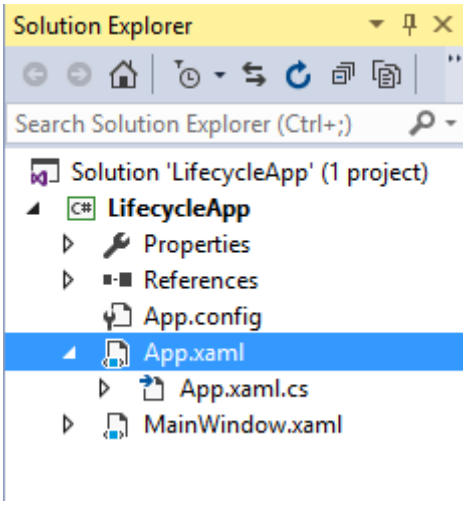
Приложение и класс Application

Класс Application

В предыдущих главах мы работали непосредственно с окном приложения, которое представлено классом Window, его разметкой, добавляли в нее элементы, создавали для него код на C#. Однако само приложение начинается не с класса **Window**, а с класса **Application**. По умолчанию при создании проекта WPF создается файл App.xaml и класс связанного кода App.xaml.cs:



Файл App.xaml выглядит примерно так:

```
<Application x:Class="LifecycleApp.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:LifecycleApp"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

С помощью атрибута x:Class элемент Application задает полное название производного класса приложения. По умолчанию класс называется **App**, с указанием названия проекта, то есть в данном случае **LifecycleApp.App**

Как правило, основная задача данного файла состоит в определении ресурсов, общих для приложения. Поэтому тут по умолчанию определен пустой элемент Application.Resources, в который, собственно, и помещаются ресурсы.

Также здесь декларативным путем можно прикрепить к событиям приложения обработчики.

С помощью атрибута **StartupUri** устанавливается путь к разметке xaml, с которого начинается выполнение приложения. По умолчанию это разметка окна MainWindow, определенного в файле MainWindow.xaml. Если у нас в приложении несколько окон, тут мы можем указать то, которое будет запускаться при вызове приложения.

файл App.xaml.cs также содержит определение класса App. По умолчанию этот класс совершенно пустой:

```
public partial class App : Application
{
}
```

В нем мы можем задать обработчики событий приложения, либо какую-то другую глобальную для всего приложения логику.

Таким образом, у нас получается, что оба файла App.xaml и App.xaml.cs содержат определение одного и тоже класса App. Однако в конечном счете они будут компилироваться в один файл приложения App.g.cs, который вы можете найти после компиляции приложения в каталоге проекта obj/Debug и который будет выглядеть примерно так:

```
public partial class App : System.Windows.Application {

    [System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuildTasks", "4.0.0.0")]
    public void InitializeComponent() {

        this.StartupUri = new System.Uri("MainWindow.xaml", System.UriKind.Relative);

    }

    [System.STAThreadAttribute()]
    [System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuildTasks", "4.0.0.0")]
    public static void Main() {
        LifecycleApp.App app = new LifecycleApp.App();
        app.InitializeComponent();
        app.Run();
    }
}
```

Входной точкой в программу (как и в любое другое приложение на C#) является метод Main, в котором создается экземпляр приложения и производится начальная инициализация и вызов главного окна программы.

Так как WPF требует, чтобы главный поток работал в однопоточном подразделении (Single-threaded apartment), то метод Main помечается атрибутом STAThreadAttribute. Однопоточное подразделение содержит один поток, в данном случае главный. Это означает, что к элементам, созданным в этом потоке можно обратиться только из этого же потока. В то же время WPF предлагает эффективный способ взаимодействия между потоками, о котором мы позже поговорим.

С помощью метода InitializeComponent происходит инициализация приложение: установка главного окна для запуска. И далее само приложение запускается через вызов app.Run().

Однако это не жестко заданная организация файлов приложения, и мы ее можем переопределить и установить точку входа в приложение сами. Для этого, во-первых, изменим код App.xaml.cs:

```
public partial class App : Application
{
    App()
    {
        InitializeComponent();
    }

    [STAThread]
    static void Main()
    {
        App app = new App();
        MainWindow window = new MainWindow();
        app.Run(window);
    }
}
```

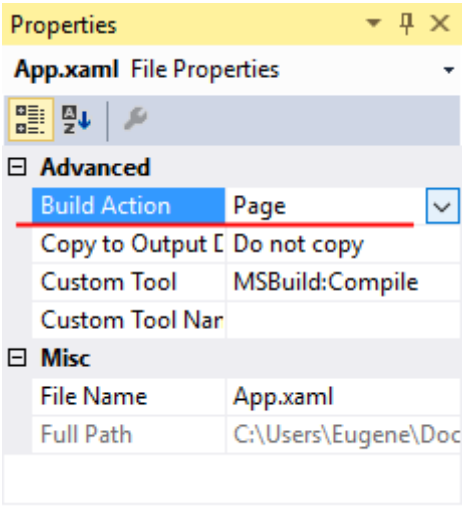
Фактически файл App.xaml.cs стал похож на компилируемый файл App.g.cs. Здесь мы сразу определяем метод Main и запускаем главное окно MainWindow.

После этого изменим файл App.xaml следующим образом:

```
<Application x:Class="LifecycleApp.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:LifecycleApp" >
    <Application.Resources>
        <Style TargetType="Grid">
            <Setter Property="Background" Value="Red" />
        </Style>
    </Application.Resources>
</Application>
```

В данном случае убран атрибут StartupUri, так как главное окно приложения теперь запускается из самого класса, и определен стиль для элемента Grid, который по умолчанию имеется в MainWindow.xaml.

Далее в окне свойств для файла App.xaml изменим значение в поле **Build Action** на **Page**:



И после запуска нам отобразится окно с гридом, окрашенным в красный цвет.

Работа с классом Application

Свойство ShutdownMode

Свойство ShutdownMode указывает на способ выхода из приложения и может принимать одно из следующих значений:

- OnMainWindowClose: приложение работает, пока открыто главное окно
- OnLastWindowClose: приложение работает, пока открыто хотя бы одно окно
- OnExplicitShutdown: приложение работает, пока не будет явно вызвано Application.Shutdown()

Задать свойство ShutdownMode можно в коде xaml:

```
<Application x:Class="LifecycleApp.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:LifecycleApp"
             StartupUri="MainWindow.xaml" ShutdownMode="OnLastWindowClose">
```

либо определить в App.xaml.cs:

```
public partial class App : Application
{
    public App()
    {
        this.ShutdownMode = ShutdownMode.OnLastWindowClose;
    }
}
```

События приложения

Класс Application определяет ряд событий, который могут использоваться для всего приложения:

- **Startup**: происходит после вызова метода Application.Run() и перед показом главного окна
- **Activated**: происходит, когда активизируется одно из окон приложения
- **Deactivated**: возникает при потере окном фокуса
- **SessionEnding**: происходит при завершении сеанса Windows при перезагрузке, выключении или выходе из системы текущего пользователя
- **DispatcherUnhandledException**: возникает при возникновении необработанных исключени
- **LoadCompleted**: возникает при завершении загрузки приложения
- **Exit**: возникает при выходе из приложения Это может быть закрытие главного или последнего окна, метод Application.Shutdown() или завершение сеанса

Так же, как и для элементов, обработчики события определяются для одноименных атрибутов в разметке xaml. Например, обработаем событие Startup:

```
<Application x:Class="LifecycleApp.App"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:LifecycleApp"
        StartupUri="MainWindow.xaml"  Startup="App_Startup">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Затем в файле связанного кода App.xaml.cs пропишем обработчик события:

```
public partial class App : Application
{
    //Запуск одной копии приложения
    System.Threading.Mutex mutex;
    private void App_Startup(object sender, StartupEventArgs e)
    {
        bool createdNew;
        string mutName = "Приложение";
        mutex = new System.Threading.Mutex(true, mutName, out createdNew);
        if (!createdNew)
        {
            this.Shutdown();
        }
    }
}
```

В результате при запуске каждой новой копии данного приложение обработчик события будет определять, запущена ли уже приложение, и если оно запущено, то данная копия завершит свою работу.

Создание заставки приложения

Приложения бывают разные, одни открываются быстро, другие не очень. И чтобы пользователя как-то известить о том, что идет загрузка приложения, нередко используют заставку или сплеш-скрин. В WPF простые заставки на основе изображений очень легко сделать. Для этого добавьте в проект какое-нибудь изображение и после добавления установите для него в окне Properties (Свойства) для свойства **BuildAction** значение **SplashScreen**. И после запуска до появления окна приложения на экране будет висеть изображение заставки.

Обращение к текущему приложению

Мы можем обратиться к текущему приложению из любой точки данного приложения. Это можно сделать с помощью свойства **Current**, определенном в классе Application. Так, изменим код нажатия кнопки на следующий:

```
private void Button1_Click(object sender, RoutedEventArgs e)
{
    foreach (Window window in Application.Current.Windows)
    {
        MessageBox.Show(window.ToString());
    }
}
```

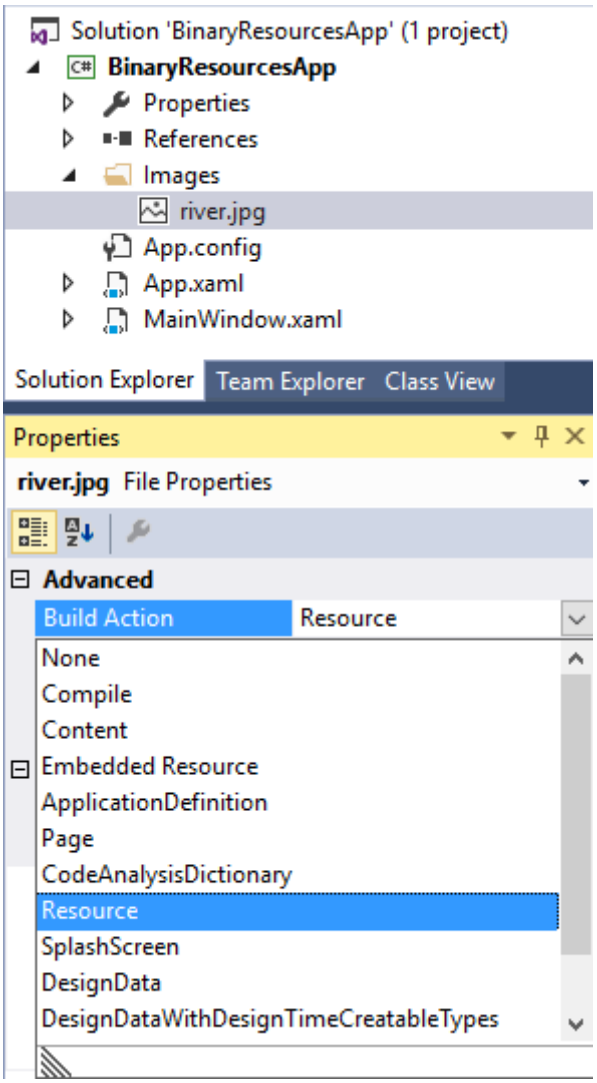
Таким образом, при нажатии на кнопку мы получим одно или несколько сообщений с именами всех окон данного приложения.

Работа с ресурсами приложения

Большинство приложений WPF так или иначе используют различные бинарные ресурсы - файлы изображений, мультимедиа и т.д. Для добавления ресурсов достаточно добавить уже существующий файл в проект. Нередко для оптимизации структуры проекта для хранения ресурсов создаются отдельные папки в проекте.

При добавлении ресурсов в проект мы можем установить одну из двух опций компиляции в окне свойств в поле **Build Action**:

- **Resources**: ресурс встраивается в сборку (значение по умолчанию)
- **Content**: ресурс физически находится отдельно от сборки, однако сборка содержит информацию о его местоположении



Доступ к ресурсам

Для доступа к ресурсам в WPF можно использовать объект Uri. Например, в проекте в папке Images помещен ресурс - файл river.jpg. И чтобы обратиться к нему из элемента Image мы можем использовать относительный путь Images/river.jpg:

```
<Image x:Name="myImage" Source="Images/river.jpg" />
```

Для доступа к ресурсу здесь применяется относительный путь с указанием папки и названия файла. Однако в реальности все используемые в xaml пути трансформируются в объекты Uri. Так, если бы нам пришлось устанавливать изображение в коде c#, то нам пришлось бы написать что-то вроде следующего:

```
myImage.Source = new BitmapImage(new Uri("Images/river.jpg", UriKind.Relative));
```

Конструктор Uri принимает два параметра: собственно относительный путь и значение из перечисления UriKind, которое указывает, что путь стоит расценивать как относительный.

Кроме относительных путей мы также можем использовать абсолютные:

```
myImage.Source = new BitmapImage(new Uri(D://forest.jpg, UriKind.Absolute));
```

Кроме относительных и абсолютных путей в WPF можно использовать определения "упакованных" URI (packageURI). Фактически относительные пути являются сокращениями упакованных Uri. Так путь "Images/river.jpg" является сокращением пути "pack://application:,,,/images/river.jpg":

```
myImage.Source = new BitmapImage(new Uri("pack://application:,,,/images/river.jpg"));
```

Аналог в xaml:

```
<Image x:Name="myImage" Source="pack://application:,,,/images/river.jpg" />
``
Синтаксис упакованных URI заимствован из спецификации стандарта XML Paper Specification (XPS). Д
```

pack://application:,,,/[относительный _путь_к_ресурсу]

```
Три запятых здесь фактически передают три слеша. То есть по факту начало пути выглядит так: application:///

Если же наши ресурсы находятся вне приложения, например, это некий файл по пути D://forest.jpg, тогда вместо appl

```Csharp
myImage.Source = new BitmapImage(new Uri("pack://D:,,,forest.jpg"));
```

Ресурсы в других сборках

Может сложиться ситуация, когда наше приложение использует внешнюю сборку, в которой были упакованы ресурсы. И мы также можем обратиться к этим ресурсам, используя синтаксис "упакованных" Uri. В этом случае путь будет иметь следующий формат:

```
pack://application:,,,/[название_библиотеки];component/[путь_к_ресурсу]
```

Допустим, файл сборки называется libs.dll, а ресурс, а сам ресурс в ней находится по пути images/forest.jpg, тогда получить его мы сможем следующим образом:
```

```
myImage.Source = new BitmapImage(new Uri("pack://application:,,,/libs;component/images/river.jpg
```

Либо можно использовать эквивалентный способ доступа к ресурсу:

```
myImage.Source = new BitmapImage(new Uri("libs;component/images/river.jpg", UriKind.Relative));
```

Получение ресурсов

Используя выше рассмотренные пути можно в программе получать сам файл ресурса и производить с ним манипуляции. Для получения ресурса приложения применяется метод **Application.GetResourceStream()**, например:

```
System.Windows.Resources.StreamResourceInfo res =
    Application.GetResourceStream(new Uri("images/river.jpg", UriKind.Relative));

res.Stream.CopyTo(new System.IO.FileStream("D://newRiver.jpg", System.IO.FileMode.OpenOrCreate))
```

В данном случае мы считываем ресурс приложения и выполняем его копирование по новому пути вне приложения.