуда дальше не передаются. Действуют как обычные события.
* **Поднимающиеся** (bubbling events) - возникают на одном элементе, а потом передаются дальше к родителю - элементу-контейнеру и далее, пока не достигнет наивысшего родителя в дереве элементов.
* **Опускающиеся, туннельные** (tunneling events) - начинает отрабатывать в корневом элементе окна приложения и идет далее по вложенным элементам, пока не достигнет элемента, вызвавшего это событие.

6.Как создать RoutedEvent**?**





7.Поясните концепцию Command в WPF? В чем ее преимущества?

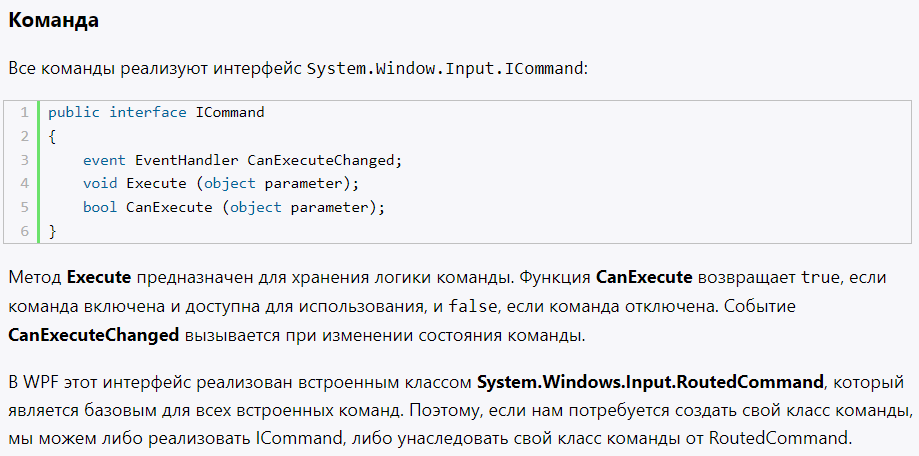
В WPF кроме обработки событий приложение может взаимодействовать с пользователем с помощью команд. **Команды** представляют механизм выполнения какой-нибудь задачи.

Использование команд помогает нам сократить объем кода и использовать одну и ту же команду для нескольких элементов управления в различных местах программы. Таким образом, команды позволяют абстрагировать набор действий от конкретных событий конкретных элементов.

Модель команд в WPF состоит из четырех аспектов:

* Сама **команда**, которая представляем выполняемую задачу
* **Привязка команд**, которая связывает команду с определенной логикой приложения
* **Источник команды** - элемент пользовательского интерфейса, который запускает команду (например, кнопка, по нажатию который выполняется команда)
* **Цель команды** - элемент интерфейса, на котором выполняется команда

1. Как используются команды?



9. Как создать RoutedUICommand?

WPF уже обладает большим набором встроенных команд. Все они представляют объекты класса **RoutedUICommand.**

семь групп встроенных команд:

