# Графический интерфейс в Xamarin Forms. Контейнеры компановки.

В Xamarin.Forms визуальный интерфейс состоит из страниц. Страница представляет собой объект класса **Page**, она занимает все пространство экрана. То есть то, что мы видим на экране мобильного устройства - это страница. Приложение может иметь одну или несколько страниц.

Страница в качестве содержимого принимает один из контейнеров компоновки, в который в свою очередь помещаются стандартные визуальные элементы типа кнопок и текстовых полей, а также другие элементы компоновки.

По умолчанию весь интерфейс создается в классе App, который располагается в файле **App.xaml.cs** и который представляет текущее приложение:

```
public partial class App : Application
  {
      public App()
      {
         InitializeComponent();
         MainPage = new MainPage();
      }
      protected override void OnStart()
      {
      }
      protected override void OnSleep()
      {
      }
      protected override void OnResume()
      {
      }
   }
```

Работа класса App начинается с конструктора, где сначала вызывается метод **InitializeComponent()**, который выполняет инициализацию объекта, а потом устанавливается свойство **MainPage**.

Через это свойство класс App устанавливает главную страницу приложения. В данном случае она определяется классом HelloApp.MainPage, то есть тем классом, который определен в файлах MainPage.xaml и MainPage.xaml.cs.

Xamarin.Forms позволяет создавать визуальный интерфейс как с помощью кода C#, так и декларативным путем с помощью языка xaml, аналогично html, либо комбинируя эти подходы.

# Создание интерфейса из кода C#

1. Добавить в проект HelloApp обычный класс на языке C#, например **StartPage**.

Определить в нем содержимое:

```
class StartPage : ContentPage
    {
        public StartPage()
        {
            Label header = new Label() { Text =
"Привет из Xamarin Forms" };
            this.Content = header;
        }
    }
```

Чтобы обозначить StartPage в качестве стартовой страницы, нужно изменить класс App:

2. В Visual Studio есть готовый шаблон для добавления новых классов страниц с простейшим кодом. Так, чтобы добавить новую страницу, надо при добавлении нового элемента выбрать шаблон **Content Page (C#)**. Добавленный класс страницы будет иметь следующий код:

```
namespace HelloApp
{
    public class Page1 : ContentPage
    {
        public Page1()
        {
            Content = new StackLayout
            {
                Children = { new Label { Text = "Hello Page" };
                }
            }
        }
    }
```

} И также в классе приложения мы можем установить эту страницу в качестве стартовой

# XAML

Более предпочтительным способом является создание интерфейса, не в коде, а его описание в XAML.

XAML представляет язык разметки на основе xml для создания объектов декларативным образом. Собственно поэтому при создании проекта уже по умолчанию в него добавляются два файла **MainPage.xaml** и **MainPage.xaml.cs**.

# Использование XAML несет некоторые преимущества

1. С помощью XAML можно отделить графический интерфейс от логики приложения, благодаря чему над разными частями приложения могут относительно автономно работать разные специалисты: над интерфейсом - дизайнеры, над кодом логики - программисты.
2. XAML позволяет описать интерфейс более ясным и понятным способом, такой код гораздо проще поддерживать и обновлять.
3. В целом XAML позволяет организовать весь пользовательский интерфейс в виде набора страниц подобно тому, как это делается в HTML.

# Структура файла XAML

Файл с разметкой на xaml представляет собой обычный файл xml, и первой строкой идет стандартное определение xml-файла.

```
<?xml version="1.0" encoding="utf-8" ?>
```

Далее здесь определен элемент ContentPage, который представляет страницу.

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        x:Class="HelloApp.MainPage">
```

В определении корневого элемента ContentPage подключаются два пространства имен с помощью атрибутов **xmlns**.

Пространство имен **http://xamarin.com/schemas/2014/forms** определяет большинство типов из Xamarin Forms, которые применяются для построения графического интерфейса.

Второе пространство имен **http://schemas.microsoft.com/winfx/2009/xaml** определяет ряд типов XAML и типы CLR. Так как только одно пространство имен может быть базовым, то это пространство используется с префиксом (или проецируется на префикс) x: xmlns:x. Это значит, что те свойства элементов, которые заключены в этом пространстве имен, будут использоваться с префиксом x - x:Name или x:Class

После подключения пространств имен идет атрибут x:Class="HelloApp.MainPage, который указывает на класс, представляющий данную страницу.

Далее внутри ContentPage определены непосредственно элементы, которые и будут представлять графический интерфейс.

В данном случае в ContentPage определен контейнер компоновки StackLayout, который по умолчанию располагает вложенные элементы в столбик. А в элементе StackLayout расположен элементы Frame и Label:

```
<StackLayout>
    <Frame BackgroundColor="#2196F3" Padding="24" CornerRadius="0">
        <Label Text="Welcome to Xamarin.Forms!" HorizontalTextAlignment="Center" TextColor="White" FontSize="36"/>
    </Frame>
    <Label Text="Start developing now" FontSize="Title" Padding="30,10,30,10"/>
    <Label Text="Make changes to your XAML file and save to see your UI update in the running app with XAML Hot Reload. Give it a try!" FontSize="16" Padding="30,0,30,0"/>
    <Label FontSize="16" Padding="30,24,30,0">
        <Label.FormattedText>
            <FormattedString>
                <FormattedString.Spans>
                    <Span Text="Learn more at "/>
                    <Span Text="https://aka.ms/xamarin-quickstart" FontAttributes="Bold"/>
                </FormattedString.Spans>
            </FormattedString>
        </Label.FormattedText>
    </Label>
</StackLayout>
```

# Элементы в XAML и их атрибуты

XAML предлагает очень простую и ясную схему определения различных элементов и их свойств. Каждый элемент, как и любой элемент XML, должен иметь открытый и закрытый тег:

<Label></Label>

Либо элемент может иметь сокращенную форму с закрывающим слешем в конце, наподобие:

<Label />

Каждый элемент в XAML представляет объект определенного класса C#, а атрибуты элементов соотносятся со свойствами этих классом.

Например, элемент Label фактически будет представлять объект одноименного класса.

Свойства классов могут принимать значения различных типов: string, double, int и т.д. Но в XAML атрибуты элементов имеют текстовые значения. Например:

```
<Label Text="Welcome to Xamarin.Forms!"
HorizontalTextAlignment="Center"
TextColor="White" FontSize="36" />
```

Дело в том, что в Xamarin Forms действуют конвертеры типов, которые позволяют преобразовать от одного типа к другому.

# Сложные свойства

Кроме простых свойств, которые могут устанавливаться с помощью простой строки, например,

<Label Text="Welcome to Xamarin.Forms!" />,

в XAML могут применяться сложные или комплексные свойства.

В таких случаях свойство класса может принимать в качестве значения какой-нибудь сложный объект, который в свою очередь имеет некоторый набор свойств.

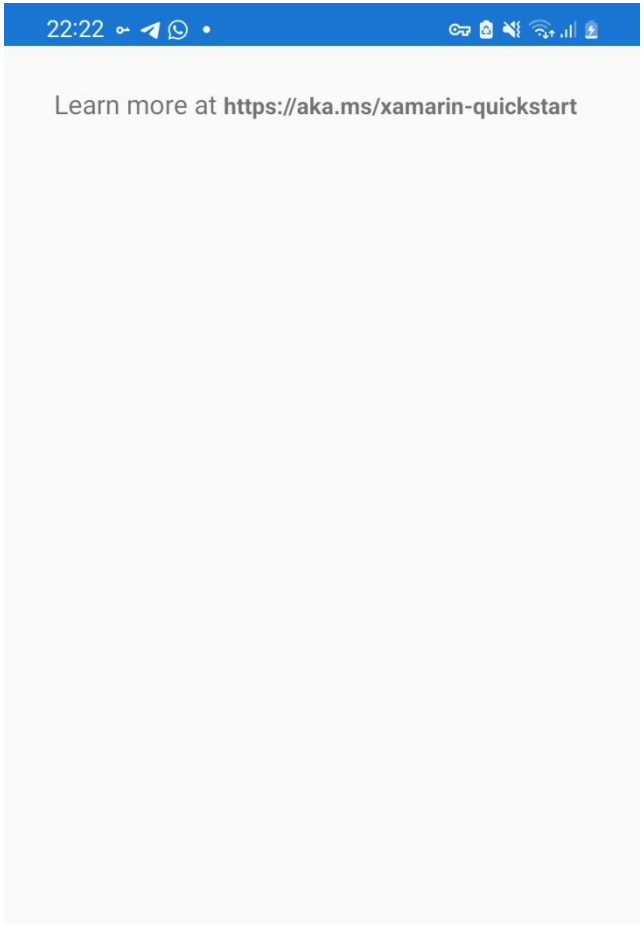
```
<Элемент.Свойство>
    <Сложный_объект />
</Элемент.Свойство>
```

Например:

```
<Label FontSize="16" Padding="30,24,30,0">
    <Label.FormattedText>
        <FormattedString>
            <FormattedString.Spans>
                <Span Text="Learn more at "/>
                <Span Text="https://aka.ms/xamarin-quickstart"
FontAttributes="Bold"/>
            </FormattedString.Spans>
        </FormattedString>
    </Label.FormattedText>
</Label>
```

Свойство **FormattedText** представляет форматированный текст (текст со сложным оформлением) и является сложным свойством. В качестве значения оно принимает объект **FormattedString**, который передается в качестве вложенного объекта.

У объекта **FormattedString**, в свою очередь, имеется свойство **Spans**, которое также является сложным свойством и которое представляет набор элементов **Span** - отдельных кусочков текста. А у каждого элемента Span устанавливается атрибут Text, который задает выводимый текст.



# Взаимодействие XAML и C#

При добавлении новой страницы XAML в проект также одновременно добавляется файл кода C#. Так, при создании проекта в него по умолчанию добавляется файл с графическим интерфейсом в XAML - **MainPage.xaml** и файл **MainPage.xaml.cs**, где, как предполагается, должна находится логика приложения, связанная с разметкой из файла XAML.

Файлы XAML позволяют нам определить интерфейс окна, но для создания логики приложения, например, для определения обработчиков событий элементов управления, нам все равно придется воспользоваться кодом C#.

По умолчанию в файле XAML определен атрибут **x:Class**:

x:Class="HelloApp.MainPage”

Он указывает на класс, который будет представлять данную страницу и в который будет компилироваться код в XAML при компиляции. То есть во время компиляции будет генерироваться класс HelloApp.MainPage, унаследованный от класса ContentPage.

В файле кода **MainPage.xaml.cs**, можно найти класс с тем же именем.

Этот практически пустой класс уже выполняет некоторую работу. Во время компиляции этот класс объединяется с классом, сгенерированным из кода XAML.

Чтобы такое слияние классов во время компиляции произошло, класс MainPage определяется как частичный с модификатором partial.

А через метод InitializeComponent() класс MainPage вызывает скомпилированный ранее код XAML, разбирает его и по нему строит графический интерфейс страницы.

**Пример!**

Определить на станице **MainPage.xaml** кнопку:

```
<Button x:Name="button1" Text="Нажать!" Clicked="Button_Click" />
```

Свойства элементов определяются в виде атрибутов, например, Text="Нажать!". События также определяются как атрибуты.

Например, с помощью атрибута **Clicked** устанавливается обработчик для события нажатия: Clicked="Button_Click".

Чтобы определить обработчик, нужно перейти в файл **MainPage.xaml.cs** и определить в классе MainPage следующий метод:

```
private void Button_Click(object sender, EventArgs e)
    {
        button1.Text = "Нажато!!!";
    }
```

С помощью атрибута **x:Name** элементу назначается имя. При компиляции приложения будет создаваться приватная переменная с этим именем. Через это имя в файле отделенного кода C# мы сможем ссылаться на этот элемент, точнее на объект, который представляет в файле кода данный элемент. В частности, здесь кнопке назначено имя "button1". Соответственно в файле кода мы можем обратиться к свойствам и методам кнопки через это имя. В итоге по нажатию на кнопку у нее изменится текст.

# Расширения разметки XAML???

Расширения разметки (Markup extensions) дают большую гибкость над тем, как устанавливать значения атрибутов в XAML.

# Контейнеры компоновки

Все элементы Xamarin унаследованы от общего класса **View** и поэтому наследуют ряд общих свойств.

Для определения содержимого страницы класс страницы **ContentPage** имеет свойство **Content**.

Но свойство Content имеет ограничение - для него можно установить только один элемент. И чтобы помещать на страницу сразу несколько элементов, нам надо исользовать один из элементов компоновки.

Элемент компоновки представляет класс, который наследуется от базового класса **Layout<T>**

- **StackLayout**
- **AbsoluteLayout**
- **RelativeLayout**
- **Grid**
- **FlexLayout**

Все элементы компоновки имеют свойство **Children**, позволяющее задать или получить вложенные элементы.

```
Label label1 = new Label() { Text = "Первая метка" };
Label label2 = new Label() { Text = "Вторая метка" };
        StackLayout stackLayout = new
StackLayout()
        {
            Children = {label1, label2}
        };
         this.Content = stackLayout;
```

# Аналогично в XAML можно написать

```
<StackLayout x:Name="stackLayout">
    <StackLayout.Children>
      <Label Text="Первая метка" />
      <Label Text="Вторая метка" />
    </StackLayout.Children>
  </StackLayout>
```

Или сократить

```
<StackLayout x:Name="stackLayout">
    <Label Text="Первая метка" />
    <Label Text="Вторая метка" />
</StackLayout>
```

Поскольку коллекция Children представляет собой обычный список, то он поддерживает операции по управлению элементами. В частности, мы можем динамически добавить новые элементы. Например:

```
stackLayout.Children.Add(new Label { Text = "Третья метка" });
```

Тот же принцип работы с элементами будет характерен и для других элементов компоновки.

# StackLayout и ScrollView

StackLayout определяет размещение элементов в виде горизонтального или вертикального стека. Для позиционирования элементов он определяет два свойства:

- **Orientation**: определяет ориентацию стека - вертикальный или горизонтальный
- **Spacing**: устанавливает пространство между элементами в стеке, по молчанию равно 6 единицам

```
public MainPage()
    {
        Label label1 = new Label()
        {
            Text = "Первая метка",
            TextColor = Color.Red
        };
        Label label2 = new Label()
        {
            Text = "Вторая метка",
            TextColor = Color.Blue
        };
        Label label3 = new Label()
        {
            Text = "Третья метка",
            TextColor = Color.Green
        };

        StackLayout stackLayout = new
StackLayout()
        {
            Children = {label1, label2, label3}
        };
        stackLayout.Spacing = 8;
        this.Content = stackLayout;
    }
```

# Аналог в xaml:

```
<StackLayout x:Name="stackLayout"
Spacing="8">
    <Label Text="Первая метка"
TextColor="Red"  />
    <Label Text="Вторая метка"
TextColor="Blue" />
    <Label Text="Третья метка"
TextColor="Green" />
  </StackLayout>
```

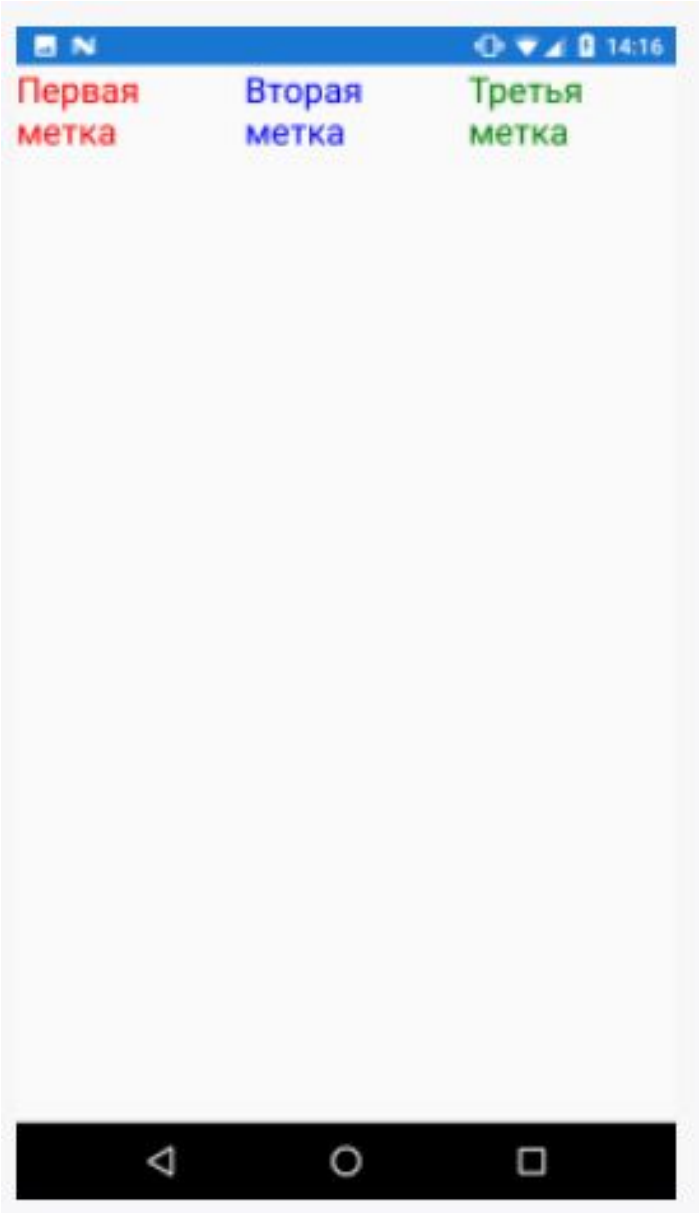

14:07
Первая метка
Вторая метка
Третья метка

По умолчанию создается вертикальный стек элементов. Чтобы создать горизонтальный стек, надо присвоить свойству Orientation значение StackOrientation.Horizontal:

```
stackLayout.Orientation = StackOrientation.Horizontal;
```

StackLayout - обычный элемент, у которого могут использоваться все стандартные свойства типа настройки цвета, отступы и т.д.

```
<StackLayout x:Name="stackLayout" Spacing="8" Orientation="Horizontal">
  <Label Text="Первая метка" TextColor="Red"  />
  <Label Text="Вторая метка" TextColor="Blue" />
  <Label Text="Третья метка" TextColor="Green" />
</StackLayout>
```


14:16
Первая
метка
Вторая
метка
Третья
метка

# ScrollView

При создании стека или любого другого элемента компоновки может сложиться ситуация, когда не все элементы будут помещаться на экране. В этом случае необходимо создать прокрутку с помощью элемента **ScrollView**:

```
StackLayout stackLayout = new StackLayout();
    for (int i = 1; i < 20; i++)
    {
        Label label = new Label
        {
            Text = "Метка " + i,
            FontSize = 23
        };
        stackLayout.Children.Add(label);
    }
    ScrollView scrollView = new ScrollView();
    scrollView.Content = stackLayout;
    this.Content = scrollView;
```

Или в xaml:

```
<ScrollView>
    <StackLayout>
      <Label Text="Метка 1" FontSize="23" />
      <Label Text="Метка 2" FontSize="23" />
      <!--.....-->
      <Label Text="Метка 20" FontSize="23" />
    </StackLayout>
  </ScrollView>
```

# AbsoluteLayout

Позволяет задавать вложенным элементам абсолютные координаты расположения на странице и подходит больше для тех случаев, когда нам нужны элементы с точными координатами.

Для создания абсолютного позиционирования нам надо определить прямоугольную область, которую будет занимать элемент. Как правило, для этого используется структура Rectangle, которая представляет прямоугольник:

```
int x = 10; // позиция координаты X на странице
int y = 10; // позиция координаты Y на странице
int width = 100; // ширина блока элемента
int height = 80;    // высота блока элемента
Rectangle rect = new Rectangle(x, y, width, height);
```

Если неизвестна точная ширина и длина элемента, то можно ограничиться позицией, с которой начинается элемент, в виде структуры **Point**:

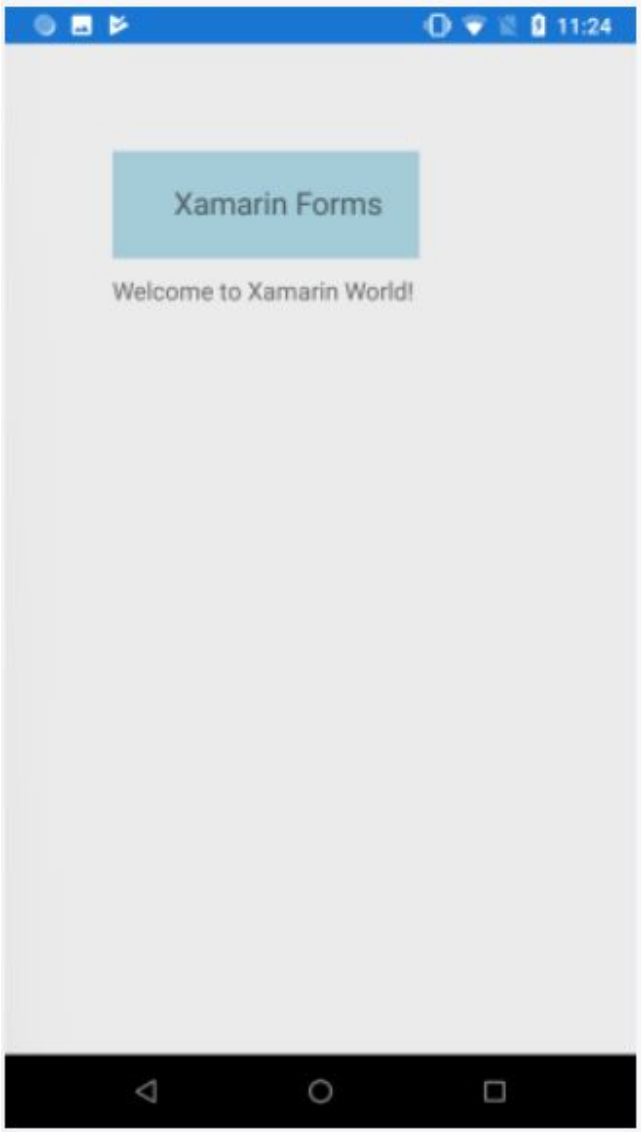
```
int x = 10; // позиция координаты X на странице
int y = 10; // позиция координаты Y на странице
Point point = new Point(x, y);
```

Для добавления элементов в контейнер AbsoluteLayout мы можем использовать метод AbsoluteLayout.Children.Add(), который имеет ряд версий:

- Add(View view): просто добавляет элемент в контейнер
- Add(View view, Point point): элемент будет располагаться на странице, начиная с точки point
- Add(View view, Rectangle rect): элемент будет располагаться на странице в области, ограниченной прямоугольником rect

```
AbsoluteLayout absoluteLayout = new
AbsoluteLayout();
absoluteLayout.Children.Add(
        new BoxView { Color = Color.LightBlue },
        new Rectangle(70, 70, 200, 70)
    );

    absoluteLayout.Children.Add(
        new Label { Text = "Xamarin Forms",
FontSize = 20},
        new Rectangle(110, 90, 150, 60)
    );
    absoluteLayout.Children.Add(
        new Label { Text = "Welcome to Xamarin
World!", FontSize = 16 },
        new Point(70, 150)
    );
    Content = absoluteLayout;
```

В данном случае одна метка накладывается на элемент BoxView, который представляет прямоугольник, заполненный цветом. Таким образом, AbsoluteLayout позволяет нам осуществлять перекрытие одних элементов другими.



Для установки позиции в качестве альтернативы можно использовать статический метод **AbsoluteLayout.SetLayoutBounds()**, который в качестве параметров принимает элемент и прямоугольную область, выделяемую для этого элемента:

```
        BoxView boxView = new BoxView { Color = Color.LightBlue };
        Label headerLbl = new Label { Text = "Xamarin Forms", FontSize = 20 };
        Label contentLbl = new Label { Text = "Welcome to Xamarin World!",
FontSize = 16 };

        // определяем позицию и размеры для BoxView
        AbsoluteLayout.SetLayoutBounds(boxView, new Rectangle(70, 70, 200,
70));
        // определяем позицию и размеры для первой метки
        AbsoluteLayout.SetLayoutBounds(headerLbl, new Rectangle(110, 90,
150, 60));
        // определяем позицию и размеры для второй метки
        AbsoluteLayout.SetLayoutBounds(contentLbl, new Rectangle(70, 150,
AbsoluteLayout.AutoSize, AbsoluteLayout.AutoSize));

        AbsoluteLayout absoluteLayout = new AbsoluteLayout()
        {
            Children = { boxView, headerLbl, contentLbl }
        };

        Content = absoluteLayout;
```

Для задания автоматических ширины и длины в данном случае можно использовать значение **AbsoluteLayout.AutoSize**. Поэтому для последней метки устанавливается только точка верхнего левого угла занимаемой области, а высота и ширина этой области рассчитываются автоматически исходя из ширины и высоты элемента.

Для создания абсолютного позиционирования в XAML у элемента устанавливается свойство **AbsoluteLayout.LayoutBounds**, которое фактически представляет прямоугольную область:

```
<AbsoluteLayout>
    <BoxView Color="LightBlue" AbsoluteLayout.LayoutBounds="70, 70, 200, 70" />
    <Label Text="Заголовок" FontSize="Large" AbsoluteLayout.LayoutBounds="110, 90, 150, 60" />
    <Label Text="Основное содержание текста" FontSize="Medium" AbsoluteLayout.LayoutBounds="70, 150, AutoSize, AutoSize" />
</AbsoluteLayout>
```

В случае со второй меткой неизвестная точная ширина и длина, поэтому вместо конкретных размеров можно использовать автоматические размеры в виде значения AutoSize.

# Флаг AbsoluteLayoutFlags. Пропорциональные значения

Часто бывает сложно подобрать точные размеры для элементов. И в этом случае можно установить пропорциональные размеры. Для этого используется перечисление **AbsoluteLayoutFlags**, которое может принимать следующие значения:

- None: все значения интерпретируются как абсолютные
- All: все значения интерпретируются как пропорциональные
- WidthProportional: пропорциональной считается только ширина прямоугольной области элемента, а все остальные значения рассматриваются как абсолютные
- HeightProportional: пропорциональной считается только высота прямоугольной области элемента, а все остальные значения рассматриваются как абсолютные
- XProportional: пропорциональной считается только координата X элемента, а все остальные значения рассматриваются как абсолютные
- YProportional: пропорциональной считается только координата Y элемента, а все остальные значения рассматриваются как абсолютные
- PositionProportional: пропорциональными считаются только координаты X и Y позиции элемента
- SizeProportional: пропорциональными считаются только ширина и высота прямоугольной области элемента

При использовании пропорциональных значений все эти значения рассчитываются в диапазоне от 0.0 до 1.0. Например, если нужно, чтобы элемент занимал половину ширины контейнера и четверть высоты, то для ширины нужно установить значение 0.5, а для высоты - 0.25.

Таким образом, ширина элемента будет вычисляться по следующей формуле:

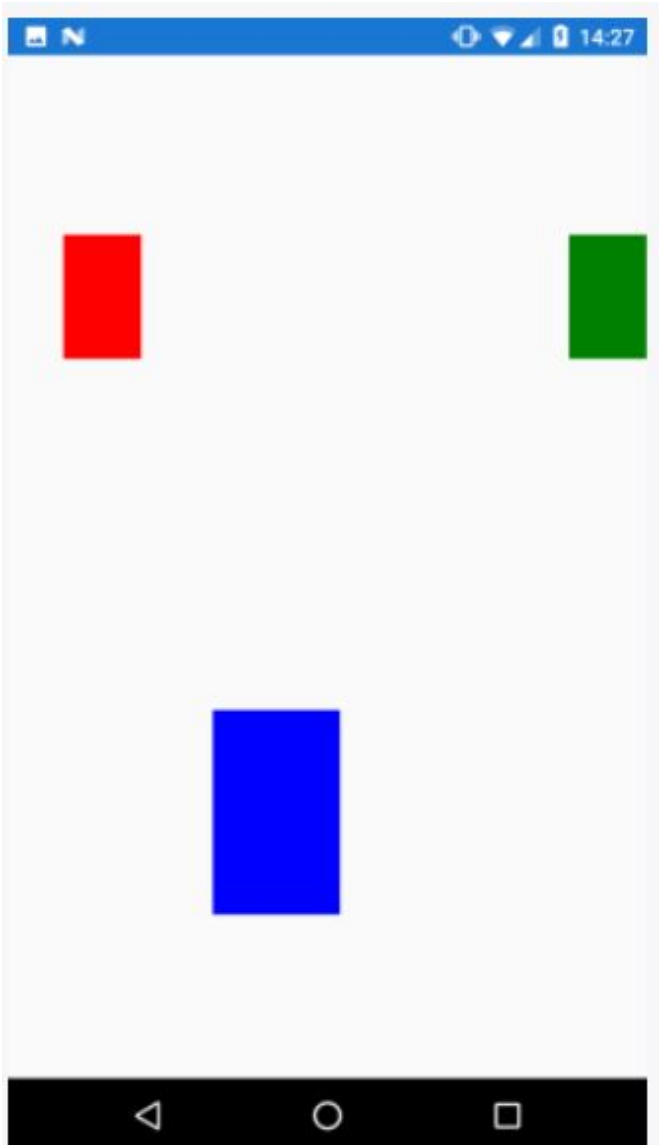
ширина_элемента = пропорциональная_ширина_элемента * ширина_контейнера

Аналогично с координатами. Например, координата X будет вычисляться по следующей формуле:

координатаX_элемента = (ширина_контейнера - ширина_элемента) * пропорциональное_значениеX_элемента

Например, пусть контейнер AbsoluteLayout имеет ширину в 400 единиц, и пусть вложенный элемент имеет ширину в 100 единиц и пропорциональное значение координаты X равное 0.2. Тогда реальная позиция координаты X будет равна (400-100) * 0.2 = 60 пикселей от левого края контейнера AbsoluteLayout.

```
AbsoluteLayout absoluteLayout = new AbsoluteLayout();

    BoxView redBox = new BoxView { BackgroundColor = Color.Red };
    BoxView greenBox = new BoxView { BackgroundColor = Color.Green };
    BoxView blueBox = new BoxView { BackgroundColor = Color.Blue };

    AbsoluteLayout.SetLayoutBounds(redBox, new Rectangle(0.1, 0.2, 50,
80));
    // устанавливаем пропорциональные координаты
    AbsoluteLayout.SetLayoutFlags(redBox,
AbsoluteLayoutFlags.PositionProportional);

    AbsoluteLayout.SetLayoutBounds(greenBox, new Rectangle(1, 0.2, 50,
80));
    AbsoluteLayout.SetLayoutFlags(greenBox,
AbsoluteLayoutFlags.PositionProportional);

    AbsoluteLayout.SetLayoutBounds(blueBox, new Rectangle(0.4, 0.8, 0.2,
0.2));
    // пропорциональные координаты и размеры
    AbsoluteLayout.SetLayoutFlags(blueBox, AbsoluteLayoutFlags.All);

    absoluteLayout.Children.Add(redBox);
    absoluteLayout.Children.Add(greenBox);
    absoluteLayout.Children.Add(blueBox);
```




Аналогично в XAML

```
<AbsoluteLayout>
   <BoxView  Color="Red"
AbsoluteLayout.LayoutBounds=".1, .2, 50, 80"
      AbsoluteLayout.LayoutFlags="PositionProporti
onal"  />
   <BoxView  Color="Green"
AbsoluteLayout.LayoutBounds="1, .2, 50, 80"
      AbsoluteLayout.LayoutFlags="PositionProporti
onal"  />
   <BoxView  Color="Blue"
AbsoluteLayout.LayoutBounds=".4, .8, .2, .2"
      AbsoluteLayout.LayoutFlags="All"
/> </AbsoluteLayout>
```

# RelativeLayout

Контейнер **RelativeLayout** задает относительное позиционирование вложенных элементов относительно сторон контейнера или относительно других элементов.

# RelativeLayout в XAML

Позиционирование и размеры элементов внутри RelativeLayout определяются с помощью ограничений, которые в XAML представляют следующие прикрепляемые свойства:

- RelativeLayout.XConstraint: задает расположение относительно оси X
- RelativeLayout.YConstraint: задает расположение относительно оси Y
- RelativeLayout.HeightConstraint: задает высоту элемента
- RelativeLayout.WidthConstraint: задает ширину элемента

RelativeLayout.HeightConstraint и RelativeLayout.WidthConstraint устанавливаются с помощью числового значения.
А RelativeLayout.XConstraint и RelativeLayout.YConstraint задаются с помощью расширения разметки **ConstraintExpression**, которое включает следующую информацию:

- **Type**: тип ограничения, который указывает, применяется ограничение относительно контейнера или других элементов
- **Property**: свойство, на основании которого устанавливается ограничение
- **Factor**: множитель, на который умножается длина между границами контейнера (0 и 1 - крайние значения)
- **Constant**: смещение относительно контейнера или относительно элемента (в зависимости от значения свойства Type)
- **ElementName**: название элемента, к которому применяется ограничение

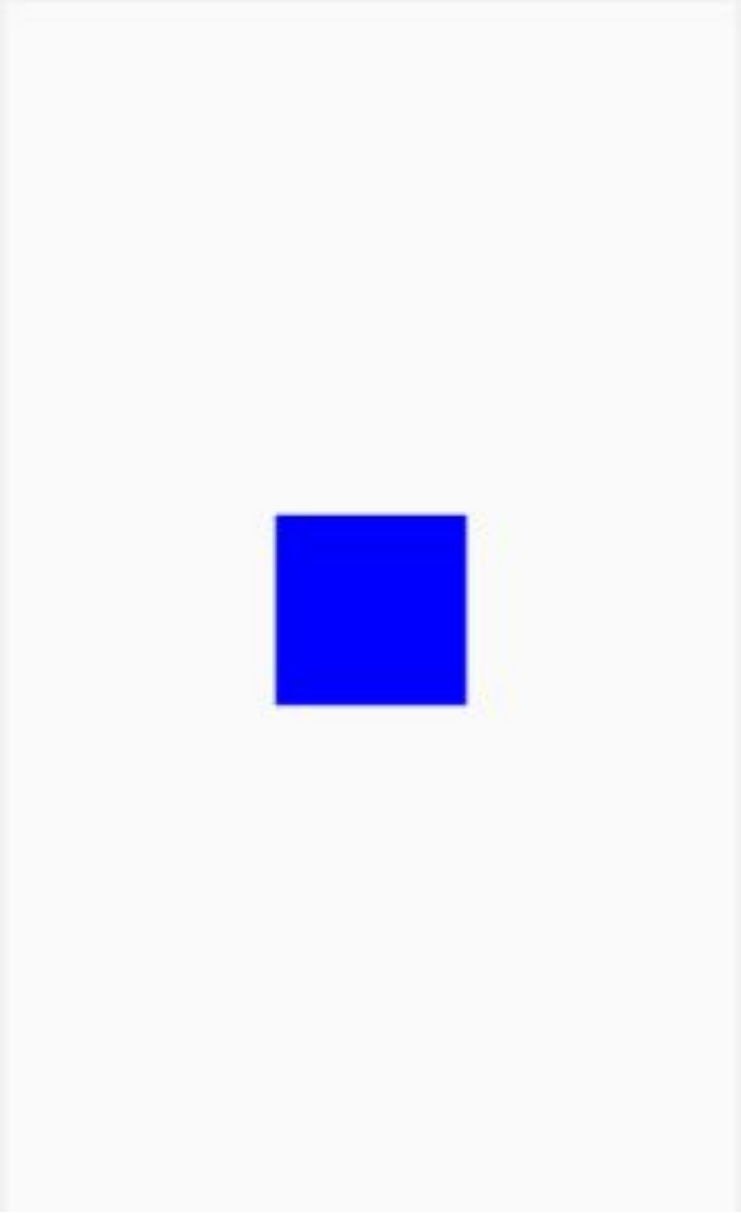
Например, позиционирование элемента BoxView в RelativeLayout в XAML:

```
<RelativeLayout>
    <BoxView WidthRequest="100"
HeightRequest="100" Color="Blue"
        RelativeLayout.XConstraint=
"{ConstraintExpression
            Type=RelativeToParent,
            Property=Width,
            Factor=0.5,
            Constant=-50}"
    />
</RelativeLayout>
```

То есть в итоге мы получаем x = Width * 0.5 - 50. Фактически в данном случае BoxView центрируется по ширине.

Подобным образом добавим второе ограничение RelativeLayout.YConstraint:

```
<RelativeLayout>
        <BoxView WidthRequest="100"
HeightRequest="100" Color="Blue"
            RelativeLayout.XConstraint=
"{ConstraintExpression Type=RelativeToParent,
                Property=Width, Factor=0.5,
Constant=-50}"
            RelativeLayout.YConstraint=
"{ConstraintExpression Type=RelativeToParent,
                Property=Height, Factor=0.5,
Constant=-50}"
        />
    </RelativeLayout>
```



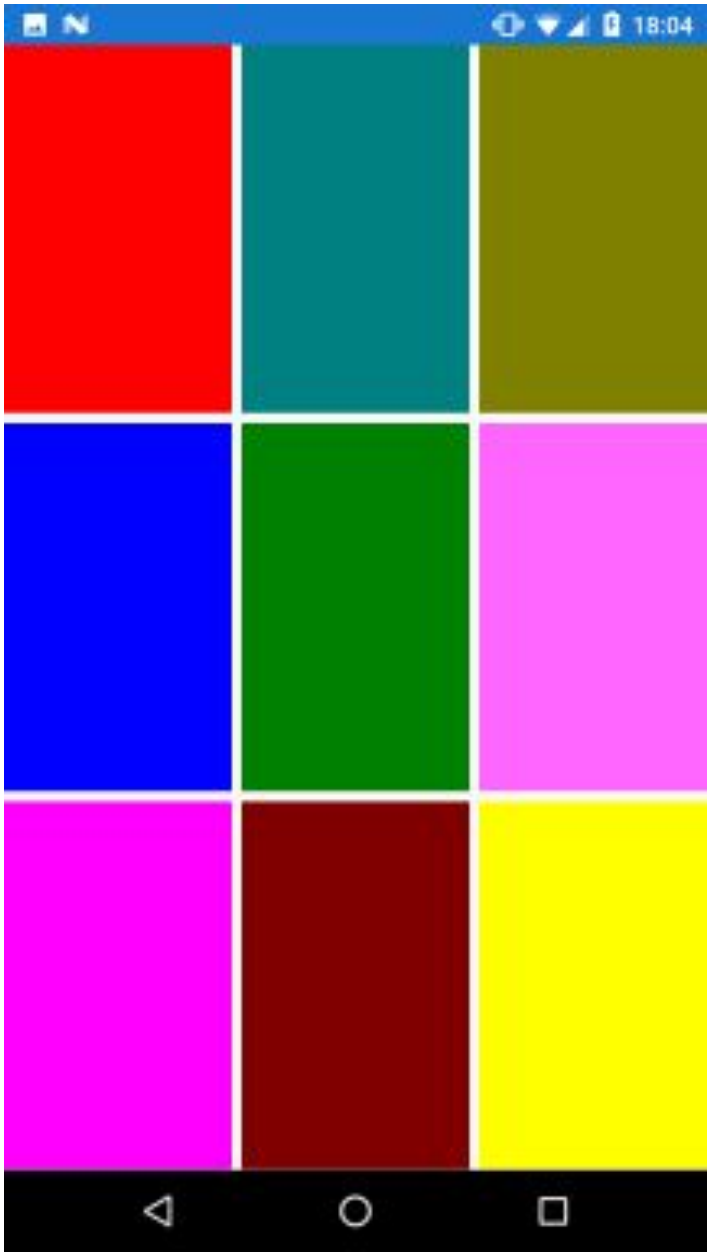
- ???

# Контейнер Grid

Контейнер **Grid** располагает вложенные элементы в виде таблицы:

```
Grid grid = new Grid
      {
         RowDefinitions =
         {
            new RowDefinition { Height = new GridLength(1,
GridUnitType.Star) },
            new RowDefinition { Height = new GridLength(1,
GridUnitType.Star) },
            new RowDefinition { Height = new GridLength(1,
GridUnitType.Star) }
         },
         ColumnDefinitions =
         {
            new ColumnDefinition { Width = new GridLength(1,
GridUnitType.Star) },
            new ColumnDefinition { Width = new GridLength(1,
GridUnitType.Star) },
            new ColumnDefinition { Width = new GridLength(1,
GridUnitType.Star) }
```

```
 grid.Children.Add(new BoxView { Color =
Color.Red}, 0, 0);
 grid.Children.Add(new BoxView { Color =
Color.Blue }, 0, 1);
 grid.Children.Add(new BoxView { Color =
Color.Fuchsia }, 0, 2);

 grid.Children.Add(new BoxView { Color =
Color.Teal}, 1, 0);
 grid.Children.Add(new BoxView { Color =
Color.Green }, 1, 1);
 grid.Children.Add(new BoxView { Color =
Color.Maroon }, 1, 2);

 grid.Children.Add(new BoxView { Color =
Color.Olive }, 2, 0);
 grid.Children.Add(new BoxView { Color =
```




С помощью свойств RowDefinitions и ColumnDefinitions грид создает набор строк и столбцов соответственно.
Для высоты строк и ширины столбцов задается значение new GridLength(1, GridUnitType.Star). Число 1 указывает на то, что данный столбец или строка будет занимать одну долю от всего пространства (так как все три строки или столбца имеют значение 1, то все пространство условно будет равно 1+1+1=3, а одна строка или столбец будет занимать 1/3). А параметр GridUnitType.Star как раз указывает, что размеры будут вычисляться пропорционально.

Чтобы добавить элемент в определенную ячейку, необходимо указывать номер строки и столбца:

```
grid.Children.Add(new BoxView { Color = Color.Blue }, 0, 1);
```

элемент BoxView добавляется в ячейку таблицы, которая находится на пересечении первого столбца и второй строки (исчисление начинается с 0, поэтому 1 будет представлять второй столбец или строку)

# Употребление Grid в xaml:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
```

```
<BoxView Color="Red" Grid.Column="0"
Grid.Row="0" />
<BoxView Color="Blue" Grid.Column="0"
Grid.Row="1" />
<BoxView Color="Fuchsia" Grid.Column="0"
Grid.Row="2" />
<BoxView Color="Teal" Grid.Column="1"
Grid.Row="0" />
<BoxView Color="Green" Grid.Column="1"
Grid.Row="1" />
<BoxView Color="Maroon" Grid.Column="1"
Grid.Row="2" />
 <BoxView Color="Olive" Grid.Column="2"
Grid.Row="0" />
```

Использование звездочки в xaml аналогично параметру GridUnitType.Star. Но кроме пропорциональных размеров мы можем задать автоматические размеры или абсолютные размеры, например:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="2*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="200" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
</Grid>
```

Аналогичное определение разных типов размеров в коде C#:

```
Grid grid = new Grid
{
    RowDefinitions =
    {
        new RowDefinition { Height = new GridLength(2, GridUnitType.Star) },
        new RowDefinition { Height = new GridLength(1, GridUnitType.Star) },
        new RowDefinition { Height = 200 }
    },
    ColumnDefinitions =
    {
        new ColumnDefinition { Width = GridLength.Auto },
        new ColumnDefinition { Width = new GridLength(1, GridUnitType.Star) }
    }
```

# Отступы

Класс Grid определяет два специальных свойства для создания отступов:

- **ColumnSpacing**: определяет пространство между столбцами
- **RowSpacing**: определяет пространство между строками

```
<Grid ColumnSpacing="5">
  <Grid.ColumnDefinitions>
    <ColumnDefinitions Width="*" />
    <ColumnDefinitions Width="*" />
  </Grid.ColumnDefinitions>
</Grid>
```

Или в коде C#:

```
var grid = new Grid { ColumnSpacing = 5 };
grid.ColumnDefnitions.Add(new
ColumnDefinition { Width = new GridLength (1,
GridUnitType.Star)});
grid.ColumnDefnitions.Add(new
ColumnDefinition { Width = new GridLength (1,
GridUnitType.Star)});
```

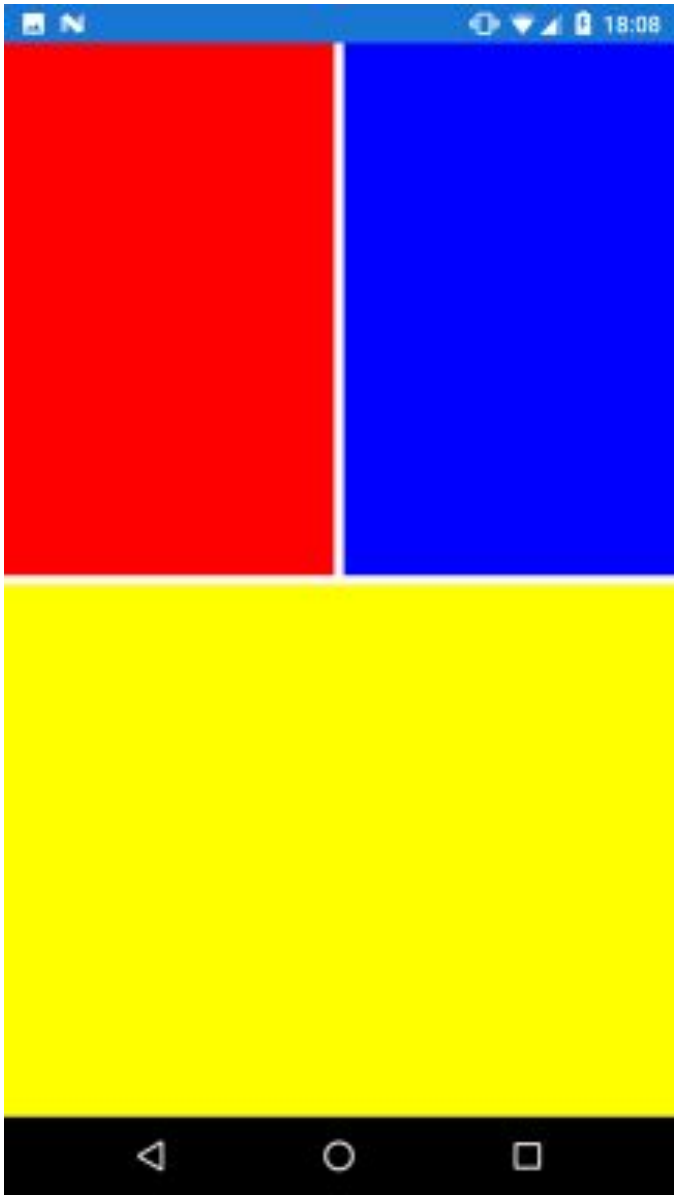
**Объединение ячеек**

С помощью свойства **ColumnSpan** можно объединить несколько столбцов, а с помощью свойства **RowSpan** - объединить ячейки:

```
Grid grid = new Grid
        {
            RowDefinitions =
            {
                new RowDefinition { Height = new GridLength(1, GridUnitType.Star)
},
                new RowDefinition { Height = new GridLength(1, GridUnitType.Star)
}
            },
            ColumnDefinitions =
            {
                new ColumnDefinition { Width = new GridLength(1,
GridUnitType.Star) },
                new ColumnDefinition { Width = new GridLength(1,
GridUnitType.Star) }
            }
        };
        BoxView redBox = new BoxView { Color = Color.Red };
        BoxView blueBox = new BoxView { Color = Color.Blue };
        BoxView yellowBox = new BoxView { Color = Color.Yellow };
        grid.Children.Add(redBox, 0, 0);
        grid.Children.Add(blueBox, 1, 0);
        grid.Children.Add(yellowBox, 0, 1);
        Grid.SetColumnSpan(yellowBox, 2);   // растягиваем на два столбца
```

# Аналог в XAML:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <BoxView Color="Red" Grid.Column="0"
Grid.Row="0" />
  <BoxView Color="Blue" Grid.Column="1"
Grid.Row="0" />
  <BoxView Color="Yellow" Grid.Column="0"
Grid.Row="1" Grid.ColumnSpan="2" />
</Grid>
```


18:08