WPF

Application Development Using WPF

Lesson 1

Introduction to WPF, Containers, Introduction to Controls

Contents

1. Description of WPF 5
2. Reasons for WPF 7
3. WPF Architecture 14
4. Overview of WPF 17
4.1. XAML 17
4.2. User Interface Trees 21
4.3. Events and Commands 23
4.4. Controls 25
4.5. Primitive elements 26
4.6. Panels 28
4.7. Flow Documents 28
5. Overview of XAML 31
5.1. XAML Syntax 31
Case sensitivity 32
Element 32
Attribute 34

Property element....35
Collection property....38
Content property....41
A combination of the Content property and the Collection property....42
Attached Property....43
Type converter....45
Extension of the markup....47
Root Element and Namespaces....49
5.2. The Code-Behind Model....51
The Name and x:Name attributes....52
6. Overview of the base classes of visual tree elements.... 54
6.1. The DispatcherObject class....54
6.2. The DependencyObject class....56
6.3. The Visual Class....56
6.4. The UIElement class....56
6.5. The FrameworkElement class....56
6.6. The Panel class....57
6.7. The Control class....57
6.8. The ContentControl class....57
6.9. The ItemsControl class....57
7. Overview of the WPF application.... 58
7.1. Creating a WPF Project....58
7.2. Structure of the WPF application....59
The App Class....60
The MainWindow class....67
8. XAML Editor Overview.... 75
9. Layout.... 81
9.1. Stages of the Layout....83

10. Hardware-independent units 86
11. Panels 87
11.1. The StackPanel 88
11.2. The WrapPanel 92
11.3. The DockPanel 96
11.4. The Grid Panel 99
11.6. The UniformGrid Panel 115
11.6. The Canvas Panel 117
12. Positioning of elements 120
12.1. Indents 122
12.2. Alignments 125
12.3. Sizes 132
13. Buttons 140
13.1. The Button Control 142
13.2. The RepeatButton Control 144
13.3. The CheckBox Control 145
13.4. The RadioButton Control 148
14. Text Boxes 152
14.1. The TextBox Control 158
14.2. The PasswordBox Control 162
15. Home Assignment 165
15.1. Task 1 165
15.2. Task 2 165

Lesson materials are attached to this PDF file. In order to get access to the materials, open the lesson in Adobe Acrobat Reader.

1. Description of WPF

Windows Presentation Foundation (or **WPF**) is a software platform developed by Microsoft to create applications that have a graphical user interface (GUI), which brought a number of fundamental changes in the operation of these applications compared to its predecessors.

Over the years, GUI applications created for Windows operating systems used USER32 and GDI32 libraries for their work, which provide a variety of window and graphic services, and are located in user32.dll and gdi32.dll, respectively. These libraries are direct descendants of their respective 16-bit versions (USER16 and GDI16) that were created in the mid-1980s. A variety of platforms for the development of user interfaces, created before the release of WPF, (for example, Windows Forms) were just "wrappers" over these libraries. WPF differs from them in that it does not use the GDI32 library to display graphical elements of the user interface and uses a minimum of functionality from the USER32 library, thus representing something more than just a "wrapper" over the old API.

WPF is part of the .NET Framework and includes fragments of both managed and unmanaged code. Despite this, the WPF platform is designed to be used only in the context of the managed .NET Framework and does not contain an open API for interaction from unmanaged code. Therefore, if you want to use WPF, you must use the .NET Framework. Because WPF is the "native" API for the .NET Framework, you can avoid a number of inconveniences associated with using an old, unmanaged API that looks unnatural in this environment.

The initial version of WPF was released as part of the .NET Framework 3.0 in 2006. Because WPF is part of the .NET Framework, their versions are the same; if we are talking about WPF 3.5, we mean the version of WPF released with the .NET Framework 3.5. For this reason, the initial version is called WPF 3.0.

The operating system Windows Vista contains a preinstalled version of the .NET Framework 3.0. You will need to install the .NET Framework of the appropriate version on the user machine in order to run WPF applications on earlier versions of Windows or in order to use a newer version of WPF.

The same rule works in all cases: in order for the WPF application to start on the target machine, it must have the .NET Framework of at least the same version that was selected as the target when creating the application. Below is a list of .NET Framework versions and a list of Windows operating systems where they are preinstalled.

.NET Framework	Windows	Windows Server	Visual Studio
.NET Framework 3.0	Vista	2008 SP2, 2008 R2 SP-1	—
.NET Framework 3.5	7, 8*, 8.1*, 10*	2008 R2 SP1	2008
.NET Framework 4.0	—	—	2010
.NET Framework 4.5	8	2012	2012
.NET Framework 4.5.1	8.1	2012 R2	2013
.NET Framework 4.5.2	—	—	—
.NET Framework 4.6	10	—	2015
.NET Framework 4.6.1	10 v1511	—	2015 Update 1
.NET Framework 4.6.2	10 v1607	2016	—
.NET Framework 4.7	10 v1703	—	2017

*. The .NET Framework 3.5 is not preinstalled on the Windows 8, 8.1, and 10. However, it can be installed through the control panel or in any other way.

Files of the samples used in this lesson are in the archive attached to the PDF of this lesson..

2. Reasons for WPF

Any new technology or platform in the world of software development is created as a means of solving a problem. WPF is a platform for developing GUI applications and, therefore, solves problems related specifically to the development of such applications. The USER32 and GDI32 libraries discussed earlier, and their APIs have proven themselves during their existence and use in a variety of applications, and were widely used and well known among developers. What for to create one more similar platform in that case, and moreover to refuse the checked up development tools?

The fact is that too much has changed since then, as the concepts behind the old user interface were laid in the foundation of the API. Microsoft spends a lot of effort to ensure that each new version of Windows has as much compatibility as possible with its predecessors. An example is the transition from a 16-bit to a 32-bit Windows operating system in the 1990s, and one of goals was to make the 32-bit API as similar as possible to the 16-bit API in order to reduce the problem of porting applications to the new version of the operating system.

In theory, one could write the source code once, which could be successfully compiled for both 16-bit and 32-bit versions of Windows. Those programmers who managed to experience all the 'joys' of such porting might object to how easy it was in practice, but, nevertheless, it was technically possible. This means that the USER32 and GDI32 libraries are rooted in the far-off first 16-bit versions of Windows that were released in the mid-1980s. Thus it follows that the deci-

sions taken during their design exist for more than a quarter of a century.

Hardware has passed a long way since that time, in particular, video cards have completely changed in nature. At the time of the creation of the first Windows OS version, the video card was a fairly simple device, containing a RAM in size not larger than that needed to store information, which must be shown on the screen, plus some electronics to display this information on the screen.

To date, video cards have significantly more power. The amount of their RAM exceeds the size of all available memory of the machine existed in the times of the first Windows. Also, the processing power of video cards is many times greater than 25 years ago. The first video cards worked for the benefit of Windows, as they were created specifically to ensure that the Windows operating systems, OS/2 and X Window System worked faster. However, over time, video card manufacturers realized that focusing on 3D accelerators can make a big profit.

Unfortunately, all the power of new video cards was not available for application developers, unless they were ready to write code using OpenGL or DirectX. Despite the fact that many do so, the use of this API adds too many problems related to the specifics of their usage and is not very well suited to creating familiar user interface controls, since it is mostly focused on working with 3D graphics, while the interfaces of most applications use 2D graphics for their display. The architecture of the USER32 and GDI32 libraries turned out to be not suitable for using the new capabilities of 3D accelerators, and therefore the video cards for the most part are idle while applications that use GDI or GDI+ to output information are running.

One of the goals of WPF is to use these idle resources when working with familiar user interfaces and controls. Since WPF was not "constrained" by the principles followed by the older technologies at the time of its creation, it turned out to use much more capabilities of modern video cards.

Video cards are not the only thing that has changed since the existence of GUI applications. Cathode-Ray Tube monitors (CRT monitors) were used to display the first interface applications. Today, it is almost impossible to meet them, since they were replaced by liquid crystal (LCD) and other "flat" monitors. An interesting feature of the technology used in creating such monitors is the possibility of achieving a much higher resolution. CRT monitors had a maximum pixel density (DPI) of about 100 pixels per inch, while modern monitors already crossed the border of 200 pixels per inch, and this is not the final one. With such a high pixel density, the human eye ceases to distinguish between individual pixels, which makes the image much more pleasant.

The main reason (other than price) why such monitors did not fill the whole world after its appearance is that applications written using predecessor technologies do not work very well with pixels of this size, since they all believe that the pixel is always of the same size in any monitor. If you connect a monitor with a larger pixel density, each pixel will be smaller, which will make the user interface of the application also smaller. This is the main problem of similar monitors. The application interface becomes so small that the application itself becomes almost completely unusable.

In truth, Windows operating systems have been providing the possibility to learn the density of the pixels of the monitor

used by software means for a long time, which in turn makes it possible to write programs that could independently scale their own interface depending on the device used. The problem is that no one bothers. Historically, all monitors used pixels of almost the same size and, therefore, why anyone had to spend time and money developing software that could support display on monitors that did not exist. WPF-applications are called to interrupt this cycle, being initially automatically scalable (DPI-aware). Unlike developers who use older development technologies, the WPF developer does not need to spend any special efforts to make his application automatically scalable. In fact, you will have to take special measures so that the WPF application ceases to automatically adapt to different pixel sizes.

For more than 25 years, not only hardware has changed, but users' requirements for applications have also changed. In the 1980s it was quite acceptable for applications to have a command line interface (CLI). Moreover, many people thought that the whole idea with "windows" was light-minded and focused more on appearance than on content. However, to date, command-line interfaces are more likely to be relics of the past than the main direction of designing application interfaces. They are still popular with certain groups of people, mostly users of UNIX platforms and fans of using command lines, but for ordinary users such interfaces are no longer acceptable. They expect either a web interface or a window interface.

Another reason for the increased requirements for interfaces has become entertainment. In the cinema, the interfaces of non-existent operating systems and applications are often used, which nevertheless look at times more attractive than

ordinary, standard ones that are inherent in earlier versions of Windows. Another example is video games, which also have user interface elements, which, by the way, are fully functional, and also look more beautiful than in normal applications.

Another push towards increasing the requirements for a graphical component of applications is the development of the web industry. The website of almost any company looks in the style inherent in this company, which requires the use of a design consistent with its advertising, booklets and others, whereby the company presents itself. Those websites that do not adhere to this policy, look amateur to date. Conventional Windows applications look rather nondescript in comparison with this. The fact is that it is much more difficult to achieve this effect in terms of graphics and design in applications of this kind than in the web environment. And in many ways the web owes this HTML and related technologies, which even allow people who do not have programming skills to create fairly complex and attractive interfaces. They do not even need to know HTML. There are many tools for "visual" development of sites that create the necessary HTML-markup themselves.

When it comes to the fact that to develop an application for Windows, it is necessary to find a person who understands the principles of the window application and is able to make it look attractive, the chances of finding such a person are rapidly decreasing. WPF solves this problem in the way similar to the solution applicable for web projects — by providing its own markup language XAML. This allows you to create development tools comparable with those used to develop websites and to involve designers more in the application development process.

Windows application developers of the pre-WPF era had all the necessary ingredients for developing attractive applications, but there was also the problem of combining them together. Most of the problems associated with visualization have already been resolved before the advent of WPF. If you want to use all the power of HTML and CSS in an application for Windows, no one will stop you. If you want a scalable and animated graphics, JavaScript provides many libraries with different types of animation, and this technology can also be used in the context of a Windows application. If you do not like the way JavaScript implements controls and the means to interact with them, you can use the Windows API. The standard controls provided by the Windows API cannot boast of a great variety of appearance, but they support all the necessary functionality, and also meet all standards in terms of accessibility, and users know how to interact with them. If you want to use all the power of hardware, then you can use DirectX or OpenGL.

The problem arises at a time when you need to combine the capabilities of these or other technologies. If you want to get the appearance of the controls created through JavaScript, but still maintain the convenience of using the Windows API controls for users, you will not be able to achieve this. Would you like to place a button from the Windows API inside a 3D scene created with DirectX? This also does not work out. Do you want to apply the animation described by CSS to the elements of the 3D scene? Bad luck. The root of the problem lies in the fact that all these technologies are isolated from each other. You can use them side by side within the same application, but you will have to separate the space allocated to

the user interface into parts, and use some specific technology in each of them so that this part does not overlap the other. In this case, you still cannot apply the effects or properties of one technology to the elements of another precisely because of the lack of integration between them.

WPF does not bring many new features from the field of visualization. Most of the individual things that WPF can do were previously available in some other technology. The unique contribution of WPF to the solution of the described problem is in integration. All of the described features are available and are linked together within the same platform. This means that if you want to put a button on the edge of a 3D shape, you can do it. If you want to animate text inside the input field, this is possible. If you want to create a completely new appearance of the control using 2D or 3D graphics, and at the same time get full support for input tools and functionality similar to that found in the Windows API, this is also possible.

3. WPF Architecture

As mentioned before, the main programming interface of WPF, which the developer should interact with, is provided in the form of managed code. In the early stages of WPF development, there was much debate about where the line separating managed and unmanaged components of the system should be drawn. The CLR provides many features that significantly simplify the development process and make it more productive (memory management, exception system, common type system, etc.), but they have their own price.

Fig. 1 illustrates the main components that are part of or used by WPF. Components marked in blue are part of WPF, and green ones are part of the operating system. It also shows which of the WPF components are managed and which are not.

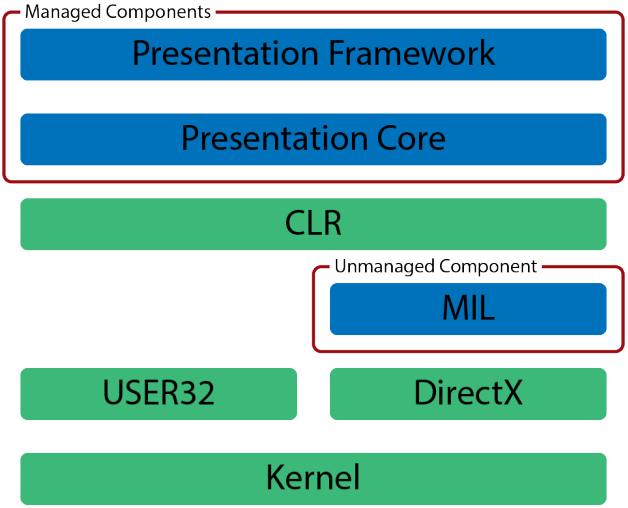


Fig. 1. WPF architecture

In most WPF descriptions, you can find mention of DirectX and the great importance of this technology. It is worthwhile to clarify that DirectX is not part of WPF, but it is used by it to display all the graphics described in the application. For more clarification, it should be noted that the API part called Direct3D is used, which, as it should be clear from the name, is responsible for working with 3D graphics. Despite the fact that in WPF it is possible to work with 3D graphics, focusing only on this means to miss the essence of the whole power of technology. WPF uses hardware that is responsible for working with 3D graphics, even when displaying 2D graphics of the user interface. This is due to the fact that all monitors that exist to date display everything in the form of a flat (two-dimensional) picture. When displaying 3D graphics, all three-dimensional models should be presented in two-dimensional form (projected) at a certain moment so that they can be displayed on a flat screen of the monitor. WPF interacts with this part of the Direct3D graphics pipeline after the flattening of the 3D scene.

Directly above DirectX there is the lowest level component of WPF — Media Integration Layer (or **MIL**), which is located in milcore.dll. This level is called so because it is able to receive different types of information (bitmaps, video, scalable graphics, text, etc.) and output them all to the same DirectX drawing surface. This is the surface where the visual part of integration, which was discussed earlier, takes place. It is worth noting that MIL is created in the form of unmanaged code. This can be unexpected, given the fact that WPF is part of the .NET Framework. The reason for this decision was that MIL interacts directly with the DirectX API, which results in a lot of COM calls. COM is a technology that is applied

in DirectX, and understanding its working principles is not necessary for developing WPF applications, so it will not be considered here. If you use COM technology directly from unmanaged code, then these calls are cheap for the system. However, accessing COM from the .NET Framework requires the use of an interoperability layer, which makes these calls significantly more expensive. Based on this, we can say that the main goal of MIL is to reduce overhead costs by providing a high-level API for accessing the .NET Framework. The API for MIL is undocumented and should not be used directly.

Above the MIL there are two other WPF components: Presentation Core and Presentation Framework, located in PresentationCore.dll and PresentationFramework.dll, respectively. Presentation Core, in fact, is an open interface for MIL that provides an API in the .NET Framework for interacting with display services described in MIL. Presentation Framework, in turn, provides higher-level services, such as: controls, templates, commands, data binding, etc. The developers of WPF applications interact most of the time with the Presentation Framework component.

4. Overview of WPF

WPF has a rather steep learning curve, and one of the biggest problems for those who start to learn this technology is that understanding the work of its individual parts comes with understanding how these parts are combined together and what role they play in the entire software platform. For this reason, before you start a detailed review of each part of WPF, it is worthwhile to consider briefly a few of the most basic ones.

4.1. XAML

One of the first things you will encounter when working with WPF is the **XAML** (*Extensible Application Markup Language*). XAML is not only the most famous feature of WPF, but also one of the most misunderstood. In general, if someone heard about WPF, then he heard about XAML. However, there are two not too widely known facts about XAML. First, WPF does not need XAML, and secondly, XAML does not need WPF. These are two separate technologies. To understand why this is so, let's look at a fragment of XAML-markup:

XAML

```
<Button x:Name="submitButton" FontFamily="Consolas"
        FontSize="20" Foreground="Green">
    Submit
</Button>
```

In this fragment of the markup, attributes describe characteristics of a button, such as font and color, and describes

its contents (text on the button). As a result, the button will be displayed on the screen (Fig. 2).

Fig. 2. Button

From this example, we can conclude that XAML is used to describe the application interface. But how is it that XAML is not an integral part of WPF? In fact, XAML is nothing more than a language for describing .NET Framework objects. In order to become clearer about what is at stake, let's take a look at the code that describes a similar button:

C#

```
var submitButton = new Button
{
    Content = "Submit",
    FontFamily = new FontFamily("Consolas"),
    FontSize = 20.0,
    Foreground = Brushes.Green
};
```

Creating the <Button> element in XAML means that you need to create a new object of the Button type. The x:Name attribute describes the name that you want to use for this object so that it can be accessed in the code. The FontFamily, FontSize, and Foreground attributes refer to the corresponding properties of the object, and the content of the element refers to yet another Content property. This is practically the whole point of XAML. Elements describe objects, and attributes describe their respective properties. Of course, there are a lot of details in this, for example, whence XAML

knows that the value for the FontSize attribute must be numeric, while it is a brush for the Foreground attribute. There are also details describing how XAML understands what to do with the contents of an element both in simple cases, such as the above button, and in more complex ones, such as for a control list that can contain more than one element as a content. There are many similar rules, and they will be discussed in detail later in the sections most suitable for them. The most important thing that you should now understand about XAML is that it describes the creation of objects and the assignment of values to their properties.

The next no less important thing to remember is that everything that can be done using XAML can be done using the code, and it's worth thinking twice before choosing to use one or the other. Quite often, those who are just beginning to learn XAML, try to do everything with it, because, despite its simplicity, XAML is a very powerful tool for managing WPF. The advice in this case is the same as in most similar situations in programming: if you have several tools with which you can solve the task assigned to you, you must solve it with the help of the most suitable one. Despite the fact that XAML is really a great technology, it is not always the best way to solve the problem.

That's what it means when it says that WPF does not need XAML. XAML does not allow you to do anything that cannot be done in the code. And what about the reverse statement? It was also said that XAML does not need WPF. Most likely, the answer to this question is already obvious. If XAML is simply a language for describing objects, then it is not very important for it to know what exactly these objects are. It uses

simple rules based on XML namespaces to determine which type of data to use for each element.

The markup fragment considered earlier does not show this, but the element describing the button requires a default namespace so that this entry makes sense. There is a namespace that contains all the elements that are most commonly used when writing XAML markup for WPF applications. There are also other namespaces, for example, for Silverlight (another technology from Microsoft) that uses the same XAML, but with a different set of elements. And there is also a namespace for the Windows Workflow Foundation technology, which uses XAML to describe workflows, which is hardly similar to the description of user interfaces. And, of course, you can create your own namespaces that will contain your custom types. With a strong desire, you can even teach XAML to create interfaces for Windows Forms applications. However, if you try this, you will quickly find out that not all APIs will be equally convenient to control from XAML. The description of the Windows Forms application interface using XAML does not look as elegant as using XAML in WPF. This is due to the fact that WPF was designed with the idea of XAML. For each WPF property, Microsoft developers have spent a lot of time and effort to consider how it will be described in XAML. At the same time, an API was created and works equally well both from the code and from the markup. Despite the fact that XAML is optional, it lies at the heart of the WPF design.

The goal of using XAML is to involve designers in the user interface development process directly. Therefore, WPF is designed in such a way that you can manage all aspects of the visual component of the application using markup.

4.2. User Interface Trees

Using XAML, the user interface of the application is described, forming a tree-like structure of objects. An example with a button that contains a text node as a child element described earlier was very trivial. Application interfaces in the vast majority of cases look much more complicated, and the tree structure in them is noticeable much better. As an example, let's look at another fragment of the XAML markup:

XAML

```
<Page>
    <StackPanel Orientation="Horizontal">
        <Label>Message:</Label>
        <TextBox Width="100"/>
    </StackPanel>
</Page>
```

The root element is a <Page> containing a single child element — <StackPanel>, whose task is to determine the location of the elements placed in it. In this example, it organizes the label elements (<Label>) and the text input field (<TextBox>) in the horizontal stack (Fig. 3).

Fig. 3. Label and TextBox

The example is still fairly simple, but nevertheless, the tree structure is clearer here: the root node contains one child node that contains several more child nodes. However, if you consider everything that is shown (or can be shown) on the screen, you will get much more than the four nodes described in the markup. The elements of the page and the panel

are not visible here, because they are designed to organize the content and do not have a visual representation. If you look at the text box on the right, you can see that this element consists of several parts. It has a contour, a flashing cursor, if the text were entered, it would also be displayed, and text can also have a selection and scroll bars. A modest text box, as it turns out, consists of many separate parts. It makes sense to consider the text box as one single element, but also no less grounded it can be considered an entity consisting of several parts. Therefore, in WPF, there are two kinds of representation of the user interface structure: a logical tree, as shown in the markup example above, and a visual tree that contains all the elements necessary for displaying the user interface (Fig. 4).

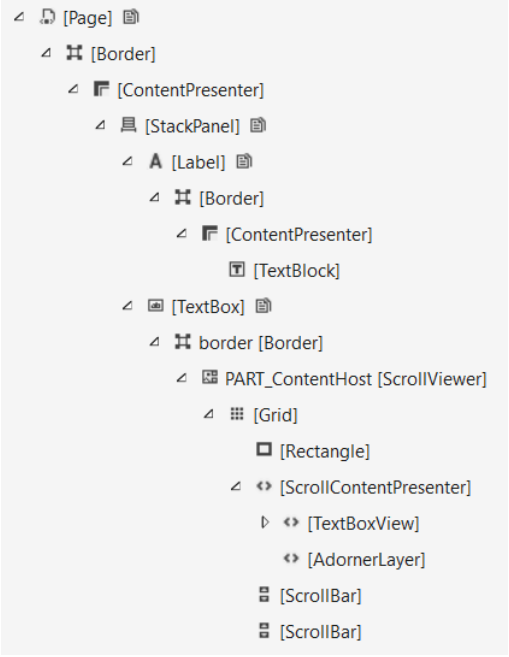


Fig. 4. Visual tree

The visual tree is a superset of the logical tree, and if you look at them, you can see that all the logical elements are also present in the visual tree. Thus, both these trees are just different representations of the same set of elements, which the application interface consists of. In this case, the logical tree does not include many details, which allows you to concentrate on the main structure of the user interface and ignore the details of how it is presented on the screen. It is the logical tree that is used by developers when designing the main framework of the application interface. The visual tree is of interest in situations where you need to focus on the presentation, for example, to change the appearance of the text box.

4.3. Events and Commands

Most applications need to interact with the user, namely, respond to input devices in one way or another. For these purposes, WPF has two constructs: events and commands. Events are closely related to the visual tree. When a user uses keyboard or mouse to input, corresponding events occur in a specific element of the user interface, in that which has the keyboard input focus, or in that which is under the mouse cursor. As you might have noticed earlier, a visual tree can be quite complex and you probably do not want to attach an event handler to each element of the visual tree just to ensure that they get input. WPF provides a mechanism for routing events to solve such problems. This mechanism allows you to handle events by the ancestors of the element for which the event originally occurred. There are two types of routing: tunneling and bubbling.

During tunneling, the event starts with the root element of the visual tree and is routed to the corresponding child

elements, up to the point for which the event was originally intended (Figure 5).

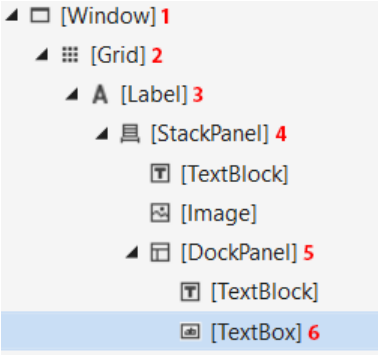


Fig. 5. Event tunneling

Event popup, on the contrary, begins with the element for which it is intended and go up the tree until an element that can handle the event is found (Fig. 6).

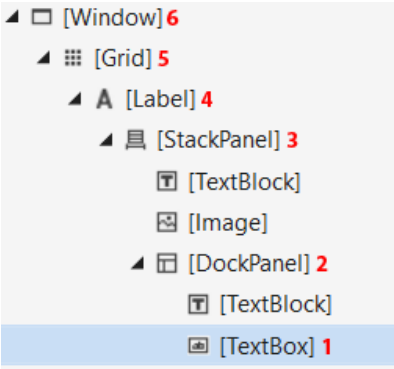


Fig. 6. Event pop-up

Events usually exist in pairs in WPF. There is a preview event, which is tunneled, and there is a main event that pops up. The idea of the preview events is to enable the parent elements to react to the event earlier. For example, a window

can respond to a mouse click before the control on which it was clicked.

Typically, an event is associated with a lower-level interaction, such as clicking a mouse button or pressing a key on a keyboard, but, often, developing an application requires a higher level of abstraction. For example, you can be absolutely indifferent whether the combination of hot keys Ctrl+P was pressed or the print button on the toolbar was clicked or the menu item 'Print...' was selected, because in all these situations you want the application to react equally — the printing process is started. WPF provides a command mechanism that allows you to get a similar level of abstraction. You can write a print command as a separate handler, and associate it with the required hotkey combination, button or menu item. A detailed process of interaction with events and commands will be considered later.

4.4. Controls

WPF supports the concept of controls, but here it means not quite the same as in earlier Microsoft technologies for representing a graphical user interface, such as Windows Forms. In these earlier technologies, the control is a visible object, something that has an appearance that a user can interact with, and an API for developers. The controls in WPF are similar, but not completely.

Let's look at the button as an example. The button provides a click event, which can be attributed to the API, as well as to interaction with the user. It can receive focus, which can be attributed to the characteristic of interactivity. With it, you can associate a command, which again is part of the API. All this, like a number of other features not mentioned here,

is supported in WPF as it was in earlier similar technologies in one form or another. But there is one very important characteristic that the button does not have in WPF, namely, the appearance. It does not know exactly how it should display itself on the screen. And this is the main difference between the controls in WPF from their predecessors. Microsoft even came up with a special word describing this feature of the controls — lookless. In this case, the appearance of the controls is specified using a separate entity — a template. One and the same control element may require different appearance depending on the context of its use (Fig. 7).

Fig. 7. Applying different templates to one button

The button, like other elements, should not know about how it looks, because different themes can require different rendering styles or the application can have its own unique style. It is for this reason that the controls are divided into two parts in WPF: the appearance is determined by the template, and everything else is determined by the control itself.

4.5. Primitive elements

Since the controls do not determine how they look, then how can anything be shown on the screen? It may seem that the appearance of, say, a rectangle element is determined by

its template, which in turn will contain a rectangle element, whose template will be used to display it and so on, which would eventually result in infinite recursion. But there is no such problem, because most of the elements that can be displayed on the screen are not control elements. Only those classes that describe any kind of interactive behavior (buttons, lists, text boxes, scroll bars, etc.) are inherited from the base class that describes the control (System.Windows.Controls.Control). These are all controls. However, classes describing a rectangle, ellipse, image, etc. are not inherited from the base class of the control. They are more primitive and inherited from the more basic class System.Windows.FrameworkElement, which is also the base class for the System.Windows.Controls.Control class. In fact, this class is basic for almost everything that is in the visual tree. Only the heirs of the System.Windows.Controls.Control class contain a template describing their visualization, which was discussed earlier. The heirs of the same class System.Windows.FrameworkElement do not have a template, but they have their own visualization. Another difference is that controls usually have some basic behavior, and primitive elements do not. For example, the text input field System.Windows.Controls.TextBox supports text input and editing, while the primitive element System.Windows.Controls.TextBlock can only display text.

Many believes that a control is everything that can be in the visual tree, including even what does not inherit from the System.Windows.Controls.Control class. In a sense, this is justified. The System.Windows.Controls namespace contains a lot of things that are not inherited from this class. In order to

avoid confusion in the terminology, it is worthwhile to summarize a little. A control is a data type that is inherited from the System.Windows.Controls.Control class and is an element that does not have an appearance, but supports working with templates. A primitive element is a data type that is inherited from the System.Windows.FrameworkElement class and has its own look and does not support working with templates. If it's just about an element from a visual tree, then it can be any of the last ones listed.

4.6. Panels

One of the most useful features of WPF is the set of panels. The panel is an element that you can add to the visual tree, and which can contain several children. The task of any panel is to determine the location of each child element according to its inherent logic. When solving any task of linking several elements, the panels are used, individually or together.

There are several kinds of panels that differ from fairly simple ones, allowing you to position elements at precisely specified coordinates, or align them in a vertical or horizontal stack, to more complex ones that allow you to arrange elements in the form of a grid. In more detail, each of the panels will be discussed later. At this stage, the key point to remember is that if you need to place several elements in any place of the visual tree, then panels will be used almost always.

4.7. Flow Documents

When it comes to an element like the button, it usually matters where exactly it will be located within the interface, in the upper left corner, the bottom right corner, etc. But when it

comes to words in a fragment of the text, the location of each individual word, usually, does not matter. Their order and that they form lines of readable length matters. The authors of the text rarely need to know exactly where the particular word will be, at the top or bottom of the page, and the exact location of the image does not always matter, most importantly, that it appears next to the text to which it relates. The precise positioning of such elements is most often counterproductive, since it is much more useful to be able to adapt the text to the available space, depending on the situation.

WPF provides the means for generating documents that can adjust their content to the space available to them, taking into account the specified formatting rules. Despite the fact that this approach to the formation of the interface is very different from what was considered before, both options can be combined together. In Fig. 8, you can see how the button and the text box are located within the text.

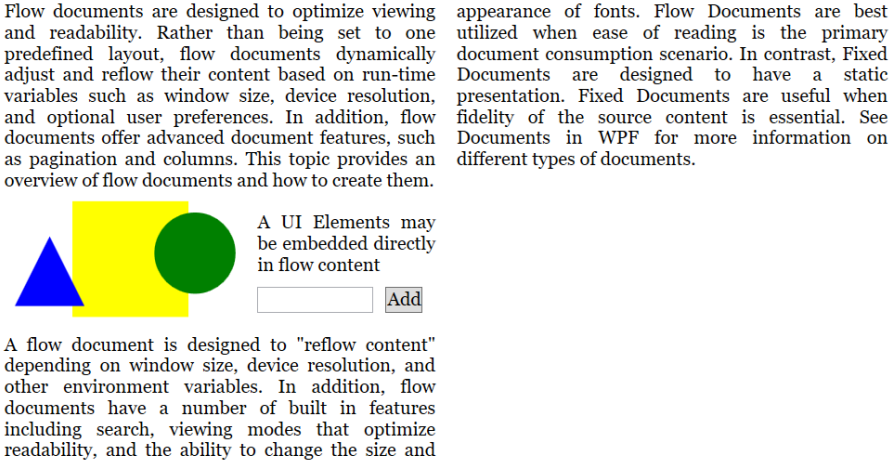


Fig. 8. Flow Document

Thus, it is very real to integrate text into the user interface, in the same way as it is real to integrate user interface elements into the text.

Opportunities for working with text provided to developers of WPF-applications are similar to those that provide HTML. Documents can use different types of text formatting, tables, numbered and unnumbered lists and much more. WPF also provides the ability to split content into pages and columns to better adapt to the size of the available workspace and increase the readability of the text.

5. Overview of XAML

As mentioned earlier, XAML is a separate technology from WPF. All information relating to this technology, which will be discussed below, will concern XAML in the form implemented in WPF.

Using XAML allows you to describe the elements of the user interface using markup by placing a description of their behavior in separate files with program code, the so-called code-behind model, which will be discussed in detail later. Unlike most markup languages, XAML directly describes the creation of objects, thus having a large binding to the underlying type system. Using XAML allows you to achieve a workflow in which different development teams can separately work on creating the user interface and application logic. In this case, completely different development tools can even be used.

Typically, XAML is contained in text files that have the .xaml extension. These files can contain information in the form of any encoding, but, most often, UTF-8 encoding is used.

5.1. XAML Syntax

This section will discuss the syntax of XAML, most of which may seem familiar to those who know XML. XAML defines a number of own concepts that are not in XML, but they fit into its syntax.

Most of the examples in this section are only fragments of full markup and will not work by themselves. This is done so that when explaining a specific feature of the syntax, you can focus only on it, and not be distracted by large fragments of the ac-

companying markup necessary for a full-fledged functioning. In addition, in most cases, it has the same structure. The same goes for the code given here. Next, we will consider examples in which we will show how individual parts are joined together.

Case sensitivity

XAML is case sensitive, like XML. This is due to the fact that the elements described in the markup correspond to software data types that are also case sensitive.

Element

In XAML, elements are intended to describe instances of objects. This means that after the markup is processed by the XAML parser, the objects of the corresponding types will be created. An important feature is that a default data type constructor will be used to create them. The syntax for the element description is as follows:

XAML

```
<TypeName>Content</TypeName>
```

The description of the element consists of the opening (<TypeName>) and the closing (</TypeName>) tags, between which the Content is located. Here TypeName means the name of the element, which, in fact, is the name of the data type. A name can have an additional prefix. Also, the attributes can be optionally located after the name. The meaning and syntax of the names and attributes prefixes will be shown later.

As content, theoretically (according to the rules of the syntax of the language), there can be both ordinary text and other elements (which are called children) in any combination. In practice, the content of each element is strictly described by

the type of data to which it refers. Some can have only one child element, and some — more than one. Others can only contain text, as content. And sometimes, there should not be any content at all. What exactly can or should contain an element will be specified in detail when considering each particular type of data.

As mentioned earlier, it happens that an element in a certain context has no content or, in principle, it cannot possess it. In this situation, nothing is written between the opening and closing tag:

XAML

```
<TypeName></TypeName>
```

In this case, you can use the second form of the element description, which is designed specifically to describe such situations. In this version of the syntax, there is no description of the contents and the closing tag:

XAML

```
<TypeName/>
```

Instead of opening and closing tags there is a so-called self-closing tag (<TypeName/>).

It should be noted that both forms are correct from the point of view of the language syntax and can be used equitably.

Let's look at an example:

XAML

```
<Grid>
    <TextBox/>
</Grid>
```

This fragment of markup demonstrates the creation of two elements: <Grid>, which has content and a closing tag, and <TextBox>, which uses a self-closing form. In fact, here we create two objects that have the types System.Windows.Controls.Grid and System.Windows.Controls.TextBox respectively:

C#

```
var textBox = new TextBox();
var grid = new Grid();
grid.Children.Add(textBox);
```

Attribute

Element attributes are used to set the property values of the corresponding objects. The syntax for describing the attributes is as follows:

XAML

```
<TypeName PropertyName="Value">Content</TypeName>
<TypeName PropertyName="Value"/>
```

If you want to specify values for more than one attribute, then the attributes are described one after the other and separated by a space. The order of the attributes does not matter.

XAML

```
<TypeName PropertyName1="Value1" PropertyName2="Value2"/>
```

As an example, let's consider the following fragment of markup:

XAML

```
<Button Height="30" Width="100">Yes</Button>
```

The following code will correspond to this markup:

C#

```
var button = new Button { Content = "Yes",
Height = 30.0, Width = 100.0 };
```

The attribute syntax can also be used to specify event handlers:

XAML

```
<TypeName EventName="HandlerName"/>
```

Here, EventName is the name of the event, and HandlerName is the name of the method that is the handler for this event. As an example, let's look at the button and click event:

XAML

```
<Button Click="Button_Click">OK</Button>
```

More details about the properties and where exactly the Button_Click method should be located will be described later.

The property can be described no more than once for each object. The following markup will generate an error:

XAML

```
<Button Height="100" Height="50">OK</Button>
```

Property element

No matter how convenient the form of assigning strings as values is, there are many situations where such an option will be inconvenient or even impossible. For example, a property

value is another object that has a number of properties that require values to be assigned to them. Fortunately, XAML provides an alternative entry for assigning values to object properties, the so-called " property element" syntax. In order to describe some property of an object in this form, you must add a child element to the one that requires a property description. The name of this child element must be arranged according to the following principle. As usual, the less-than sign (<), then the name of the parent element, then the dot (.) followed by the name of the property, and the sequence ends with the greater-than sign (>):

XAML

```
<TypeName>
    <TypeName.PropertyName>
        <AnotherTypeName AnotherPropertyName="Value"/>
    </TypeName.PropertyName>
</TypeName>
```

As an example, let's look at how you can describe the value for the property that is responsible for the background of the text box.

XAML

```
<TextBox FontFamily="Consolas" FontSize="20"
         Foreground="Yellow">
    <TextBox.Background>
        <SolidColorBrush Color="Aqua" Opacity="0.5"/>
    </TextBox.Background>

</TextBox>
```

The following code will correspond to this markup:

C#

```
var textBox = new TextBox
{
    Background = new SolidColorBrush { Color =
                                       Colors.Aqua },
    FontFamily = new FontFamily("Consolas"),
    FontSize = 20.0,
    Foreground = Brushes.Yellow
};
```

A similar form of record can be used for ordinary string values. Two forms of assigning a value for the font size are shown below. Both versions do absolutely identical things.

XAML

```
<TextBox FontSize="20"/>
<TextBox>
    <TextBox.FontSize>20</TextBox.FontSize>
</TextBox>
```

If necessary, you can make more than one nested description for different properties of one object. It is also possible to combine them together with a description of the usual attributes.

XAML

```
<TextBox Foreground="Red">
    <TextBox.FontFamily>Consolas</TextBox.FontFamily>
    <TextBox.FontSize>20</TextBox.FontSize>
</TextBox>
```

You cannot describe the same property for one object, applying both forms simultaneously. The following markup will generate an error:

XAML

```
<TextBox Foreground="Red">
    <TextBox.Foreground>Red</TextBox.Foreground>
</TextBox>
```

Collection property

XAML includes a number of optimizations that are designed to create more readable markup. One such optimization is how values are assigned to properties that work with collections of objects in XAML. The syntax for describing collection properties is as follows:

XAML

```
<TypeName>
    <TypeName.CollectionPropertyName>
        <CollectionName>
            <Item1/>
            <Item2/>
            <Item3/>
        </CollectionName>
    </TypeName.CollectionPropertyName>
</TypeName>
```

There is a simplified version of this entry, which does almost the same thing, and which is used much more often. In this case, you cannot explicitly specify the collection element.

XAML

```
<TypeName>
    <TypeName.CollectionPropertyName>
        <Item1/>
```

```
        <Item2/>
        <Item3/>
    </TypeName.CollectionPropertyName>
</TypeName>
```

The following example shows how to assign a collection of values to the System.Windows.Media.GradientBrush.GradientStops property.

XAML

```
<TextBox FontFamily="Consolas" FontSize="20"
         Foreground="Yellow">
    <TextBox.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                   <GradientStop Color=
                            "Red" Offset="0.0"/>
                   <GradientStop Color=
                            "Pink" Offset="1.0"/>
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </TextBox.Background>
</TextBox>
```

Here, the System.Windows.Media.GradientStopCollection collection object is created using the default constructor, after which the System.Windows.Media.GradientStop objects are added to it using the System.Windows.Media.GradientStopCollection.Add method. Next, the collection object is assigned to the System.Windows.Media.GradientBrush.GradientStops property. This markup can also be written in another way:

XAML

```
<TextBox FontFamily="Consolas"
                      FontSize="20" Foreground="Yellow">
    <TextBox.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStop Color="Red" Offset="0.0"/>
                <GradientStop Color="Pink" Offset="1.0"/>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </TextBox.Background>
</TextBox>
```

This case is slightly different from the previous one. Here, for the get-method is called the System.Windows.Media.GradientBrush.GradientStops property, and it returns the current collection object, after which the System.Windows.Media.GradientStop objects are added to it using the System.Windows.Media.GradientStopCollection.Add method.

It may seem that these differences are not essential and you cannot pay attention to them, but it is not so. For example, in the first case, the System.Windows.Media.GradientStop elements are added to the object that was created in advance, which means there may be other elements that will not be deleted. In the second case, the default constructor is used to create a collection object, which may not always be available, which will make this option not usable. Also, this option requires the use of the collection property set method.

The main requirement for the use the syntax of collection properties in part of the code is as follows. The collection property must have a data type that implements the System.Collections.IList interface.

Content property

XAML provides the functionality by which each class can designate any one of its properties as a property for storing content. The content (child or text) of an element of this class will be automatically assigned to this property. This possibility makes it possible to obtain a more vivid structure of the related links of the elements. Most often this property is called Content, but, in principle, it can be called as you like.

As an example, consider the class System.Windows.Controls.Border, in which the content property is the System.Windows.Controls.Decorator.Child property. Below is a fragment of the markup, where two elements are created in different ways, and both variants are completely identical.

XAML

```
<Border>
    <Border.Child>
        <TextBlock Text="Hello"/>
    </Border.Child>
</Border>
<Border>
    <TextBlock Text="Hello"/>
</Border>
```

According to the XAML syntax, the value for the content property should be fully described either before all property elements, or after all, as shown in the example below:

XAML

```
<Button>
    <Button.Background>Red</Button.Background>
    Some long content for button 1
</Button>
```

```
<Button>
    Some long content for button 2
    <Button.Background>Red</Button.Background>
</Button>
```

The following example demonstrates the option of incorrect writing, since a part of the content is described before the property element, and a part — after:

XAML

```
<Button>
    Some long
    <Button.Background>Red</Button.Background>
    content for button 3
</Button>
```

A combination of the Content property and the Collection property

Let's look at an example:

XAML

```
<StackPanel>
    <Button>Button 1</Button>
    <Button>Button 2</Button>
</StackPanel>
```

In this example, the value is assigned to the content property of the <StackPanel> element, which, moreover, is also a collection property. In its full form, this code fragment would look something like this:

XAML

```
<StackPanel>
    <StackPanel.Children>
        <!--<UIElementCollection>-->
            <Button>Button 1</Button>
            <Button>Button 2</Button>
        <!--</UIElementCollection>-->
    </StackPanel.Children>
</StackPanel>
```

The System.Windows.Controls.Panel.Children property is a content property for the System.Windows.Controls.StackPanel class. The type of this property is the System.Windows.Controls.UIElementCollection collection. However, this collection does not have an available default constructor, so this part of the markup is commented out.

Attached Property

The attached properties mechanism allows you to describe a property in one data type, and use it as attributes for an element of a completely different type. This mechanism is most often used in situations where child elements need to specify some data for their parent element, but in addition, the attached properties can be used in other situations.

The syntax of the attached properties resembles the syntax of the property element. In order to describe such an attribute, you need to write the name of the class as the attribute name, in which the attached property is declared, add the dot (.), and write the name of the property:

XAML

```
<TypeName AnotherTypeName.PropertyName="Value"/>
```

As an example, consider the markup option in which a <Grid> panel is created, into which a <TextBox> is placed. In this case, you must specify for the text box in which particular grid cell it should be placed. This is done using the attached properties of System.Windows.Controls.Grid.Column and System.Windows.Controls.Grid.Row.

XAML

```
<Grid>
    <TextBox Grid.Column="0" Grid.Row="0"/>
</Grid>
```

In fact, the attached properties are not properties, they are static methods. Here is an example of the program code that will be generated for the markup described above:

C#

```
var textBox = new TextBox();

var grid = new Grid();
grid.Children.Add(textBox);

Grid.SetColumn(textBox, 0);
Grid.SetRow(textBox, 0);
```

Apparently, there are no properties here, only methods. The rule for translating markup of attached properties into the program code is quite simple. The record TypeName.PropertyName="Value" is translated into TypeName.SetPropertyName(obj, Value), while the obj argument is the object for which the value is assigned to the attached property, i.e. in the example above, this is the text box.

If necessary, you can create your own attached properties and use them in the same way as those described in standard classes.

The attached properties can also be described using the syntax of the property element:

XAML

```
<Grid>
    <TextBox>
        <Grid.Column>0</Grid.Column>
        <Grid.Row>0</Grid.Row>
    </TextBox>
</Grid>
```

It is impossible to describe the same property for one element twice, even using two different forms of writing. The following example will cause an error, because the System.Windows.Controls.Grid.Column property is described twice.

XAML

```
<Grid>
    <TextBox Grid.Column="0">
        <Grid.Column>0</Grid.Column>
        <Grid.Row>0</Grid.Row>
    </TextBox>
</Grid>
```

Type Converter

Let's look at an example that describes the <Button> control, with the property that is responsible for the thickness of the border — System.Windows.Controls.Control.BorderThickness.

XAML

```
<Button Content="Submit">
    <Button.BorderThickness>
        <Thickness Bottom="10" Left="15" Right="10"
                   Top="5"/>
    </Button.BorderThickness>
</Button>
```

As you can see from the example, the System.Windows.Controls.Control.BorderThickness property is assigned to the System.Windows.Thickness object, whose properties prescribe the thickness of each part of the button frame. This entry looks quite logical, and, obviously, we could not immediately assign four values to the System.Windows.Controls.Control.BorderThickness property without using the property element. However, there is an alternative form of recording exactly the same markup:

XAML

```
<Button BorderThickness="15,5,10,10" Content="Submit"/>
```

In this case, the System.Windows.Controls.Control.BorderThickness property, which has the System.Windows.Thickness data type, was assigned a string of the form 15,5,10,10, and it worked because of the fact that for the System.Windows.Thickness there is the so-called type converter, which is a class derived from System.ComponentModel.TypeConverter. This is a fairly simple class that contains a couple of methods for converting a string to the required data type and vice versa. Therefore, if you want XAML markup to be able to assign lines of a specific format that would automatically be converted to objects of the required types, you need to implement your

own type converter. How to do this will be discussed in more detail later.

In WPF, there are many ready-made converters for frequently used data types, which we will also look more closely at as needed.

Extension of the Markup

The ways to set values in object properties, considered earlier, perfectly cope with the task set before them in most situations. Although, they all have one big drawback — they set a fixed value, which is hard-coded directly in the markup. This is not always convenient or possible. There are a number of situations where a value must be received dynamically at runtime, or a specific piece of code must be executed to evaluate it. In both cases, the usual options for setting the values will not work. To solve this problem, the so-called XAML markup extensions were invented. These extensions allow the XAML parser to retrieve values from a third-party class, instead of markup directly.

All markup extensions are implemented as separate classes, which are inherited from the base class System.Windows.Markup.MarkupExtension. This class is very simple — it contains only one ProvideValue method that provides the required value. In other words, when you need to get a value, an instance of the specified markup extension class is created and its method System.Windows.Markup.MarkupExtension.ProvideValue is called, which returns the required value. There are several ready-made markup extensions that will be discussed later, and also, you can create your own one. The syntax for using the markup extension is as follows:

XAML

```
<TypeName PropertyName="{ExtensionName Argument}"/>
```

Here, ExtensionName specifies the name of the class describing the desired markup extension, and Argument passed to the constructor of this class. Here is an example of using one of the standard extensions:

XAML

```
<Button Style="{StaticResource OkButtonStyle}">OK</Button>
```

Here, the markup extension class called System.Windows.StaticResourceExtension is used.

It is customary to name extension classes so that they end with an Extension suffix, but this is not a syntactic restriction. In this case, XAML allows you not to write this suffix.

The syntax for using XAML markup extensions describes several other possibilities, for example, assigning additional properties to an extension class object:

XAML

```
<TypeName PropertyName="{ExtensionName Argument,
                         PropertyName=Value}"/>
```

In the event that the markup extension class contains a default constructor, it can also be used without specifying an argument:

XAML

```
<TypeName PropertyName="{ExtensionName
                         PropertyName=Value}"/>
```

You can also use the markup extension class without setting any properties and specifying the default constructor:

XAML

```
<TypeName PropertyName="{ExtensionName}"/>
```

Root Element and Namespaces

Any XAML file must contain one and only one root element to be a syntactically correct XML and XAML document. In most cases, the root element will be one of several standard ones, such as Window>, <Page>, <UserControl>, <ResourceDictionary> or <Application>, depending on the purpose of the file.

A typical XAML file looks something like this:

XAML

```
<Window xmlns="http://schemas.microsoft.com/
                winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/
                winfx/2006/xaml">
</Window>
```

As the example shows, the root element also contains two attributes: xmlns and xmlns:x. These attributes indicate for the XAML parser which namespaces contain a description of the data types referenced by the elements. The xmlns attribute describes the default namespace. Elements that describe objects from this namespace may not contain a prefix in their name. xmlns:x defines the namespace for some additional XAML constructs.

It is worth noting that it is enough to place the declaration of the attributes responsible for namespaces only in the root element, and they will be visible in all nested ones. It is syntactically possible to make such a declaration for any element of the document, and again, such a declaration will work for all its children. However, this approach quite quickly clogs the file and reduces readability, so it's customary to declare namespaces at the root element level.

If you want to use a data type in the XAML markup that is declared in a namespace other than the default, you will need to declare an alias for that namespace within the XAML file.

As an example, let's imagine the following situation. Someone created their own control and supplies it as a separate assembly Controls.dll. In this assembly, the control is described in the ProgressBars.RoundProgressBar class. The following is a snippet of code that allows you to use this element in your project. In this case, of course, the Controls.dll assembly must be connected to the project.

XAML

```
<Window xmlns="http://schemas.microsoft.com/
                winfx/2006/xaml/presentation"
        xmlns:controls="clr-namespace:ProgressBars;
                assembly=Controls"
        xmlns:x="http://schemas.microsoft.com/
                winfx/2006/xaml">
    <controls:RoundProgressBar/>
</Window>
```

In this case, the alias controls was selected for the ProgressBars namespace. Describing namespaces in XAML, you can rightfully come up with any name for the alias that suits you

best in this situation. In fact, the name for the x namespace, which was declared above, is nothing more than an agreement. You can also call it something different.

The main rule when using elements from other namespaces is that when you specify their name, you will always have to specify a prefix.

In case you want to create an alias for the namespace described in the same assembly containing the XAML markup that will use it, when declaring an alias, you can omit the part describing the name of the assembly in which the namespace is located.

XAML

```
<Window xmlns="http://schemas.microsoft.com/
                winfx/2006/xaml/presentation"
        xmlns:c="clr-namespace:WpfApplication.Controls"
        xmlns:x="http://schemas.microsoft.com/
                winfx/2006/xaml">
    <c:FancyButton/>
</Window>
```

5.2. The Code-Behind Model

The code-behind model describes how XAML markup is combined with the program code. XAML provides the means to associate a file containing the program code with a markup file from the markup side. The way XAML and the program code should integrate together is not solved by XAML. This is done by the software platform that uses it (in this case WPF).

In fact, the following happens: the description of one class is divided into two parts. One part describes the user inter-

face fragment that is implemented by this class in the form of XAML-markup. The other part contains the program code describing the logic of the behavior of this user interface.

From this description it should be obvious that the code-behind model implies the use of two files: a file with markup (.xaml) and a code file (for example, .cs).

When using the code-behind model, it is necessary to observe a number of restrictions and requirements:

- The description of the partial class must be a descendant of the data type that corresponds to the root element of the XAML markup. In fact, when describing a partial class in a code file, you cannot even write that it is inherited from someone. The compiler will automatically finish this. However, it is strongly discouraged to do this, as this introduces ambiguity into the code and is considered a bad practice.
- Methods that are event handlers described in the partial class must be instantiated, and cannot be static.
- The signature of the method that is the event handler must completely match the delegate of the corresponding event.

An example with a detailed description of the code-behind model will be discussed later.

The Name and x:Name Attributes

When describing elements in XAML-markup, you can use two of their attributes with very similar names: Name and x:Name. The x:Name attribute is a concept that belongs to XAML. In fact, this attribute means that you need to create a class field in the file with code-behind model. The Name

attribute is a property of objects owned by WPF that stores the logical name of the visual tree element. Also, the WPF Name attribute is an alias for the x:Name attribute of the XAML attribute, which makes the choice between them in no more than a coding style. However, it is worth remembering that conceptually these attributes are different and belong to different technologies.

6. Overview of the Base Classes of Visual Tree Elements

Various elements that can be present in the visual tree will be considered in the following sections. Elements that have similar functionality are inherited from the same base classes in which this functionality is implemented. In this case, part of the class hierarchy (Fig. 9) is practically identical for all these classes. Therefore, in this section, we will briefly discuss the most basic classes and the relationships between them. As necessary, they will be considered in more detail.

Despite the fact that not all classes in this diagram are abstract, it is not worth creating their instances directly. In most situations, you will need to create objects derived from their classes.

6.1. The DispatcherObject Class

Virtually every WPF-element has a so-called binding to the thread. In other words, access to such an element should only be done from the thread that created it. If you try to access the properties of an element for reading or writing from a thread in which it was not created, then an exception of type System.InvalidOperationException will be thrown, and the corresponding error will be described in the message.

In order to implement such functionality, each object that needs to be bound to a thread is ultimately inherited from the System.Windows.Threading.DispatcherObject class.

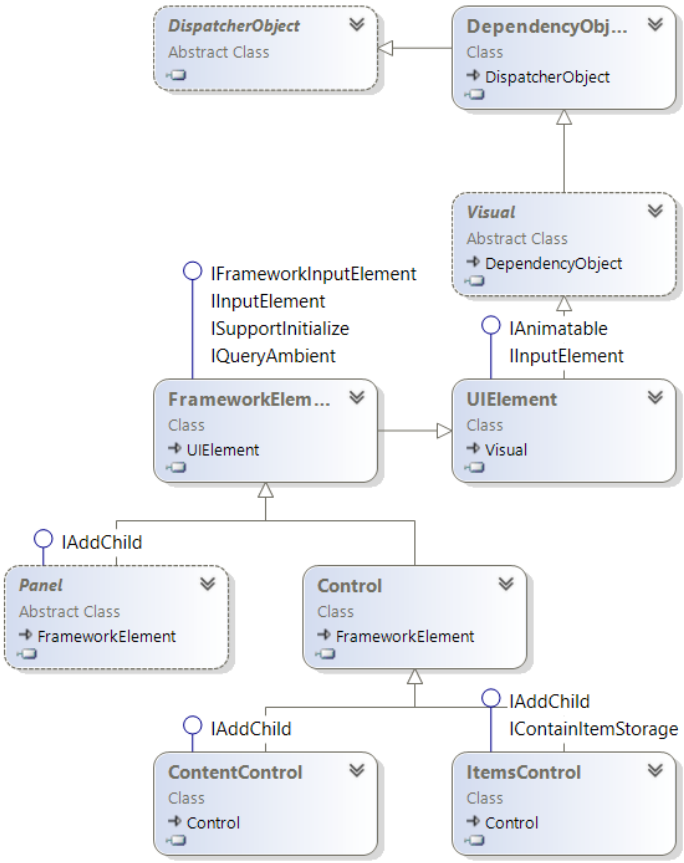


Fig. 9. Basic classes of visual tree elements

This class provides a property that returns an object by which the necessary operations are performed on the element with which it is associated. This dispatch object has a task queue, and is responsible for ensuring that they are executed in the element thread.

If the project does not use multithreading when working with interface elements, then the situation described above is not terrible.

6.2. The DependencyObject Class

By default, any property created in the .NET Framework has very basic functionality. The System.Windows.DependencyObject class is used to implement the WPF property system, which is slightly different from the usual, providing an extended set of features, such as notification for value changes, validation, style binding, default value, etc.

6.3. The Visual Class

The System.Windows.Media.Visual class provides a set of methods and properties that are required to display elements on the screen. Also, the class provides functionality for interacting with the visual tree and checking for a visual hit of the cursor on the element (hit test).

6.4. The UIElement Class

The System.Windows.UIElement class contains a large number of methods, properties, and events, which are mainly responsible for interacting with the user. Among the implemented functionality, it can be noted: the acquisition of data through the keyboard, mouse and other input devices, input focus control, and control of the overall status of the element (visibility, inclusion, etc.).

6.5. The FrameworkElement Class

The System.Windows.FrameworkElement class adds even more functionality to the WPF elements, which is required to implement the element layout system, the logical tree, styles, and data binding. Also here is a set of events

describing the life cycle of the elements. You can use them to determine, for example, when an element is initialized, loaded, or unloaded.

6.6. The Panel Class

The System.Windows.Controls.Panel class is the base class for panels that are responsible for arranging and locating the elements placed in them. Panels are the main mechanism for composing user interfaces in WPF.

6.7. The Control Class

The System.Windows.Controls.Control class is the base class for all controls, the main distinguishing feature of which is the presence of a template responsible for their visualization.

6.8. The ContentControl Class

The System.Windows.Controls.ContentControl class is the base class for all controls that contain only one child element.

6.9. The ItemsControl Class

The System.Windows.Controls.ItemsControl class is the base class for all controls that can contain multiple children.

7. Overview of the WPF Application

This section will detail the process of creating a WPF application, its structure, and the use of the code-behind model.

7.1. Creating a WPF Project

The process of creating a WPF application in the Visual Studio development environment is almost the same as the process of creating any other .NET Framework project. First of all, select the menu item File → New → Project..., or press Ctrl+Shift+N (Fig. 10).

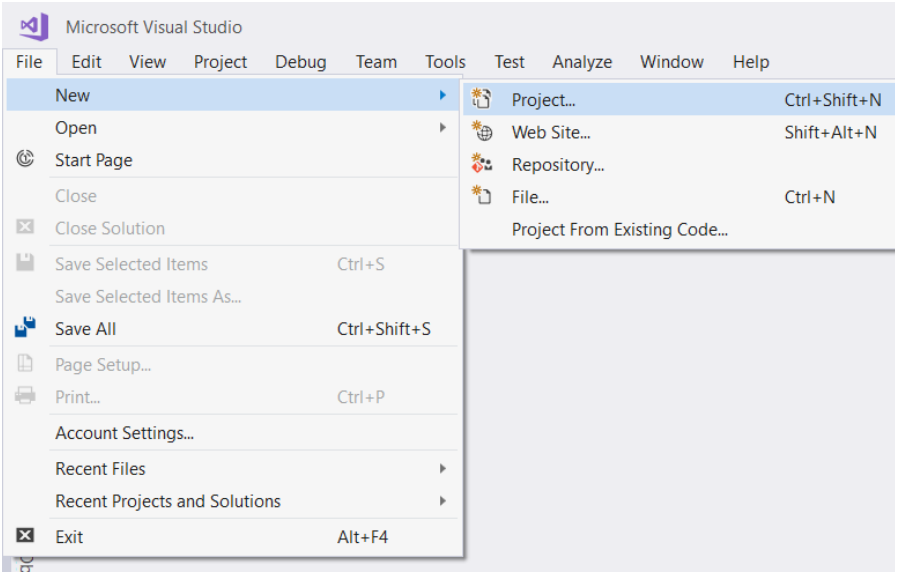


Fig. 10. Select the main menu item to create a new project

In the window that opens, on the left, select Templates → Visual C# → Windows Classic Desktop, and then select the project type WPF App (.NET Framework). At the bottom of the window, as in other types of projects, there are Name and Location fields, which you can change if you want to change the project name or its location accordingly (Fig. 11).

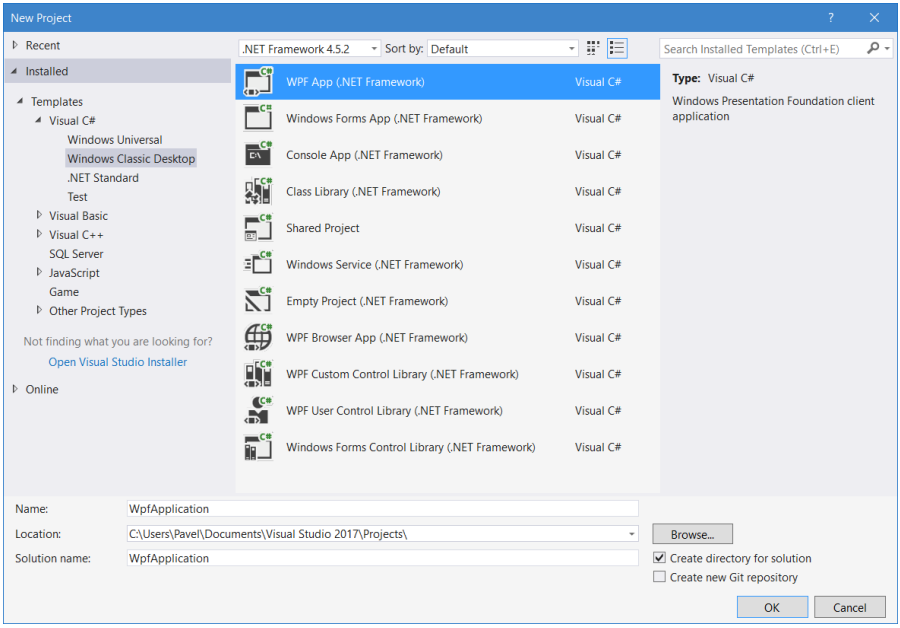


Fig. 11. Select the type of project

7.2. Structure of the WPF Application

Each new project has the same content, the so-called "stub". You can immediately compile the project and see the finished application, consisting only of an empty window. In different versions of Visual Studio, the automatically generated start code and the set of default assemblies that are installed by default

can be different. In addition, some of the generated code is not always needed, so here we will consider the minimum code required at this stage. Therefore, if you create a project and notice a piece of code in it that is not covered in this section, you are more likely to delete it. The initial file structure of the new project is shown in Fig. 12.

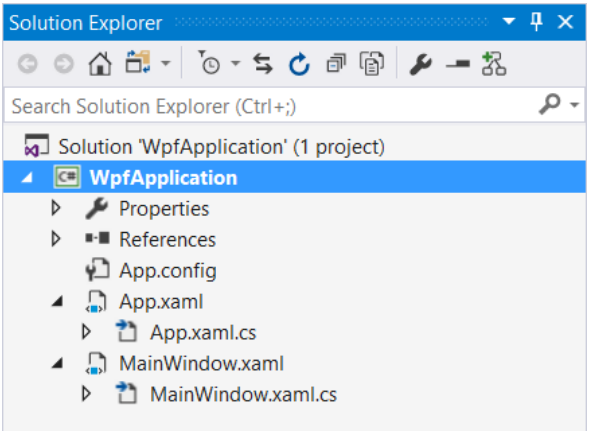


Fig. 12. The initial structure of the files of the new project

Here you can see that the files are grouped in pairs. In fact, there are 4 files in the project folder that will be discussed in this section: App.xaml, App.xaml.cs, MainWindow.xaml, MainWindow.xaml.cs. From the titles it is seen that the pairs of files describe two separate entities. This file structure is necessary for implementing the code-behind model.

The App Class

The WPF application does not have a familiar entry point, which is usually the Main method. Technically, the Main method, of course, exists, since this is part of the .NET Framework infrastructure, but the developer should not explicitly

describe it, because it will be generated automatically. Instead, the application developer must create a class that will inherit from the System.Windows.Application class. In our case, this is the App class, the description of which is in two files: App.xaml and App.xaml.cs.

The compiler will generate the Main method of about the following form:

C#

```
public static void Main()
{
    var application = new App();
    application.InitializeComponent();
    application.Run();
}
```

The App.InitializeComponent method is also generated automatically and is required to configure the main application window, and the System.Windows.Application.Run method starts the application.

The App.xaml file might look like this:

XAML

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

The most interesting part of this markup is the x:Class="WpfApplication.App" attribute. This is a key point in the implementation of the code-behind model. This is how you spec-

ify where the program code pertaining to this markup file is contained. The value of the x:Class attribute must be the name of the corresponding class, specifying the full namespace.

Another not unimportant moment here is the StartupUri attribute, which contains the name of the markup file containing the description of the main application window. The path to the file here is written relative, which is calculated relative to the project's root directory. If the directory structure were the same as shown in Fig. 13, then the value of the StartupUri attribute would be Windows/MainWindow.xaml.

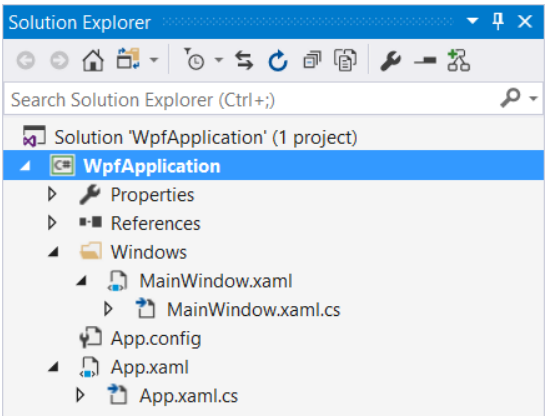


Fig. 13. Alternative structure of project files

The file App.xaml.cs contains the following code:

C#

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

As you can see, this file describes only the empty App class. The most important here is the indication that it is inherited from the System.Windows.Application class, and is partial. This class, of course, can contain a certain code, but it is not optionally.

It is also very important that the class has a default constructor (which can be declared using any access specifier). By default, all classes that use the code-behind model must be marked with a public access specifier. For obvious reasons, this may not always be convenient. In certain situations, it may be required that the class has an internal access specifier. For this to work, you will need to add a XAML file to the root element that contains the x:Class attribute, and another attribute — x:ClassModifier. The value of this attribute differs depending on which programming language is used in conjunction with XAML. For C#, the value of this attribute must be internal, i.e. the markup will look like this:

XAML

```
<Application x:Class="WpfApplication.App"
             x:ClassModifier="internal"
             xmlns=  "http://schemas.microsoft.com/
                      winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/
                      winfx/2006/xaml"
             StartupUri="MainWindow.xaml"/>
```

It is worth mentioning the need for consistency of these access specifiers. If you specified the x:ClassModifier attribute in the markup with the internal value, then the class in the code must be marked with the same access qualifier.

The same applies to the consistency of partial descriptions of the public class.

The Application class

Static properties of the System.Windows.Application class:

- **Current** (System.Windows.Application). Gets the object of the current application.
- Properties of the System.Windows.Application class:
- **MainWindow** (System.Windows.Window). Gets or sets the main application window.
- **ShutdownMode** (System.Windows.ShutdownMode). Gets or sets the application's closing mode. The default value is System.Windows.ShutdownMode.OnLastWindowClose.
- **StartupUri** (System.Uri). Gets or sets a URI that describes what exactly should be shown when the application starts.
- **Windows** (System.Windows.WindowCollection). Gets the collection of created windows in the application.
- Events of the System.Windows.Application class:
- **Activated** (System.EventHandler). It works when the application becomes active.
- **Deactivated** (System.EventHandler). It works when the application stops being active.
- **Exit** (System.Windows.ExitEventHandler). It works just before the application is closed. This event cannot be cancelled.
- **Startup** (System.Windows.StartupEventHandler). It works when the System.Windows.Application.Run method is called. In fact, this is the earliest place in the life cycle of

a WPF application that can be accessed. Typically, here you configure the resources and properties of the entire application level.

Methods of the System.Windows.Application:

- **Run**(). Launches the application.
- **Run**(window). Launches the application and displays the specified window.
- **Shutdown**(). Shutdowns the application and returns the exit code — 0 to the operating system.
- **Shutdown**(exitCode). Shutdowns the application and returns the specified exit code to the operating system.

In order to specify the application shutdown mode, you must use the System.Windows.ShutdownMode enumeration, which contains the following options:

- OnExplicitShutdown. The application will be closed only if the System.Windows.Application.Shutdown method is explicitly called.
- OnLastWindowClose. The application will be closed if the last application window is closed or the System.Windows.Application.Shutdown method is explicitly called.
- OnMainWindowClose. The application will be closed if the main application window is closed or the System.Windows.Application.Shutdown method is explicitly called.

Events in the standard WPF classes are implemented according to the Microsoft recommendations. This means that in addition to the event, there is also a method in the class that you can override. The following is an example of how you can subscribe to the System.Windows.Application.Startup event.

C#

```
public partial class App : Application
{
    public App()
    {
        Startup += App_Startup;
    }

    private void App_Startup(object sender,
            StartupEventArgs e)
    {
        // Startup code...
    }
}
```

Another option is to override the method:

C#

```
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);
        // Startup code...
    }
}
```

The examples shown above are completely analogous in their functionality. However, you should pay attention to the string base.OnStartup(e). If it is removed, the logic of the work will change. Calling the base version of the System.Windows.Application.OnStartup method is required in order to notify all the subscribers of the System.Windows.Application.Startup event. In other words, if you combine both variants together

and remove the specified string, the App.App_Startup method will not be called:

C#

```
public partial class App : Application
{
    public App()
    {
        Startup += App_Startup;
    }

    private void App_Startup(object sender,
            StartupEventArgs e)
    {
        // Startup code that would never be executed...
    }

    protected override void OnStartup(StartupEventArgs e)
    {
        // Startup code...
    }
}
```

Everything that has been described regarding the System.Windows.Application.Startup event applies to almost all events in the WPF classes that will be discussed later.

The MainWindow Class

Another group of files (MainWindow.xaml and MainWindow.xaml.cs) contains a description of the class of the main application window. This group of files also uses the code-behind model, and all the rules relating to it that apply to the App class are applicable to it.

The MainWindow.xaml file is arranged like this:

XAML

```
<Window x:Class="WpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/
        winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/
        winfx/2006/xaml"
        Height="350"
        Title="MainWindow"
        Width="525">
    <Grid></Grid>
</Window>
```

The <Window> element describes the window and is probably the most commonly used root element in WPF. The x:Class attribute still contains the name of the partial class in which the code of the code-behind model is located. The Width and Height attributes specify the size of the window, and the Title attribute is the header.

The <Window> element can contain only one child element, so the default option here is the <Grid> panel, which is the most "advanced" of all. You can change it to any other control or another panel.

The MainWindow.xaml.cs file contains the following code:

C#

```
public partial class MainWindow : Window
{
    public Window()
    {
        InitializeComponent();
    }
}
```

Here the most important is to call the MainWindow.InitializeComponent method, which is generated automatically. It is in this method that the corresponding XAML-file is analyzed and objects are created according to its description.

The Window class

Fig. 14 shows the class System.Windows.Window and its base class.

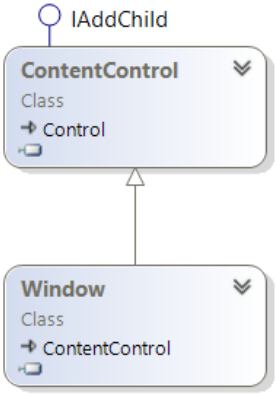


Fig. 14. A diagram of the classes that describe the window

Properties of System.Windows.Window class:

- **AllowsTransparency** (System.Boolean). Gets or sets a value that indicates whether the client area supports transparency. true if the client area supports transparency; otherwise, false. The default value is false.
- **DialogResult** (System.Nullable<System.Boolean>). Gets or sets the value to be returned from the System.Windows.Window.ShowDialog method. The default value is false.
- **Icon** (System.Windows.Media.ImageSource). Gets or sets the window icon.

- **IsActive** (System.Boolean). Gets a value that indicates whether the window is active. true if the window is active; otherwise, false.
- **Left** (System.Double). Gets or sets the position of the left border of the window relative to the left border of the desktop.
- **OwnedWindows** (System.Windows.WindowCollection). Gets a collection of windows for which the current window is the owner.
- **Owner** (System.Windows.Window). Gets or sets the window that owns the current window.
- **ResizeMode** (System.Windows.ResizeMode). Gets or sets the window resizing mode. The default value is System.Windows.ResizeMode.CanResize.
- **RestoreBounds** (System.Windows.Rect). Gets the size of the window before it was minimized or expanded to full screen.
- **ShowActivated** (System.Boolean). Gets or sets a value that indicates whether the window should be activated on the first display. true if the window should be activated on the first display; otherwise, false. The default value is true.
- **ShowInTaskbar** (System.Boolean). Gets or sets a value that indicates whether the window should be displayed on the taskbar. true if the window should be displayed on the taskbar; otherwise, false. The default value is true.
- **SizeToContent** (System.Windows.SizeToContent). Gets or sets the mode for fitting the window sizes to the content. The default value is System.Windows.SizeToContent.Manual.
- **Title** (System.String). Gets or sets the window title.

- **Top** (System.Double). Gets or sets the position of the top border of the window relative to the top border of the desktop.
- **Topmost** (System.Boolean). Gets or sets a value that indicates whether the window should be on top of other windows. true if the window should be on top of other windows; otherwise, false. The default value is false.
- **WindowStartupLocation** (System.Windows.WindowStartupLocation). Gets or sets the start position of the window on the first display. The default value is System.Windows.WindowStartupLocation.Manual.
- **WindowState** (System.Windows.WindowState). Gets or sets the window state (minimized, maximized, restored). The default value is System.Windows.WindowState.Normal.
- **WindowStyle** (System.Windows.WindowStyle). Gets or sets the window frame style. The default value is System.Windows.WindowStyle.SingleBorderWindow.
- Events of the System.Windows.Window class:
- **Activated** (System.EventHandler). It works when the window becomes active (moved to the foreground).
- **Closed** (System.EventHandler). It works when the window is closed.
- **Closing** (System.ComponentModel.CancelEventHandler). It is triggered immediately after the call of the System.Windows.Window.Close method, and before the System.Windows.Window.Closed event. Can be processed to cancel the process of closing the window.
- **ContentRendered** (System.EventHandler). It works when the contents of the window are displayed.

- **Deactivated** (System.EventHandler). It works when the window stops being active (moved to the background).
- **LocationChanged** (System.EventHandler). It works when the position of the window changes.
- **SourceInitialized** (System.EventHandler). Triggers when the window handle is initialized.
- **StateChanged** (System.EventHandler). It works when the value of the System.Windows.Window.WindowState property changes.

Methods of the System.Windows.Window class:

- **Activate**(). Moves the window to the front. Returns true if the operation succeeded; otherwise, false.
- **Close**(). Closes the window.
- **Hide**(). Hides the window.
- **Show**(). Shows the window.
- **ShowDialog**(). Shows the window and waits for it to close. Returns the value that describes the result of the interaction with the window: the action was taken (true) or canceled (false).

In order to specify the window resizing mode, you must use the System.Windows.ResizeMode enumeration, which contains the following options:

- NoResize. The dimensions of the window cannot be changed. The "minimize" and "expand" buttons do not appear in the window title.
- CanMinimize. The window can be minimized and restored. The "minimize" and "expand" buttons are displayed, but only the "minimize" button is active.

- CanResize. The dimensions of the window can be changed. The "minimize" and "expand" buttons are displayed and active.
- CanResizeWithGrip. The dimensions of the window can be changed. The "minimize" and "expand" buttons are displayed and active. In the lower right corner of the window, a size grip element is displayed for resizing the window.

In order to specify the mode for fitting a window to the content, you must use the System.Windows.SizeToContent enum, which contains the following options:

- Height. The height of the window will automatically adjust to the height of the content, and the width will not.
- Manual. The window's dimensions will not automatically adjust to the content. Instead, the size of the window will be controlled by a set of properties responsible for explicitly indicating the dimensions and the allowed ranges of resizing.
- Width. The width of the window will automatically adjust to the width of the content, and the height will not.
- WidthAndHeight. The window sizes will automatically adjust to the content sizes.

In order to specify the initial position of the window on the first display, you must use the System.Windows.WindowStartupLocation enumeration, which contains the following options:

- CenterOwner. The window is positioned in the center of its owner, which is described by the System.Windows.Window.Owner property.

- CenterScreen. The window is positioned in the center of the screen.
- Manual. The initial position of the window is determined by the values set explicitly or the default position is used.

In order to specify the window state (minimized, expanded, restored), you must use the System.Windows.WindowState enumeration, which contains the following options:

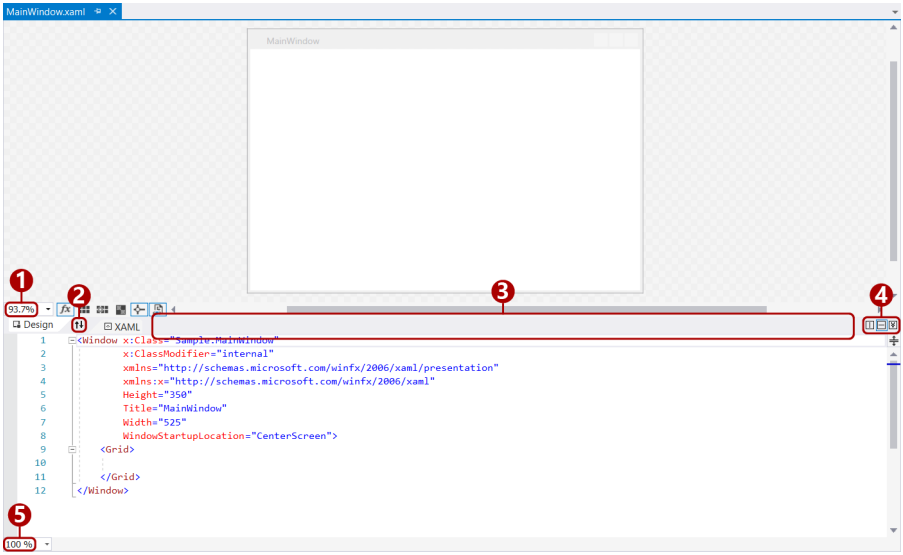
- Maximized. The window is maximized.
- Minimized. The window is minimized.
- Normal. The window is restored.

In order to specify the window frame style, you must use the System.Windows.WindowStyle enumeration, which contains the following options:

- None. The window title and frame are not displayed. Only the client area of the window is displayed.
- SingleBorderWindow. A window with a conventional border.
- ThreeDBorderWindow. Window with a 3D border.
- ToolWindow. Window with a thin border without the buttons "minimize" and "expand".

8. XAML Editor Overview

When you open the XAML file in Visual Studio, you will be taken to the markup editor window, shown in Fig. 15. The working area of this window is divided into two areas. The top shows the result of the current markup in real time, which, in turn, is described in the lower area. If necessary, you can change the size of each of these areas, swap them, change the scale or change the separation from horizontal to vertical. If you interact with any of these areas, making changes (editing the markup or moving controls), these



1. Editable drop-down list that allows you to change the scale of the top work area.
2. A button that allows you to swap work areas.
3. Slider, which allows you to change the size of each working area due to the other.
4. Buttons for switching between horizontal and vertical division of areas.
5. Editable drop-down list allowing to change the scale of the lower working area.

Fig. 15. The XAML editor window

changes will immediately be synchronized and displayed in the other.

You, as a developer, have several ways to edit XAML-markup using this editor.

The most obvious way is to edit the markup manually. To do this, you need to interact with the bottom of the editor. There are no special features worth talking about. A typical text editor that any developer who has previously interacted with Visual Studio knows.

The following method involves "visual" editing of markup. For this type of editing you will need to interact with additional windows of Visual Studio. The first one is the Toolbox. In order to open this window, you need to select the menu item View → Toolbox, or press the combination of hot keys Ctrl+Alt+X (Fig. 16).

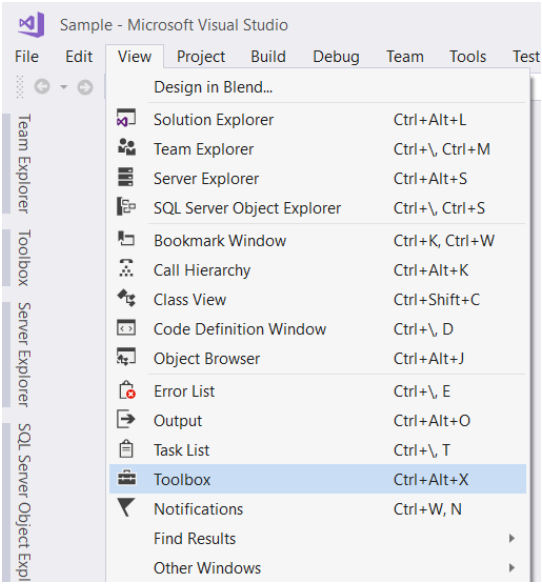


Fig. 16. Select the main menu item to open the Toolbox window

The Toolbox window (Fig. 17) contains a list of elements (controls, primitive elements, panels) that you can use to create markup.

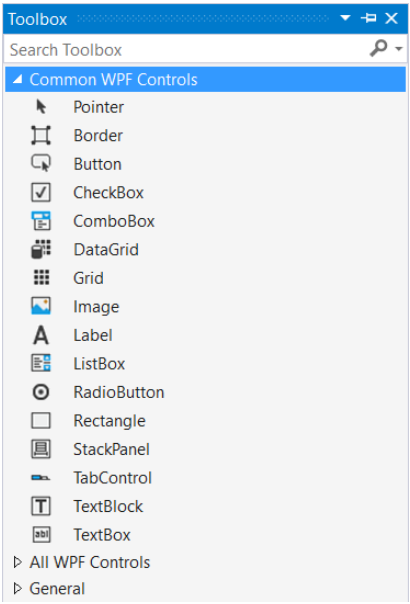


Fig. 17. The Toolbox window

Some windows in Visual Studio can work differently, depending on what window or interface element is active at the moment. Toolbox works on this principle. In order for this window to be filled with elements that can be used to create the interface, it is necessary that the XAML-editing window is active. If you have any other window active, for example, the C# code editor, the Toolbox will have a different appearance.

If you look at this window, you can see two drop-down items: Common WPF Controls and All WPF Controls. They contain the most frequently used elements for WPF development and all available elements respectively.

In order to use one of them, you must hover the mouse cursor over the desired element, press and hold the left mouse button and drag it to the area with the window (the top part of the XAML editor) and release the button. You can also move existing items in the same way. In this case, a fragment of the markup, which will display the executed manipulations will automatically add to XAML the file.

Another option for placing an element within the visual part of the editor is double-clicking the left mouse button on

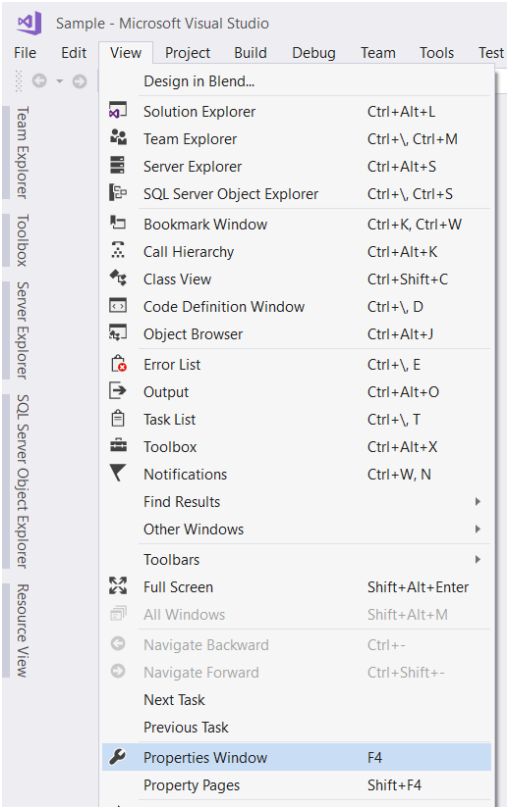
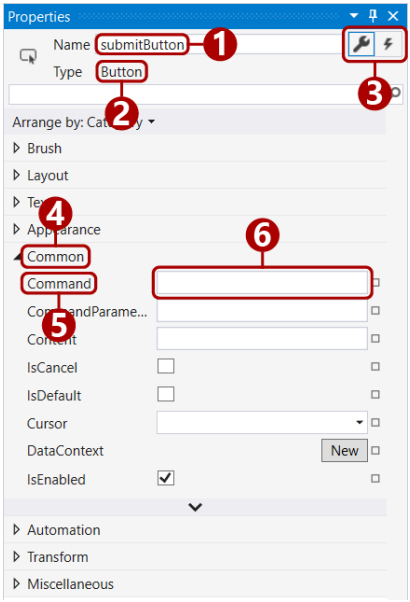


Fig. 18. Select the main menu item to open the Properties window

the required element in the Toolbox. In this case, it is placed as a child element for the active (selected) one within the markup.

To configure the created items, you need another window — Properties. In order to open this window, it is necessary to select the menu item View → Properties Window, or press F4 (Fig. 18).

The Properties window (Fig. 19) allows you to configure the properties and events of the markup elements. This window always displays information about the active (selected) markup



1. A text box containing the name of the control needed to interact with it from the code.
2. A mark that displays the data type of the control.
3. Buttons to switch the Properties window to display properties or events.
4. A mark that displays the name of the property category.
5. A mark that displays the name of the property.
6. A control that allows you to edit the value of a property.

Fig. 19. The Properties window

element. This window is arranged very simply. The properties of each element are divided into categories (color, location, text, etc.) and are displayed as two columns: the property name and the control for setting the property value.

The XAML editor and the Properties window are fully synchronized. It means that changing the markup by means of the XAML editor in some way, the corresponding values in the Properties window will also change, and vice versa.

9. Layout

When creating the user interface of the application, a lot of effort is spent to make it attractive and convenient to use. But no less effort is required its other aspect — flexibility. The flexibility of the interface should be understood as its ability to cope with changes in the size of the window and its filling. When changing the size of a window or part of it, the interface must adapt to the new dimensions in an adequate way, if necessary, increasing or decreasing the size of certain parts of it.

Another, no less common, reason that requires changes to the interface is to change the contents of the controls. For example, localized applications support multiple options for translating their interface into other languages. The words and phrases that make up the interface in different languages have different lengths, and, consequently, the sizes, which is a problem when using fixed sizes and positioning controls. This is a problem, because an interface designed using controls that perfectly fit the strings of one language will look quite different when using a different language: some elements will trim their content, showing only a portion of the line, and others will contain too much unused space. All these problems are connected with layout, i.e. with the location of the elements and their dimensions.

Using technologies that preceded the advent of WPF, the ability to manage the layout of the interface were, to put it mildly, not very extensive. Basically, they boiled down to specifying a fixed size and location for each control. If the application should have had the ability to resize the window,

the developer had to write code that resized and positioned the controls manually. Some technologies allowed automating the most basic scenarios of interface behavior when resizing. For example, in Windows Forms, there are concepts of anchoring and docking an element to the sides of the window. However, this only helps in cases with a very simple interface. In more complex situations, you had to develop your own algorithm that describes the behavior of the controls.

WPF solves the problems described above by changing the interface layout process. Instead of specifying fixed coordinates and size for each element of the interface, it is suggested to use a set of panels. Each panel has its own logic of ordering the elements placed in it.

At the core of the layout model laid out in WPF, there are two basic rules:

- Controls do not need to be fixed in size. Instead, their sizes should be determined by their contents and the logic of the panel in which they are located. At the same time, the elements can be limited by setting the minimum and maximum sizes, which will allow them to adapt to the changes, but only in the specified range.
- The controls should not have hard-coded coordinates. They should be arranged using the logic of the panels in which they are located and the indentation given to them.

As with any rule, there are exceptions to these rules. There are situations in which you need to specify a fixed size or location of the element. You should try to avoid them if possible and regard them as an extreme measure, as this is contrary to the WPF layout philosophy.

9.1. Stages of the Layout

Before proceeding to consider the distinctive features of each of the existing panels, it is worth considering a general principle of their work.

The arrangement of the elements is calculated in several situations: on the first display and if something happens that affects the size or location (for example, changing the content or the size of the window). The layout takes place in two stages: the measurement stage and the arrangement stage.

At the measurement stage, all elements of the visual tree are traversed and the preferred sizes are obtained for each of them. To do this, the System.Windows.UIElement.Measure method is called, and an available size, which can affect the preferred size of the element, is transferred to it whenever possible. For example, if an item that displays text has a word wrap enabled, such an element will require more height when the width of the available space is reduced. However, it will not always be known how much space is available. Elements containing scrollable areas provide an infinite "virtual" space, i.e. they can contain elements of any size, while showing only a part of them. In such cases, a value describing an infinitely accessible space will be transferred as an available space. Proceeding from this, the layout is of two kinds: limited and unlimited.

Limited layout occurs in cases where it is known how much space is available, which must be divided between the elements. Unlimited layout occurs in situations where the size of available space is unknown or unlimited. Also it is worth noting that when composing the layout there is a distinction between vertical and horizontal space. As an example, let's imagine that

an area with a vertical scroll bar is only available. This means that it has an unlimited vertical space and a limited horizontal space. In this case, a certain number describing the width of the available area and the positive infinity (System.Double.PositiveInfinity) as the height will be transferred to the System.Windows.UIElement.Measure method.

When traversing, controls cannot require unlimited space. They are required to return the final preferred size no matter how much space is available. In the case of unlimited layout, the elements return the minimum size they need. This is also called 'size to content'. In fact, unlimited layout is used much more often than it might seem at first glance. This kind of layout is applied every time you need to know the "real" dimensions of the element. This is very useful even in simple operations such as right alignment or centering. In these situations, you need to know the size of the element to add the necessary indents. Therefore, regardless of the interface, a mixture of limited and unlimited layout is often used for calculating the location of elements.

After the completion of the first stage, it becomes known what dimensions are necessary for each element of the visual tree, and a second traversal is made along it. This time, the System.Windows.UIElement.Arrange method is called for each element to which the calculated size and location of the element is transferred. Obviously, not all elements can always get the preferred size. In these cases, different things can happen. Some elements can be truncated, and some can be displayed normally even with a smaller size. For example, the element that displays an image, will require the size of the image itself as the preferred size. At the same time, having

received a smaller size as the calculated one, the image can be simply compressed and shown in a smaller version.

This two-stage layout looks like this for developers. Instead of calculating the size and location of the elements themselves, the developer describes the layout requirements in a declarative form in the form of XAML markup, thereby shifting the work of calculating the exact values to the shoulders of WPF.

10. Hardware-Independent Units

In WPF, the dimensions are often specified in so-called hardware-independent units, whose size is 1/96 of an inch. These units are used instead of real pixels in order to get the same size of the elements of the user interface on any screen. In fact, this means that if you set the element size to 96 such units, it should be 1 inch, no matter how many pixels are placed in inch in the monitor you are using.

11. Panels

In WPF, all panels are inherited from the base class System.Windows.Controls.Panel (Fig. 20).

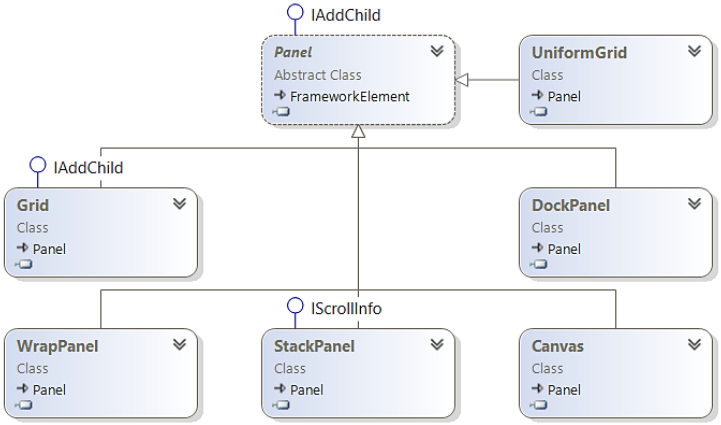


Fig. 20. Diagram of classes that describe panels

Properties of the System.Windows.Controls.Panel class:

- **Background** (System.Windows.Media.Brush). Gets or sets the brush that is used to fill the area between the borders of the panel. The default value is null.
- **Children** (System.Windows.Controls.UIElementCollection). Gets a collection of items placed in the panel. This collection stores only child elements of the first level. The default value is an empty collection.
- **IsItemsHost** (System.Boolean). Gets or sets a value that indicates whether the panel is a container for child elements generated by controls that are inherited from the System.Windows.Controls.ItemsControls class. true — if it is; otherwise, false. The default value is false.

Attached properties of the System.Windows.Controls.Panel class:

- **ZIndex** (System.Int32). Gets or sets the z-order value for the item in the panel. The higher the value is, the more likely the element will appear in the foreground. For example, an element with a value of z-order 5 will be shown on top of the element with a value of 4. This property can also take negative values. In the case where two elements have the same z-order value, they will be displayed in the order of appearance in the visual tree. In other words, the element that was declared earlier will be shown earlier.

11.1. The StackPanel

The System.Windows.Controls.StackPanel panel organizes the elements placed in it into a horizontal or vertical stack, i.e. builds them one after another.

Properties of the System.Windows.Controls.StackPanel class:

- **Orientation** (System.Windows.Controls.Orientation). Gets or sets the child ordering mode. The default value is System.Windows.Controls.Orientation.Vertical.

In order to specify the ordering of items, you must use the System.Windows.Controls.Orientation enumeration, which contains the following options:

- Horizontal. The panel arranges the items in the horizontal stack.
- Vertical. The panel arranges the items in the vertical stack.

Vertically oriented panel disposes the elements placed in it from top to bottom, in the order of their declaration in the markup. The height of each element is set by the value nec-

essary to display its contents. At the same time, the elements are stretched across the panel in width.

The following markup fragment demonstrates a vertically oriented panel (the complete example is in the Wpf.Panels.StackPanel.Vertical.Xaml folder):

XAML

```
<StackPanel>
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
    <Button>Button 5</Button>
</StackPanel>
```

The result of the above markup is shown in Fig. 21.

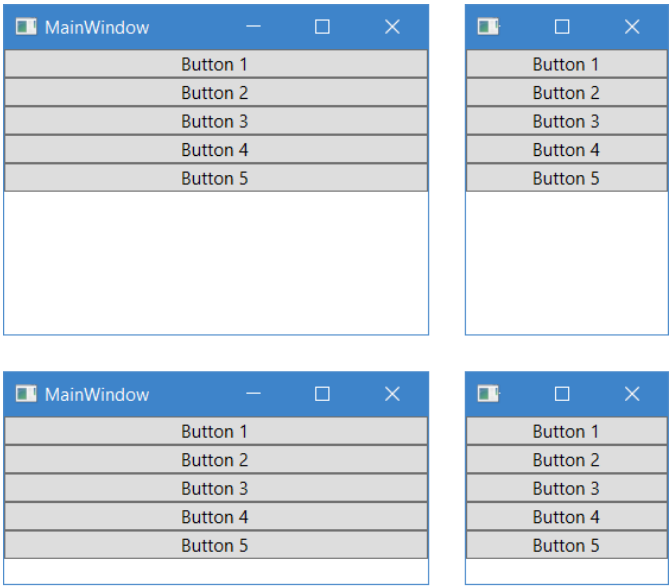


Fig. 21. Layout of controls using the vertically oriented StackPanel

The following code corresponds to the markup described above (the complete example is in Wpf.Panels.StackPanel.Vertical.CSharp):

C #

```
var button1 = new Button { Content = "Button 1" };
var button2 = new Button { Content = "Button 2" };
var button3 = new Button { Content = "Button 3" };
var button4 = new Button { Content = "Button 4" };
var button5 = new Button { Content = "Button 5" };

var stackPanel = new StackPanel();

stackPanel.Children.Add(button1);
stackPanel.Children.Add(button2);
stackPanel.Children.Add(button3);
stackPanel.Children.Add(button4);
stackPanel.Children.Add(button5);
```

By default, the System.Windows.Controls.StackPanel panel arranges the items in the vertical stack. In order to change the sorting order from vertical to horizontal, you must set the System.Windows.Controls.StackPanel.Orientation property to System.Windows.Controls.Orientation.Horizontal.

Horizontally oriented panel disposes the elements placed in it from left to right, in the order of their declaration in the markup. The width of each element is given by the value necessary to display its contents. In this case, the elements are stretched to the height along the entire panel.

The following markup fragment demonstrates a horizontally oriented panel (the complete example is in the folder Wpf.Panels.StackPanel.Horizontal.Xaml):

XAML

```
<StackPanel Orientation="Horizontal">
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
    <Button>Button 5</Button>
</StackPanel>
```

The result of the above markup is shown in Fig. 22.

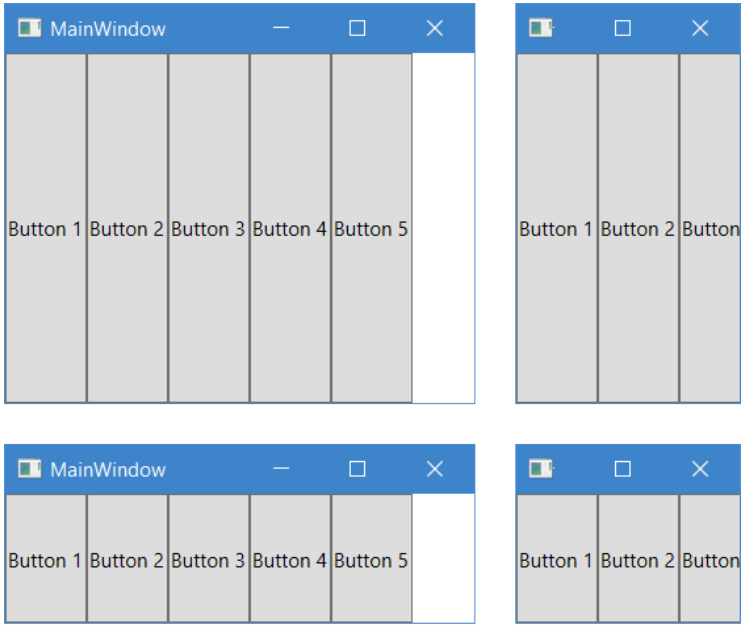


Fig. 22. Layout of controls using the horizontally oriented StackPanel

The following code corresponds to the markup described above (the complete example is in Wpf.Panels.StackPanel.Horizontal.CSharp):

C#

```
var button1 = new Button { Content = "Button 1" };
var button2 = new Button { Content = "Button 2" };
var button3 = new Button { Content = "Button 3" };
var button4 = new Button { Content = "Button 4" };
var button5 = new Button { Content = "Button 5" };
var stackPanel = new StackPanel { Orientation =
                 Orientation.Horizontal };
stackPanel.Children.Add(button1);
stackPanel.Children.Add(button2);
stackPanel.Children.Add(button3);
stackPanel.Children.Add(button4);
stackPanel.Children.Add(button5);
```

11.2. The WrapPanel

The System.Windows.Controls.WrapPanel panel organizes the elements placed on it in a horizontal or vertical stack, but, unlike the System.Windows.Controls.StackPanel panel, if the elements do not fit within the available panel space, then those that do not fit are transferred to another row or column.

Properties of the System.Windows.Controls.WrapPanel class:

- **Orientation** (System.Windows.Controls.Orientation). Gets or sets the child ordering mode. The default value is System.Windows.Controls.Orientation.Horizontal.

Horizontally oriented panel disposes the elements placed in it from left to right, in the order of their declaration in the markup. The width and height of each element is specified by the value necessary to display its contents. In this case, if there are elements with different heights in one line, then all will be stretched to the maximum of the existing ones.

The following markup fragment demonstrates a horizontally oriented panel (the complete example is in the Wpf.Panels.WrapPanel.Horizontal.Xaml folder):

XAML

```
<WrapPanel>
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
    <Button>Button 5</Button>
</WrapPanel>
```

The result of the above markup is shown in Fig. 23.

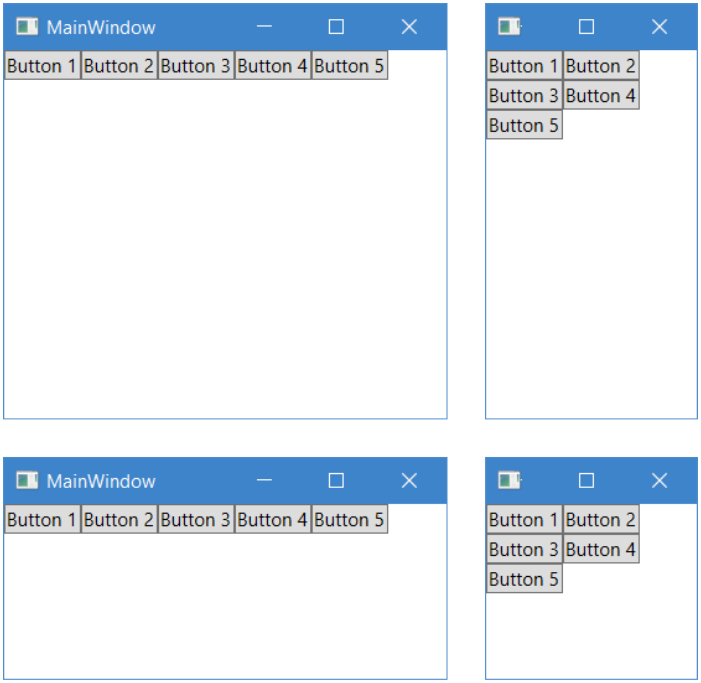


Fig. 23. Layout of controls using the horizontally oriented WrapPanel

The following code corresponds to the markup described above (the complete example is in Wpf.Panels.WrapPanel.Horizontal.CSharp):

C #

```
var button1 = new Button { Content = "Button 1" };
var button2 = new Button { Content = "Button 2" };
var button3 = new Button { Content = "Button 3" };
var button4 = new Button { Content = "Button 4" };
var button5 = new Button { Content = "Button 5" };
var wrapPanel = new WrapPanel();
wrapPanel.Children.Add(button1);
wrapPanel.Children.Add(button2);
wrapPanel.Children.Add(button3);
wrapPanel.Children.Add(button4);
wrapPanel.Children.Add(button5);
```

By default, the System.Windows.Controls.WrapPanel panel arranges the items in the horizontal stack. In order to change the ordering type from horizontal to vertical, you must set the System.Windows.Controls.WrapPanel.Orientation property to System.Windows.Controls.Orientation.Vertical.

Vertically oriented panel disposes the elements placed in it from top to bottom, in the order of their declaration in the markup. The width and height of each element is specified by the value necessary to display its contents. In this case, if there are elements with different widths in one column, then all will be stretched to the maximum of the existing ones.

The following markup fragment demonstrates a vertically oriented panel (the complete example is in the Wpf.Panels.WrapPanel.Vertical.Xaml folder):

XAML

```
<WrapPanel Orientation="Vertical">
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
    <Button>Button 5</Button>
</WrapPanel>
```

The result of the above markup is shown in Fig. 24.

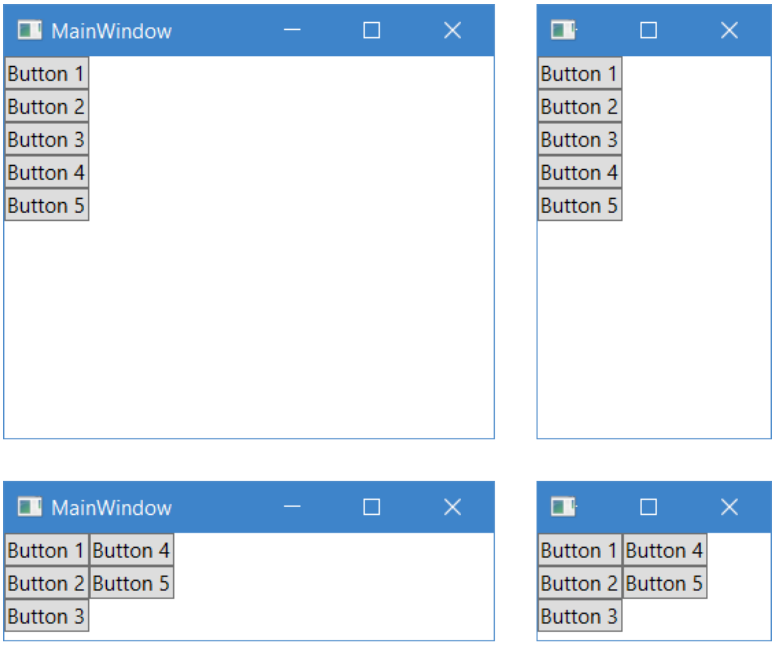


Fig. 24. Layout of controls using a vertically oriented WrapPanel

The following code corresponds to the markup described above (the complete example is in Wpf.Panels.WrapPanel.Vertical.CSharp):

C#

```
var button1 = new Button { Content = "Button 1" };
var button2 = new Button { Content = "Button 2" };
var button3 = new Button { Content = "Button 3" };
var button4 = new Button { Content = "Button 4" };
var button5 = new Button { Content = "Button 5" };
var wrapPanel = new WrapPanel { Orientation =
                Orientation.Vertical };
wrapPanel.Children.Add(button1);
wrapPanel.Children.Add(button2);
wrapPanel.Children.Add(button3);
wrapPanel.Children.Add(button4);
wrapPanel.Children.Add(button5);
```

11.3. The DockPanel

The System.Windows.Controls.DockPanel panel organizes the elements placed in it docking them to one of its borders.

Properties of the System.Windows.Controls.DockPanel class:

- **LastChildFill** (System.Boolean). Gets or sets a value that indicates whether the last element placed on the panel should be stretched to the remaining unallocated space. true if the last element should occupy all the remaining space; otherwise, false. The default value is true.

Attached properties of the System.Windows.Controls.DockPanel class:

- **Dock** (System.Windows.Controls.Dock). Gets or sets the side of the panel to which the element should be docked.

Elements placed in the panel are docked in the order of their declaration. Each next element uses only the space left

after docking the previous one. In order to specify the kind of docking an element, you must use the attached panel property — System.Windows.Controls.DockPanel.Dock.

The following markup fragment demonstrates the panel (the complete example is in the Wpf.Panels.DockPanel.Xaml folder):

XAML

```
<DockPanel>
    <Button DockPanel.Dock="Top">Button 1</Button>
    <Button DockPanel.Dock="Bottom">Button 2</Button>
    <Button DockPanel.Dock="Left">Button 3</Button>
    <Button DockPanel.Dock="Top">Button 4</Button>
    <Button>Button 5</Button>
</DockPanel>
```

The result of the above markup is shown in Fig. 25.

The result may make you wonder why both buttons docked to the top of the panel look different. In order to answer this question, let's consider the algorithm of this panel.

As it was said earlier, when calculating the size and location of elements, the panel processes them in the order of declaration, i.e. first the first button will be processed. It should be docked to the upper border of the panel and at the same time it can use the entire panel space. Docking to the top side of the panel, the element is placed as high as possible (to the very edge of the panel or under other docked elements), occupying the entire available width. The size of the height is calculated as the minimum possible, i.e. under the contents. The second button is docked to the bottom panel exactly according to the same principle (stretch over the entire width of the panel

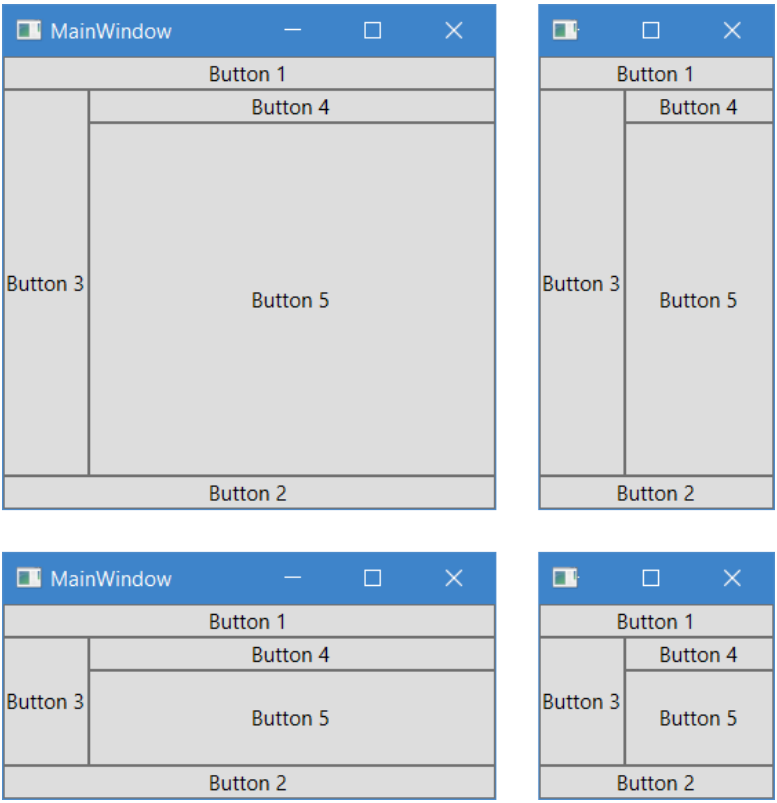


Fig. 25. Layout of controls using the DockPanel

and the height of the content). The third button should be docked to the left side. When an element docks to the left or right border of the panel, it is stretched to the height of the entire free space, and to the width of the content. Therefore, docking to the left side, the button stretched out in height not to the entire panel size, but only to the height of the remaining (unused) area. The fourth button, docking to the top side of the panel, is placed under the first one and also stretched to the entire available width. In this case, it also does not occupy the width of the entire panel, since part of the area in this

place has already been given to the third button. The last, fifth button occupies all the remaining unused panel space, due to the fact that the property of the System.Windows.Controls.DockPanel.LastChildFill panel is set to true by default.

The following code corresponds to the markup described above (the complete example is in Wpf.Panels.DockPanel.CSharp):

C#

```
var button1 = new Button { Content = "Button 1" };
var button2 = new Button { Content = "Button 2" };
var button3 = new Button { Content = "Button 3" };
var button4 = new Button { Content = "Button 4" };
var button5 = new Button { Content = "Button 5" };
var dockPanel = new DockPanel();
dockPanel.Children.Add(button1);
dockPanel.Children.Add(button2);
dockPanel.Children.Add(button3);
dockPanel.Children.Add(button4);
dockPanel.Children.Add(button5);
DockPanel.SetDock(button1, Dock.Top);
DockPanel.SetDock(button2, Dock.Bottom);
DockPanel.SetDock(button3, Dock.Left);
DockPanel.SetDock(button4, Dock.Top);
```

11.4. The Grid Panel

The System.Windows.Controls.Grid panel organizes the elements placed into it in the form of a grid with customizable row and column sizes.

Properties of the System.Windows.Controls.Grid class:

- **ColumnDefinitions** (System.Windows.Controls.ColumnDefinitionCollection). Gets the collection of grid columns.

The default value is an empty collection.

- **RowDefinitions** (System.Windows.Controls.RowDefinitionCollection). Gets the collection of grid rows. The default value is an empty collection.
- **ShowGridLines** (System.Boolean). Gets or sets a value that indicates whether the grid should be displayed. true if the grid should be displayed; otherwise, false. The default value is false.

Attached properties of the System.Windows.Controls.Grid class:

- **Column** (System.Int32). Gets or sets the index of the grid column in which the element is to be located. In order to place an element in the first column, you must specify a value of 0, in the second — 1, etc.
- **ColumnSpan** (System.Int32). Gets or sets the number of columns that the element should occupy, starting with the one in which it is located.
- **Row** (System.Int32). Gets or sets the index of the grid row in which the element is to be located. In order to place the element in the first row, you must specify a value of 0, in the second — 1, etc.
- **RowSpan** (System.Int32). Gets or sets the number of rows that the item should occupy, starting with the one in which it is located.

When describing the panel, you must specify the number of rows and columns of the resulting grid. If necessary, you can also set the height for each row and the width for each column. The elements placed in the panel are arranged in cells formed by the intersection of rows and columns. If you omit

the description of the rows, then one row will still be present in the grid. A similar situation occurs with the columns. Thus, if you ignore the declaration of rows and columns completely, then the grid will consist of only one cell.

Cells cannot be combined with each other, but elements placed in the panel can be set to the parameter responsible for how many rows and/or columns it should occupy.

If several elements are placed in the same cell, they will overlap. In this case, the elements will be displayed in the order of their declaration, i.e. the one declared later will be on top of that which was declared earlier.

In order to specify the cell in which it should be placed, it is necessary to use the attached panel properties: System.Windows.Controls.Grid.Row and System.Windows.Controls.Grid.Column. If any of them (or both) are not specified, the value of 0 will be taken for both the row and the column.

To describe the rows, use the System.Windows.Controls.RowDefinition class, and to describe the columns, use System.Windows.Controls.ColumnDefinition.

Properties of the System.Windows.Controls.RowDefinition class:

- **ActualHeight** (System.Double). Gets the calculated row height after two stages of the layout. The default value is 0.0.
- **Height** (System.Windows.Controls.GridLength). Gets or sets the height of the row. The default value is 1.0.
- **MaxHeight** (System.Double). Gets or sets the maximum possible row height. The default value is System.Double.PositiveInfinity.
- **MinHeight** (System.Double). Gets or sets the minimum possible row height. The default value is 0.0.

Properties of the System.Windows.Controls.ColumnDefiniton class:

- **ActualWidth** (System.Double). Gets the calculated column width after two stages of the layout. The default value is 0.0.
- **MaxWidth** (System.Double). Gets or sets the maximum possible column width. The default value is System.Double.PositiveInfinity.
- **MinWidth** (System.Double). Gets or sets the minimum possible column width. The default value is 0.0.
- **Width** (System.Windows.Controls.GridLength). Gets or sets the width of the column. The default value is 1.0.

The following markup fragment demonstrates a variant of the grid, in which the space between the columns and the rows is divided equally (the complete example is in the Wpf.Panels.Grid.Uniform.Xaml folder):

XAML

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Grid.Column="0" Grid.Row="0">1</Button>
    <Button Grid.Column="1" Grid.Row="0">2</Button>
    <Button Grid.Column="2" Grid.Row="0">3</Button>
    <Button Grid.Column="3" Grid.Row="0">4</Button>
```

```
    <Button Grid.Column="0" Grid.Row="1">5</Button>
    <Button Grid.Column="1" Grid.Row="1">6</Button>
    <Button Grid.Column="2" Grid.Row="1">7</Button>
    <Button Grid.Column="3" Grid.Row="1">8</Button>
    <Button Grid.Column="0" Grid.Row="2">9</Button>
    <Button Grid.Column="1" Grid.Row="2">10</Button>
    <Button Grid.Column="2" Grid.Row="2">11</Button>
    <Button Grid.Column="3" Grid.Row="2">12</Button>
</Grid>
```

The result of the above markup is shown in Fig. 26.

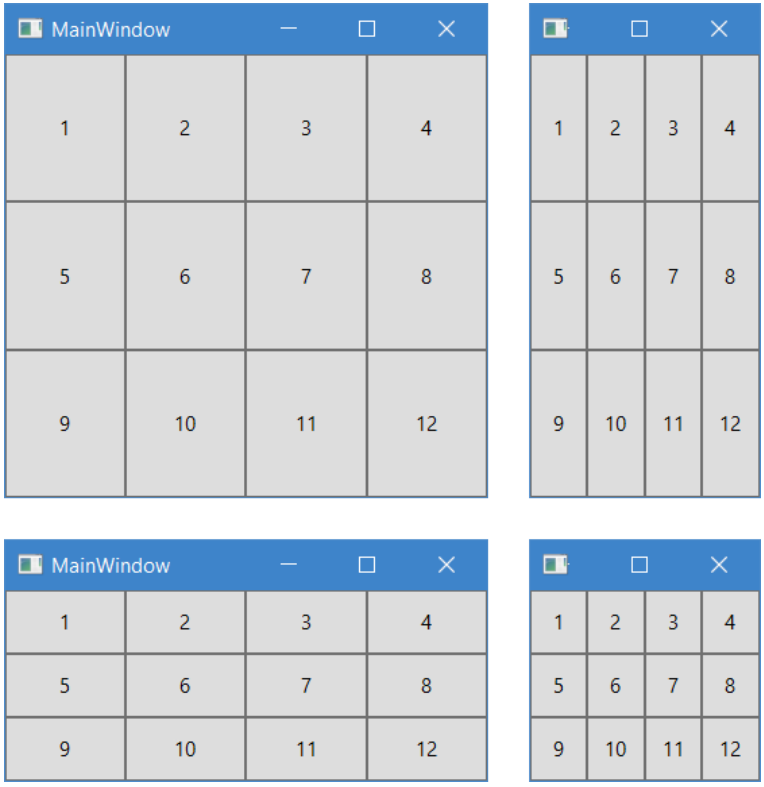


Fig. 26. Layout of controls using the Grid panel

The following code corresponds to the markup described above (the complete example is in Wpf.Panels.Grid.Uniform.CSharp):

C#

```
var button1 = new Button { Content = 1 };
var button2 = new Button { Content = 2 };
var button3 = new Button { Content = 3 };
var button4 = new Button { Content = 4 };
var button5 = new Button { Content = 5 };
var button6 = new Button { Content = 6 };
var button7 = new Button { Content = 7 };
var button8 = new Button { Content = 8 };
var button9 = new Button { Content = 9 };
var button10 = new Button { Content = 10 };
var button11 = new Button { Content = 11 };
var button12 = new Button { Content = 12 };

var grid = new Grid();
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());

grid.Children.Add(button1);
grid.Children.Add(button2);
grid.Children.Add(button3);
grid.Children.Add(button4);
grid.Children.Add(button5);
grid.Children.Add(button6);
grid.Children.Add(button7);
grid.Children.Add(button8);
grid.Children.Add(button9);
grid.Children.Add(button10);
```

```
grid.Children.Add(button11);
grid.Children.Add(button12);

Grid.SetColumn(button1, 0);
Grid.SetRow(button1, 0);
Grid.SetColumn(button2, 1);
Grid.SetRow(button2, 0);
Grid.SetColumn(button3, 2);
Grid.SetRow(button3, 0);
Grid.SetColumn(button4, 3);
Grid.SetRow(button4, 0);
Grid.SetColumn(button5, 0);
Grid.SetRow(button5, 1);
Grid.SetColumn(button6, 1);
Grid.SetRow(button6, 1);
Grid.SetColumn(button7, 2);
Grid.SetRow(button7, 1);
Grid.SetColumn(button8, 3);
Grid.SetRow(button8, 1);
Grid.SetColumn(button9, 0);
Grid.SetRow(button9, 2);
Grid.SetColumn(button10, 1);
Grid.SetRow(button10, 2);
Grid.SetColumn(button11, 2);
Grid.SetRow(button11, 2);
Grid.SetColumn(button12, 3);
Grid.SetRow(button12, 2);
```

When describing columns and rows, you do not need to specify explicitly what width and height they should have. In this case, the entire available width of the panel will be divided equally into the columns, and the height — into the rows. However, it may be necessary to specify these parameters according to some criterion.

The dimension of each column and row can be specified in one of the following ways:

- Explicit indication. In this case, the column width or row height will be equal to the specified value.
- Automatic calculation. In this case, the width of the column or the height of the row will be minimally possible to fully contain its content (size to content).
- Indication of ratios. In this case, several columns or rows indicate how their width or height should relate to each other. For example, 1:2 or 3:5.

To describe the dimension of a column or row, use the System.Windows.GridLength structure.

Constructors of the System.Windows.GridLength structure:

- **GridLength**(pixels). Initializes an instance of the structure with the specified number of hardware-independent units.
- **GridLength**(value, type). Initializes the instance of the structure with the specified value and the type of this value.

Static properties of the System.Windows.GridLength structure:

- **Auto** (System.Windows.GridLength). Gets an object that describes the automatically calculated dimension.
- Properties of the System.Windows.GridLength structure:
- **GridUnitType** (System.Windows.GridUnitType). Gets a value that describes the type of the dimension value that is contained in the object.
- **IsAbsolute** (System.Boolean). Gets a value that indicates whether the value stored in the object is a value in hardware-independent units. true if the value of the System.

Windows.GridLength.GridUnitType property is set to System.Windows.GridUnitType.Pixel; otherwise, false.

- **IsAuto** (System.Boolean). Gets a value that indicates whether the value stored in the object is an automatically calculated value, depending on the content. true if the value of the System.Windows.GridLength.GridUnitType property is equal to System.Windows.GridUnitType.Auto; otherwise, false.
- **IsStar** (System.Boolean). Gets a value that indicates whether the value stored in the object is a ratio. true if the value of the System.Windows.GridLength.GridUnitType property is set to System.Windows.GridUnitType.Star; otherwise, false.
- **Value** (System.Double). Gets the value stored in the object.

The following markup fragment demonstrates a variant of the grid with different options for specifying the dimensions of columns and rows (the complete example is in the Wpf.Panels.Grid.NonUniform.Xaml folder):

XAML

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="30"/>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="2*"/>
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
        <RowDefinition Height="40"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="3*"/>
    </Grid.RowDefinitions>
```

```
    <Button Grid.Column="0" Grid.Row="0">1</Button>
    <Button Grid.Column="1" Grid.Row="0">2</Button>
    <Button Grid.Column="2" Grid.Row="0">3</Button>
    <Button Grid.Column="3" Grid.Row="0">4</Button>
    <Button Grid.Column="0" Grid.Row="1">5</Button>
    <Button Grid.Column="1" Grid.Row="1">6</Button>
    <Button Grid.Column="2" Grid.Row="1">7</Button>
    <Button Grid.Column="3" Grid.Row="1">8</Button>
    <Button Grid.Column="0" Grid.Row="2">9</Button>
    <Button Grid.Column="1" Grid.Row="2">10</Button>
    <Button Grid.Column="2" Grid.Row="2">11</Button>
    <Button Grid.Column="3" Grid.Row="2">12</Button>
    <Button Grid.Column="0" Grid.Row="3">13</Button>
    <Button Grid.Column="1" Grid.Row="3">14</Button>
    <Button Grid.Column="2" Grid.Row="3">15</Button>
    <Button Grid.Column="3" Grid.Row="3">16</Button>
</Grid>
```

The result of the above markup is shown in Fig. 27.

In the markup above, all three forms of specifying the dimension are demonstrated:

- **Explicit indication.** In this case, a value indicating the number of hardware-independent units is specified as the value.
- **Automatic calculation.** In this case, the value is Auto.
- **Indication of ratios.** In this case, the value is indicated by a number and an asterisk (if the number is 1, then it can be omitted, i.e. * the same as 1*). For example, if one column has a value of 2*, and the other has 3*, then they are related as 2:3.

If you do not specify a height or width value for a column or row, the value will be set to *.

First of all, the panel space is divided between columns and rows, for which the size was specified fixed or "to con-

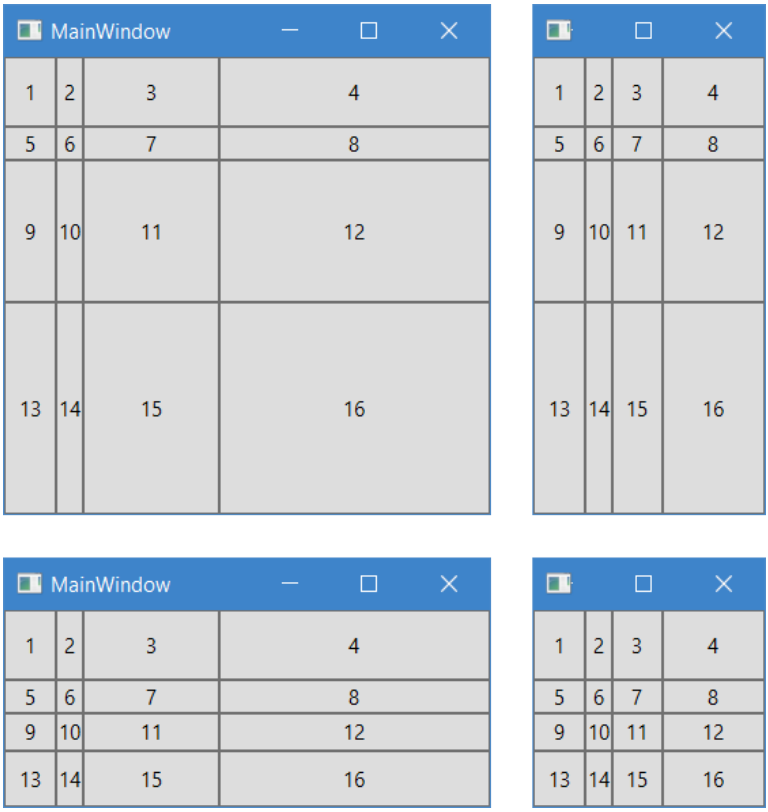


Fig. 27. Layout of controls using the Grid panel

tent." After this, all the remaining space is divided according to the indicated ratios.

The following code corresponds to the markup described above (the complete example is in Wpf.Panels.Grid.NonUniform.CSharp):

C#

```
var button1 = new Button { Content = 1 };
var button2 = new Button { Content = 2 };
var button3 = new Button { Content = 3 };
```

```
var button4 = new Button { Content = 4 };
var button5 = new Button { Content = 5 };
var button6 = new Button { Content = 6 };
var button7 = new Button { Content = 7 };
var button8 = new Button { Content = 8 };
var button9 = new Button { Content = 9 };
var button10 = new Button { Content = 10 };
var button11 = new Button { Content = 11 };
var button12 = new Button { Content = 12 };
var button13 = new Button { Content = 13 };
var button14 = new Button { Content = 14 };
var button15 = new Button { Content = 15 };
var button16 = new Button { Content = 16 };
var grid = new Grid();

grid.ColumnDefinitions.Add(new ColumnDefinition
     { Width = new GridLength(30.0) });
grid.ColumnDefinitions.Add(new ColumnDefinition
     { Width = GridLength.Auto });
grid.ColumnDefinitions.Add(
    new ColumnDefinition { Width =
    new GridLength(1.0, GridUnitType.Star) });
grid.ColumnDefinitions.Add(
    new ColumnDefinition { Width =
    new GridLength(2.0, GridUnitType.Star) });
grid.RowDefinitions.Add(new RowDefinition { Height =
    new GridLength(40.0) });
grid.RowDefinitions.Add(new RowDefinition { Height =
    GridLength.Auto });
grid.RowDefinitions.Add(
    new RowDefinition { Height =
    new GridLength(2.0, GridUnitType.Star) });
grid.RowDefinitions.Add(
    new RowDefinition { Height =
    new GridLength(3.0, GridUnitType.Star) });

grid.Children.Add(button1);
grid.Children.Add(button2);
```

```
grid.Children.Add(button3);
grid.Children.Add(button4);
grid.Children.Add(button5);
grid.Children.Add(button6);
grid.Children.Add(button7);
grid.Children.Add(button8);
grid.Children.Add(button9);
grid.Children.Add(button10);
grid.Children.Add(button11);
grid.Children.Add(button12);
grid.Children.Add(button13);
grid.Children.Add(button14);
grid.Children.Add(button15);
grid.Children.Add(button16);

Grid.SetColumn(button1, 0);
Grid.SetRow(button1, 0);
Grid.SetColumn(button2, 1);
Grid.SetRow(button2, 0);
Grid.SetColumn(button3, 2);
Grid.SetRow(button3, 0);
Grid.SetColumn(button4, 3);
Grid.SetRow(button4, 0);
Grid.SetColumn(button5, 0);
Grid.SetRow(button5, 1);
Grid.SetColumn(button6, 1);
Grid.SetRow(button6, 1);
Grid.SetColumn(button7, 2);
Grid.SetRow(button7, 1);
Grid.SetColumn(button8, 3);
Grid.SetRow(button8, 1);
Grid.SetColumn(button9, 0);
Grid.SetRow(button9, 2);
Grid.SetColumn(button10, 1);
Grid.SetRow(button10, 2);
Grid.SetColumn(button11, 2);
Grid.SetRow(button11, 2);
```

```
Grid.SetColumn(button12, 3);
Grid.SetRow(button12, 2);
Grid.SetColumn(button13, 0);
Grid.SetRow(button13, 3);
Grid.SetColumn(button14, 1);
Grid.SetRow(button14, 3);
Grid.SetColumn(button15, 2);
Grid.SetRow(button15, 3);
Grid.SetColumn(button16, 3);
Grid.SetRow(button16, 3);
```

In some situations, it is necessary for an element placed in the panel to occupy more than one row and/or column. In such situations, you must use the attached panel properties: System.Windows.Controls.Grid.RowSpan and System.Windows.Controls.Grid.ColumnSpan.

For each element, these properties are specified separately. If more than one item is in the same cell, and one of them is set to these properties in order to stretch it, then it will not affect the other elements in this cell.

The following markup fragment demonstrates a grid variant with elements that occupy more than one cell (the complete example is in the Wpf.Panels.Grid.Span.Xaml folder):

XAML

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
```

```
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Grid.Column="0" Grid.ColumnSpan="2"
            Grid.Row="0">1</Button>
    <Button Grid.Column="2" Grid.Row="0"
            Grid.RowSpan="2">2</Button>
    <Button Grid.Column="0" Grid.Row="1">3</Button>
    <Button Grid.Column="1" Grid.Row="1">4</Button>
    <Button Grid.Column="0" Grid.ColumnSpan="3"
            Grid.Row="2">5</Button>
</Grid>
```

The result of the above markup is shown in Fig. 28.

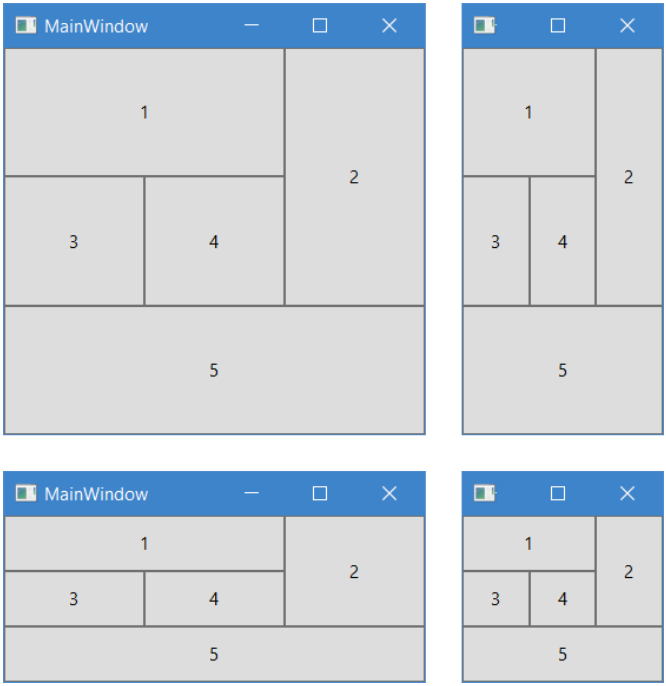


Fig. 28. Layout of the controls using the Grid panel

The following code corresponds to the markup described above (the complete example is in the Wpf.Panels.Grid.Span.CSharp):

C#

```
var button1 = new Button { Content = 1 };
var button2 = new Button { Content = 2 };
var button3 = new Button { Content = 3 };
var button4 = new Button { Content = 4 };
var button5 = new Button { Content = 5 };
var grid = new Grid();

grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.Children.Add(button1);
grid.Children.Add(button2);
grid.Children.Add(button3);
grid.Children.Add(button4);
grid.Children.Add(button5);

Grid.SetColumn(button1, 0);
Grid.SetRow(button1, 0);
Grid.SetColumn(button2, 2);
Grid.SetRow(button2, 0);
Grid.SetColumn(button3, 0);
Grid.SetRow(button3, 1);
Grid.SetColumn(button4, 1);
Grid.SetRow(button4, 1);
Grid.SetColumn(button5, 0);
Grid.SetRow(button5, 2);
Grid.SetColumnSpan(button1, 2);
Grid.SetColumnSpan(button5, 3);
Grid.SetRowSpan(button2, 2);
```

11.6. The UniformGrid Panel

The System.Windows.Controls.Primitives.UniformGrid panel organizes the elements placed in it in the form of a grid with the same sizes of rows and columns. This panel is a simplified version of the System.Windows.Controls.Grid panel.

Properties of the System.Windows.Controls.Primitives.UniformGrid class:

- **Columns** (System.Int32). Gets or sets the number of columns in the grid. The default value is 0.
- **FirstColumn** (System.Int32). Gets or sets the number of empty cells at the beginning of the first row. The default value is 0.
- **Rows** (System.Int32). Gets or sets the number of rows in the grid. The default value is 0.

The following markup fragment demonstrates the grid (the complete example is in the Wpf.Panels.UniformGrid.Xaml folder):

XAML

```
<UniformGrid Columns="3" Rows="2">
    <Button>1</Button>
    <Button>2</Button>
    <Button>3</Button>
    <Button>4</Button>
    <Button>5</Button>
    <Button>6</Button>
</UniformGrid>
```

The result of the above markup is shown in Fig. 29.

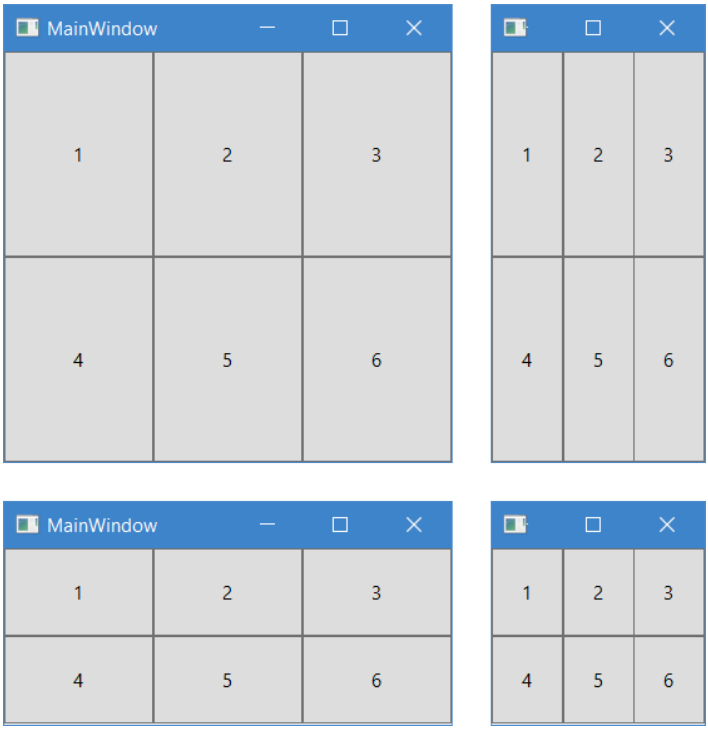


Fig. 29. Layout of controls using the UniformGrid panel

The following code corresponds to the markup described above (the complete example is in Wpf.Panels.UniformGrid.CSharp):

C#

```
var button1 = new Button { Content = 1 };
var button2 = new Button { Content = 2 };
var button3 = new Button { Content = 3 };
var button4 = new Button { Content = 4 };
var button5 = new Button { Content = 5 };
var button6 = new Button { Content = 6 };
var uniformGrid = new UniformGrid { Columns = 3,
                                    Rows = 2 };
```

```
uniformGrid.Children.Add(button1);
uniformGrid.Children.Add(button2);
uniformGrid.Children.Add(button3);
uniformGrid.Children.Add(button4);
uniformGrid.Children.Add(button5);
uniformGrid.Children.Add(button6);
```

11.6. The Canvas Panel

The System.Windows.Controls.Canvas panel organizes the elements placed in it according to exactly specified coordinates.

Attached properties of the System.Windows.Controls.Canvas class:

- **Bottom** (System.Double). Gets or sets the indent from the bottom border of the panel to the element. The value is specified in hardware-independent units.
- **Left** (System.Double). Gets or sets the indent from the left border of the panel to the element. The value is specified in hardware-independent units.
- **Right** (System.Double). Gets or sets the indent from the right border of the panel to the element. The value is specified in hardware-independent units.
- **Top** (System.Double). Gets or sets the indent from the top border of the panel to the element. The value is specified in hardware-independent units.

If you do not set a horizontal indent for the element, it will be aligned to the left edge of the panel, i.e. the same as setting the left indent to 0.0. If you do not set a vertical indent for

the element, it will be aligned along the top edge of the panel, i.e. the same as setting the top indent to 0.0.

If you specify mutually exclusive indentd, then the indents System.Windows.Controls.Canvas.Left and System.Windows.Controls.Canvas.Top will be priority.

The following markup fragment demonstrates the panel (the complete example is in the Wpf.Panels.Canvas.Xaml folder):

XAML

```
<Canvas>
    <Button>Button 1</Button>
    <Button Canvas.Left="70">Button 2</Button>
    <Button Canvas.Top="70">Button 3</Button>
    <Button Canvas.Bottom="70">Button 4</Button>
    <Button Canvas.Right="70">Button 5</Button>
    <Button Canvas.Left="70" Canvas.Top="70">Button 6
            </Button>
</Canvas>
```

The result of the above markup is shown in Fig. 30.

The following code corresponds to the above markup (the complete example is in Wpf.Panels.Canvas.CSharp):

C#

```
var button1 = new Button { Content = "Button 1" };
var button2 = new Button { Content = "Button 2" };
var button3 = new Button { Content = "Button 3" };
var button4 = new Button { Content = "Button 4" };
var button5 = new Button { Content = "Button 5" };
var button6 = new Button { Content = "Button 6" };

var canvas = new Canvas();
canvas.Children.Add(button1);
```

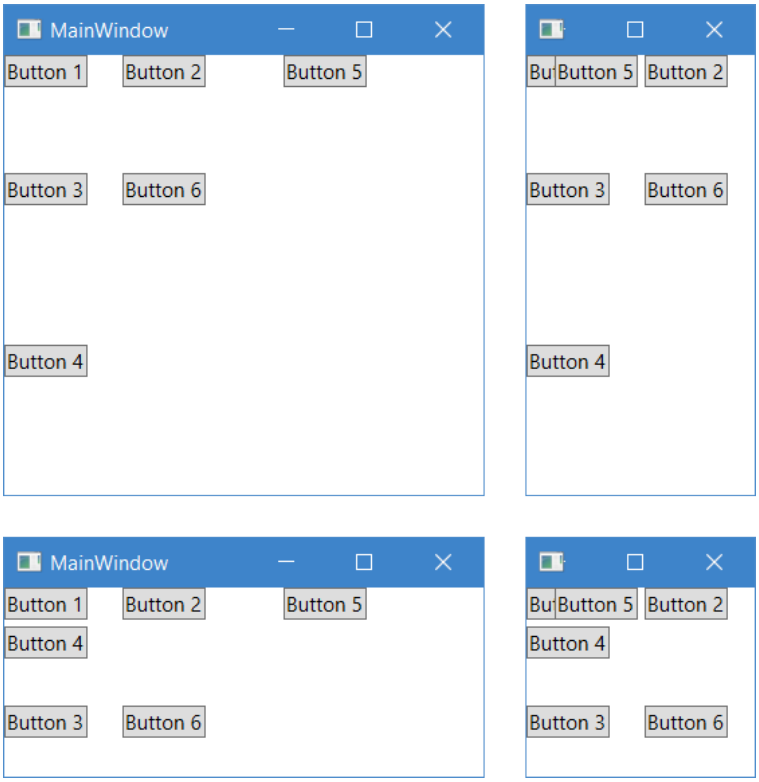


Fig. 30. Layout of controls using the Canvas panel

```
canvas.Children.Add(button2);
canvas.Children.Add(button3);
canvas.Children.Add(button4);
canvas.Children.Add(button5);
canvas.Children.Add(button6);

Canvas.SetLeft(button2, 70.0);
Canvas.SetTop(button3, 70.0);
Canvas.SetBottom(button4, 70.0);
Canvas.SetRight(button5, 70.0);
Canvas.SetLeft(button6, 70.0);
Canvas.SetTop(button6, 70.0);
```

12. Positioning of Elements

Virtually all elements that are in the visual tree are inherited from the System.Windows.FrameworkElement class. This section will discuss the properties of this class, which are used to more accurately position and control the size of visual elements.

Properties of the System.Windows.FrameworkElement class associated with positioning:

- **ActualHeight** (System.Double). Gets the actual (calculated) height of the element in hardware-independent units. The default value is 0.0.
- **ActualWidth** (System.Double). Gets the actual (calculated) width of the element in hardware-independent units. The default value is 0.0.
- **Height** (System.Double). Gets or sets the proposed height of the element in hardware-independent units. This value must be greater than or equal to 0.0. The default value is System.Double.NaN.
- **HorizontalAlignment** (System.Windows.HorizontalAlignment). Gets or sets the horizontal alignment of the element for those cases when it is placed inside another element (for example, a panel). The default value is System.Windows.HorizontalAlignment.Stretch.
- **Margin** (System.Windows.Thickness). Gets or sets the outer indents of an element. The default value is the System.Windows.Thickness object, which describes zero indents on all sides.
- **MaxHeight** (System.Double). Gets or sets the maximum allowed height of an element in hardware-independent

units. This value must be greater than or equal to 0.0. The default value is System.Double.PositiveInfinity.

- **MaxWidth** (System.Double). Gets or sets the maximum allowed width of an element in hardware-independent units. This value must be greater than or equal to 0.0. The default value is System.Double.PositiveInfinity.
- **MinHeight** (System.Double). Gets or sets the minimum allowed height of an element in hardware-independent units. This value must be greater than or equal to 0.0, but it cannot be equal to System.Double.PositiveInfinity or System.Double.NaN. The default value is 0.0.
- **MinWidth** (System.Double). Gets or sets the minimum allowed width of an element in hardware-independent units. This value must be greater than or equal to 0.0, but it cannot be equal to System.Double.PositiveInfinity or System.Double.NaN. The default value is 0.0.
- **VerticalAlignment** (System.Windows.VerticalAlignment). Gets or sets the vertical alignment of the element for the cases when it is placed inside another element (for example, a panel). The default value is System.Windows.VerticalAlignment.Stretch.
- **Width** (System.Double). Gets or sets the proposed width of the element in hardware-independent units. This value must be greater than or equal to 0.0. The default value is System.Double.NaN.

Properties of the System.Windows.Controls.Control class associated with positioning:

- **HorizontalContentAlignment** (System.Windows.HorizontalAlignment). Gets or sets the horizontal alignment

of the content of the element. The default value is System.Windows.HorizontalAlignment.Left.

- **Padding** (System.Windows.Thickness). Gets or sets the internal indents for the element (between the border and the content). The default value is the System.Windows.Thickness object, which describes zero indents on all sides.
- **VerticalContentAlignment** (System.Windows.VerticalAlignment). Gets or sets the vertical alignment of the content of the element. The default value is System.Windows.VerticalAlignment.Top.

12.1. Indents

The System.Windows.FrameworkElement.Margin property is responsible for which indents (margins) should be on each side of the element. Indents means the free space between the outer border of an element and the inner border of its parent element (for example, a panel). This property is specified in hardware-independent units and if it is not specified, the indents will be zero, i.e. will be absent. Each of the four sides of the element can be indented separately and indents can be different. To describe the indents, use the System.Windows.Thickness structure, which allows you to describe the thickness of the frame around a rectangular area.

Constructors of the System.Windows.Thickness structure:

- **Thickness**(uniformLength). Initializes an instance of the structure using the specified value for all sides.
- **Thickness**(left, top, right, bottom). Initializes the instance with the specified values for all sides.

Properties of the System.Windows.Thickness structure:

- **Bottom** (System.Double). Gets or sets the height of the bottom of the frame in hardware-independent units. The default value is 0.0.
- **Left** (System.Double). Gets or sets the width of the left side of the frame in hardware-independent units. The default value is 0.0.
- **Right** (System.Double). Gets or sets the width of the right side of the frame in hardware-independent units. The default value is 0.0.
- **Top** (System.Double). Gets or sets the height of the top of the frame in hardware-independent units. The default value is 0.0.

If two elements describe the indents relative to each other, these indents do not overlap, but are summed. For example, there are two buttons arranged horizontally one after the other. The first indicates the indent of 10 hardware-independent units, and the second on the left also indicates 10 hardware-independent units. This means that between the buttons there will be an indent of 20 hardware-independent units.

The System.Windows.Thickness structure has a type converter that allows you to use strings of a certain format to initialize properties of this type. There are several options for specifying indents in the markup.

If you specify only one number, then it will be used for all sides. The example below shows how to create a button with indents of 20 hardware-independent units on all sides.

XAML

```
<Button Margin="20">OK</Button>
```

If you specify two numbers separated by a comma, the first will be used for the left and right sides, and the second for the top and bottom.

XAML

```
<Button Margin="20,50">OK</Button>
```

If you specify four numbers separated by a comma, they will be used for each individual side in the order: left, upper, right, lower.

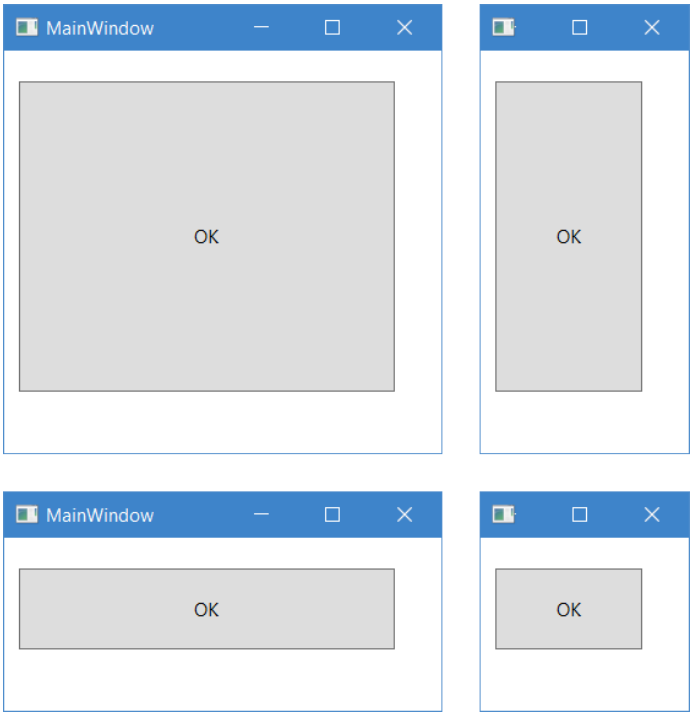


Fig. 31. Using the Margin property

XAML

```
<Button Margin="20,50,30,40">OK</Button>
```

As an example, let's look at the following markup fragment (the complete example is in Wpf.Properties.Margin.Xaml):

XAML

```
<Grid>
    <Button Margin="10,20,30,40">OK</Button>
</Grid>
```

The result shows a button with the following indents: 10.0 on the left, 20.0 on the top, 30.0 on the right, and 40.0 at the bottom (Fig. 31).

The following code corresponds to the above markup (the complete example is in Wpf.Properties.Margin.CSharp):

C#

```
var button = new Button { Content = "OK",
    Margin = new Thickness(10.0, 20.0, 30.0, 40.0) };

var grid = new Grid();
grid.Children.Add(button);
```

12.2. Alignments

Quite often there is a situation that the available space for an element is greater than it needs. In such situations, alignment properties are used to calculate exactly where to place the element or to stretch it to occupy all available space. Two properties can help: System.Windows.FrameworkElement.

HorizontalAlignment and System.Windows.FrameworkElement.VerticalAligment.

In order to specify the horizontal alignment, you must use the System.Windows.HorizontalAlignment enum, which contains the following options:

- Center. Center alignment.
- Left. Align to the left.
- Right. Align to the right.
- Stretch. The element must be stretched to the entire width provided.

As an example, let's look at the following markup fragment (the complete example is in Wpf.Properties.HorizontalAlignment.Xaml):

XAML

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>

    <Button Grid.Column="0"
            Grid.Row="0" HorizontalAlignment="Left">
            Left
    </Button>
    <Button Grid.Column="0"
            Grid.Row="1" HorizontalAlignment="Center">
            Center
    </Button>
    <Button Grid.Column="0"
            Grid.Row="2" HorizontalAlignment="Right">
```

```
            Right
    </Button>
    <Button Grid.Column="0"
            Grid.Row="3" HorizontalAlignment="Stretch">
             Stretch</Button>
</Grid>
```

The result demonstrates the creation of four buttons (Fig. 32). In case of using any alignment option, except for System.Windows.HorizontalAlignment.Stretch, the element in width will be adjusted according to the content.

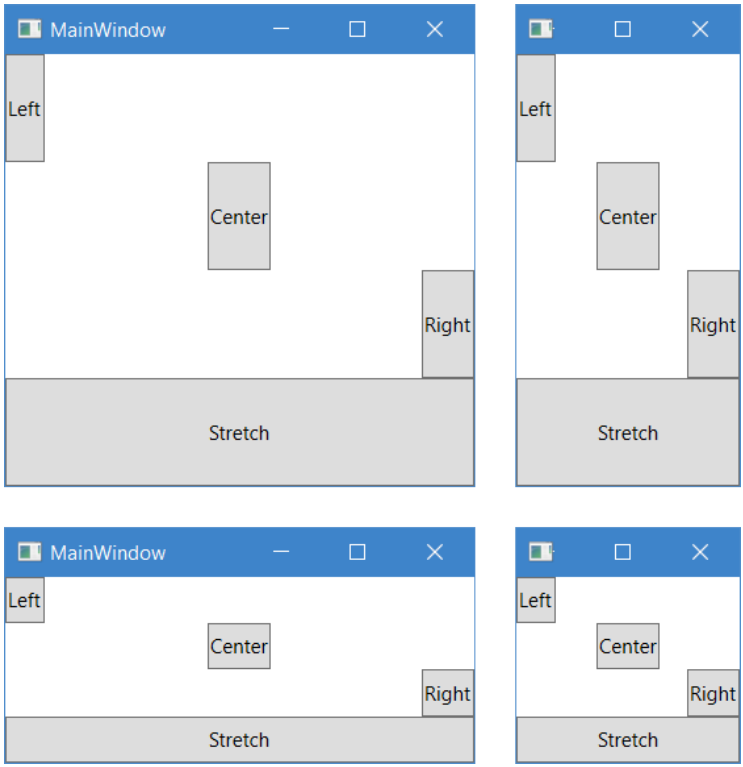


Fig. 32. Using the HorizontalAlignment property

The following code corresponds to the above markup (the complete example is in Wpf.Properties.HorizontalAlignment.CSharp):

C#

```
var button1 = new Button
{
    Content = "Left",
    HorizontalAlignment = HorizontalAlignment.Left
};
var button2 = new Button
{
    Content = "Center",
    HorizontalAlignment = HorizontalAlignment.Center
};
var button3 = new Button
{
    Content = "Right",
    HorizontalAlignment = HorizontalAlignment.Right
};
var button4 = new Button
{
    Content = "Stretch",
    HorizontalAlignment = HorizontalAlignment.Stretch
};
var grid = new Grid();
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.Children.Add(button1);
grid.Children.Add(button2);
grid.Children.Add(button3);
grid.Children.Add(button4);

Grid.SetRow(button1, 0);
```

```
Grid.SetRow(button2, 1);
Grid.SetRow(button3, 2);
Grid.SetRow(button4, 3);
```

In order to specify the vertical alignment, you must use the System.Windows.VerticalAlignment enumeration, which contains the following options:

- Bottom. Alignment at the bottom edge.
- Center. Center alignment.
- Stretch. The element must be stretched to the entire height provided.
- Top. Alignment at the top edge.

As an example, let's consider the following fragment of the markup (the complete example is in Wpf.Properties.VerticalAlignment.Xaml):

XAML

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Button Grid.Column="0" Grid.Row="0"
            VerticalAlignment="Top">Top
    </Button>

    <Button Grid.Column="1" Grid.Row="0"
            VerticalAlignment="Center">Center
    </Button>
```

```
    <Button Grid.Column="2" Grid.Row="0"
            VerticalAlignment="Bottom">Bottom
    </Button>
    <Button Grid.Column="3" Grid.Row="0"
            VerticalAlignment="Stretch">Stretch
    </Button>
</Grid>
```

The result demonstrates the creation of four buttons (Fig. 33). If you use any alignment option, except for System.Windows.VerticalAlignment.Stretch, the element will be adjusted according to the content in height.

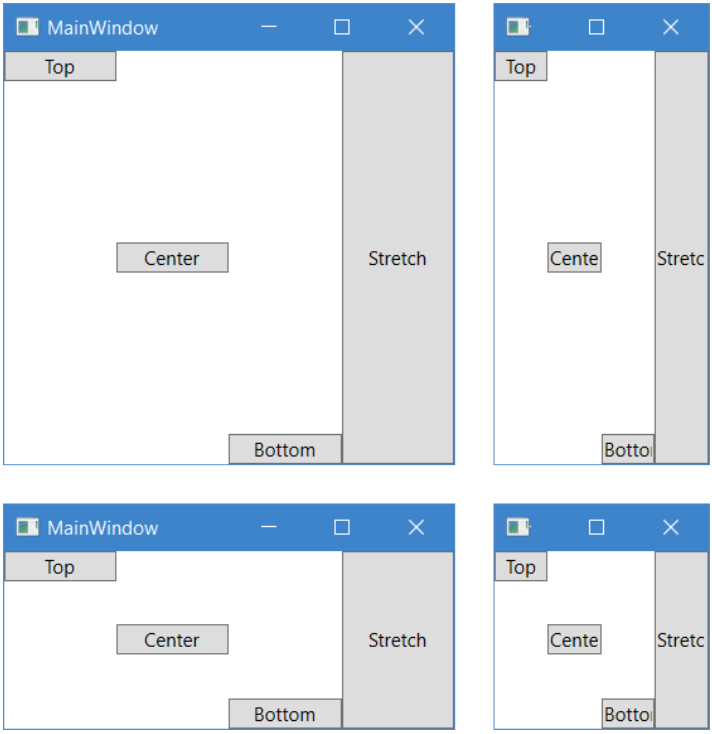


Fig. 33. Using the VerticalAlignment property

The following code corresponds to the markup described above (the complete example is in Wpf.Properties.VerticalAlignment.CSharp):

C#

```
var button1 = new Button
{
    Content = "Top",
    VerticalAlignment = VerticalAlignment.Top
};
var button2 = new Button
{
    Content = "Center",
    VerticalAlignment = VerticalAlignment.Center
};
var button3 = new Button
{
    Content = "Bottom",
    VerticalAlignment = VerticalAlignment.Bottom
};
var button4 = new Button
{
    Content = "Stretch",
    VerticalAlignment = VerticalAlignment.Stretch
};

var grid = new Grid();
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.Children.Add(button1);
grid.Children.Add(button2);
grid.Children.Add(button3);
grid.Children.Add(button4);
Grid.SetColumn(button1, 0);
```

```
Grid.SetColumn(button2, 1);
Grid.SetColumn(button3, 2);
Grid.SetColumn(button4, 3);
```

12.3. Sizes

Sometimes, the sizes offered to an element as a result of the layout calculation may not be the most optimal in terms of its appearance. For example, it is quite common practice to specify fixed sizes for buttons. If you allow the buttons to stretch to all available space, they often look ridiculous, and if you specify them to determine the size of the content, they may look too small.

Although fixed sizes may be appropriate for some elements, the use of this approach is not flexible and therefore should not be the default option when designing the interface. There is a more flexible approach to specifying the dimensions of an element: using the System.Windows.FrameworkElement.MinWidth and System.Windows.FrameworkElement.MaxWidth properties to specify the minimum and maximum widths, and System.Windows.FrameworkElement.MinHeight and System.Windows.FrameworkElement.MaxHeight to indicate the minimum and maximum height. These properties allow you to specify a valid range for the dimensions of the element.

As an example, let's look at the following markup fragment (the complete example is in Wpf.Properties.Width.Xaml):

XAML

```
<StackPanel>
    <Button Margin="5,5,5,0">1</Button>
    <Button Margin="5,5,5,0" Width="100">2</Button>
```

```
    <Button Margin="5,5,5,0" MinWidth="100">3</Button>
    <Button Margin="5,5,5,0" MaxWidth="100">4</Button>
    <Button Margin="5,5,5,0" MinWidth="100"
            Width="150">5</Button>
    <Button Margin="5,5,5,0" MinWidth="150"
            Width="100">6</Button>
    <Button Margin="5,5,5,0" MaxWidth="100"
            Width="150">7</Button>
    <Button Margin="5,5,5,0" MaxWidth="150"
            Width="100">8</Button>
    <Button Margin="5,5,5,0" MaxWidth="200"
            MinWidth="150">9</Button>
</StackPanel>
```

The result demonstrates the creation of buttons with different combinations of the width (Fig. 34). If you specify a value for the System.Windows.FrameworkElement.Width property that is inconsistent with the values of the System.Windows.FrameworkElement.MinWidth or System.Windows.Frame-

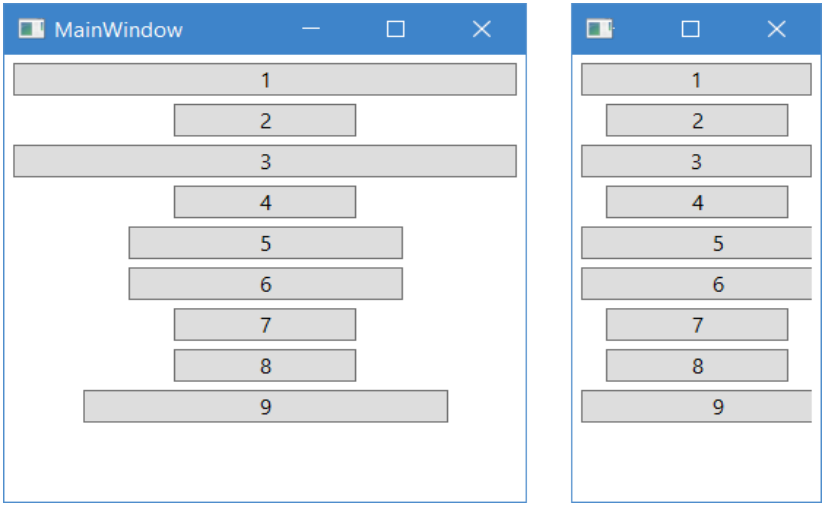


Fig. 34. Use the Width, MinWidth, and MaxWidth properties

workElement.MaxWidth properties, then it will be ignored. If you set the property correctly, but the element will not fit in the space provided to it, it will be truncated.

The following code corresponds to the markup described above (the complete example is in Wpf.Properties.Width.CSharp):

C#

```
var margin = new Thickness(5.0, 5.0, 5.0, 0.0);

var button1 = new Button {
    Content = 1,
    Margin = margin
};
var button2 = new Button {
   Content = 2,
    Margin = margin,
   Width = 100.0
};
var button3 = new Button {
    Content = 3,
    Margin = margin,
    MinWidth = 100.0
};
var button4 = new Button {
    Content = 4,
    Margin = margin,
    MaxWidth = 100.0
};
var button5 = new Button {
    Content = 5,
    Margin = margin,
    MinWidth = 100.0,
    Width = 150.0
};
var button6 = new Button {
    Content = 6,
```

```
    Margin = margin,
    MinWidth = 150.0,
    Width = 100.0
};
var button7 = new Button {
    Content = 7,
    Margin = margin,
    MaxWidth = 100.0,
    Width = 150.0
};
var button8 = new Button {
    Content = 8,
    Margin = margin,
    MaxWidth = 150.0,
    Width = 100.0
};
var button9 = new Button{
    Content = 9,
    Margin = margin,
    MaxWidth = 200.0,
    MinWidth = 150.0
};
var stackPanel = new StackPanel();
stackPanel.Children.Add(button1);
stackPanel.Children.Add(button2);
stackPanel.Children.Add(button3);
stackPanel.Children.Add(button4);
stackPanel.Children.Add(button5);
stackPanel.Children.Add(button6);
stackPanel.Children.Add(button7);
stackPanel.Children.Add(button8);
stackPanel.Children.Add(button9);
```

The properties associated with the height of the element work just like the width-related properties discussed above.

As an example, let's look at the following markup fragment (the complete example is in Wpf.Properties.Height.Xaml):

XAML

```
<StackPanel Orientation="Horizontal">
    <Button Margin="5,5,0,5">1</Button>
    <Button Height="100" Margin="5,5,0,5">2</Button>
    <Button Margin="5,5,0,5" MinHeight="100">3
            </Button>
    <Button Margin="5,5,0,5" MaxHeight="100">4
            </Button>
    <Button Height="150" Margin="5,5,0,5"
            MinHeight="100">5</Button>
    <Button Height="100" Margin="5,5,0,5"
            MinHeight="150">6</Button>
    <Button Height="150" Margin="5,5,0,5"
            MaxHeight="100">7</Button>
    <Button Height="100" Margin="5,5,0,5"
            MaxHeight="150">8</Button>
    <Button Margin="5,5,0,5" MaxHeight="200"
            MinHeight="150">9</Button>
</StackPanel>
```

The result shows the creation of buttons with different combinations of the height (Fig. 35). If you specify the value of the System.Windows.FrameworkElement.Height property that is inconsistent with the values of the System.Windows.FrameworkElement.MinHeight or System.Windows.FrameworkElement.MaxHeight properties, then it will be ignored. If you set the property correctly, but the element will not fit in the space provided to it, it will be truncated.

The following code corresponds to the markup described above (the complete example is in Wpf.Properties.Height.CSharp):

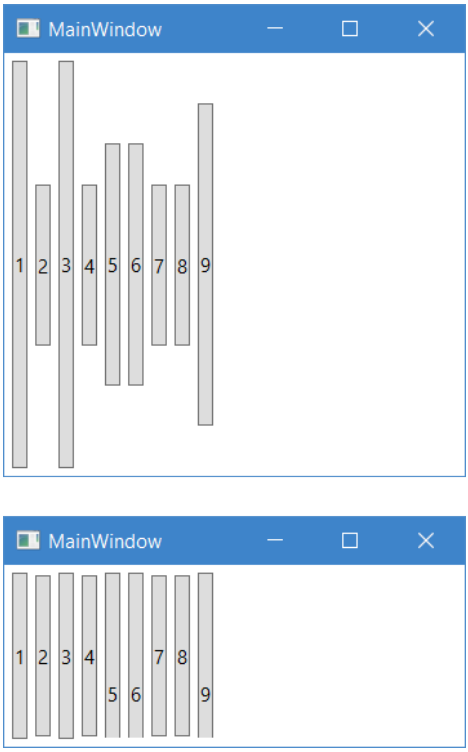


Fig. 35. Using the Height, MinHeight, and MaxHeight properties

C #

```
var margin = new Thickness(5.0, 5.0, 0.0, 5.0);
var button1 = new Button {
    Content = 1,
    Margin = margin
};
var button2 = new Button {
    Content = 2,
    Height = 100.0,
    Margin = margin
};
```

```
var button3 = new Button {
    Content = 3,
    Margin = margin,
    MinHeight = 100.0
};
var button4 = new Button {
    Content = 4,
    Margin = margin,
    MaxHeight = 100.0
};
var button5 = new Button {
    Content = 5,
    Height = 150.0,
    Margin = margin,
    MinHeight = 100.0
};
var button6 = new Button {
    Content = 6,
    Height = 100.0,
    Margin = margin,
    MinHeight = 150.0
};
var button7 = new Button {
    Content = 7,
    Height = 150.0,
    Margin = margin,
    MaxHeight = 100.0
};
var button8 = new Button {
    Content = 8,
    Height = 100.0,
    Margin = margin,
    MaxHeight = 150.0
};
var button9 = new Button {
    Content = 9,
    Margin = margin,
    MaxHeight = 200.0,
    MinHeight = 150.0
};
```

```
var stackPanel = new StackPanel { Orientation =
    Orientation.Horizontal };
stackPanel.Children.Add(button1);
stackPanel.Children.Add(button2);
stackPanel.Children.Add(button3);
stackPanel.Children.Add(button4);
stackPanel.Children.Add(button5);
stackPanel.Children.Add(button6);
stackPanel.Children.Add(button7);
stackPanel.Children.Add(button8);
stackPanel.Children.Add(button9);
```

Explicitly specifying values for the System.Windows.FrameworkElement.Width and System.Windows.FrameworkElement.Height properties is of higher priority than the requirement for the element to be stretched. In other words, if you set the System.Windows.FrameworkElement.Width property and set the System.Windows.FrameworkElement.HorizontalAlignment property to System.Windows.HorizontalAligment.Stretc, the last will be ignored. A similar situation occurs with the properties System.Windows.FrameworkElement.Height and System.Windows.FrameworkElement.VerticalAlignment.

In certain situations, you may need to know the actual dimensions of the element. In this case, do not use the System.Windows.FrameworkElement.Width and System.Windows.FrameworkElement.Height properties, because they reflect the desired dimensions, not the actual ones. It is worth using the properties System.Windows.FrameworkElement.ActualWidth and System.Windows.FrameworkElement.ActualHeight, which store the dimensions used to render the element.

13. Buttons

One of the most common controls is the button. In WPF, there are several kinds of buttons that will be discussed in this section.

- System.Windows.Controls.Button. A usual button.
- System.Windows.Controls.CheckBox. A checkbox that allows you to control two states: on and off.
- System.Windows.Controls.RadioButton. A radio button that allows you to select one option from the provided set (group).
- System.Windows.Controls.Primitives.RepeatButton. A button that constantly generates a click event, from the moment it is clicked and until it is released.

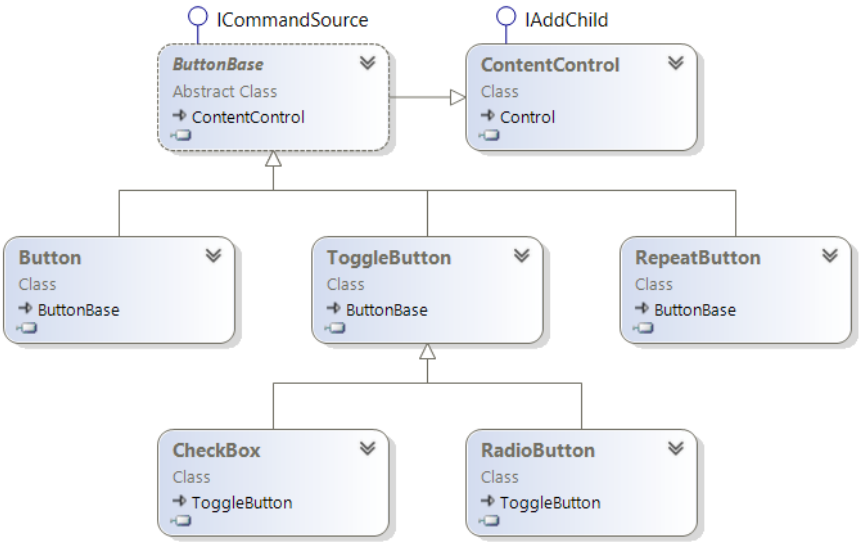


Fig. 36. Diagram of classes that describe buttons

Despite the fact that these are different controls described by separate classes, they all have the same basic functionality described in the base classes. Fig. 36 shows a diagram of the classes that describe the buttons.

In the diagram above, there are several base classes from which the control elements discussed above are inherited:

- System.Windows.Controls.Primitives.ButtonBase. This class is the base for all buttons. Here you can find all the common functionality inherent in buttons of any kind.
- System.Windows.Controls.Primitives.ToggleButton. This class is the base for buttons that support two states: on and off.

Properties of the System.Windows.Controls.Primitives.ButtonsBase class:

- **ClickMode** (System.Windows.Controls.ClickMode). Gets or sets the trigger mode for the button click event. The default value is System.Windows.Controls.ClickMode.Release.
- **IsPressed** (System.Boolean). Gets a value that indicates whether the button is pressed. true if the button is pressed; otherwise, false.

Events of the System.Windows.Controls.Primitives.ButtonBase class:

- **Click** (System.Windows.RoutedEventHandler). It trigs when the button is pressed.

In order to specify the click mode, you must use the System.Windows.Controls.ClickMode enumeration, which contains the following options:

- Hover. The click event should work when the mouse hovers over the button.

- Press. The click event should work when the button is clicked.
- Release. The click event should work when the button is released.

Properties of the System.Windows.Controls.Primitives.ToggleButton class:

- **IsChecked** (System.Nullable<System.Boolean>). Gets or sets a value that indicates whether the button is on or off. true if the button is on; false if the button is off; otherwise, null. The default value is false.
- **IsThreeState** (System.Windows.Boolean). Gets or sets a value that indicates whether the button supports three switching states. true if the button supports three states; otherwise, false. The default value is false.

Events of the System.Windows.Controls.Primitives.ToggleButton class:

- **Checked** (System.Windows.RoutedEventHandler). It works when the button is turned on.
- **Indeterminate** (System.Windows.RoutedEventHandler). It works when the button is transferred to the third state (not on and not off).
- **Unchecked** (System.Windows.RoutedEventHandler). It works when the button is turned off.

13.1. The Button Control

The System.Windows.Controls.Button control is the most common kind of button used in designing the user interface. The main task of this control is to handle the click and generate a corresponding event, with which, as a rule, a certain software action is associated.

The following markup fragment demonstrates the control (the full example is in the Wpf.Controls.Button.Xaml folder):

XAML

```
<Button Click="Button_Click" Height="23"
        Width="75">Button</Button>
```

The result of the above markup is shown in Fig. 37.

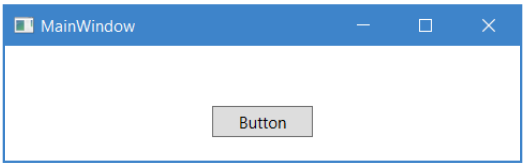


Fig. 37. The Button control

This example shows how to subscribe to the button event System.Windows.Controls.ButtonBase.Click, which is responsible for clicking it. The name of the attribute should be the name of the event, and the value — the name of the event handler method. This means that in the file with the code that corresponds to the current code-behind markup file there must be a method with this name and signature suitable for the delegate of the event:

C#

```
private void Button_Click(object sender,
        RoutedEventArgs e)
{
    Title = $"Button clicked at {DateTime.Now}";
}
```

The following code corresponds to the markup described above (the complete example is in Wpf.Controls.Button.CSharp):

C#

```
var button = new Button {
    Content = "Button",
    Height = 23.0,
    Width = 75.0 };
button.Click += Button_Click;
```

13.2. The RepeatButton Control

The System.Windows.Controls.Primitives.RepeatButton control differs from the usual button only in that it generates a click event after pressing the button and before it is released.

Properties of the System.Windows.Controls.Primitives.RepeatButton class:

- **Delay** (System.Int32). Gets or sets the number of milliseconds that must elapse before starting to generate click events, after the button has been pressed. The default value is System.Windows.SystemParameters.KeyboardDelay.
- **Interval** (System.Int32). Gets or sets the number of milliseconds that must elapse before the second click event. The default value is System.Windows.SystemParameters.KeyboardSpeed.

The following markup fragment demonstrates the control (the complete example is in the Wpf.Controls.RepeatButton.Xaml folder):

XAML

```
<RepeatButton Click="RepeatButton_Click"
              Height="23" Width="100"> Repeat Button
</RepeatButton>
```

The following code corresponds to the markup discussed above (the complete example is in Wpf.Controls.RepeatButton.CSharp):

C#

```
var repeatButton = new RepeatButton
{
    Content = "Repeat Button",
    Height = 23.0,
    Width = 100.0
};
repeatButton.Click += RepeatButton_Click;
```

13.3. The CheckBox Control

The System.Windows.Controls.CheckBox control allows the user to operate with a certain parameter with two states: on and off. If necessary, it is possible to create an element that will support not two but three states. In addition to the fact that an element generates an event every time it is clicked, it also generates more detailed events related to each state (on, off).

The following markup fragment demonstrates the control (the complete example is in the Wpf.Controls.CheckBox.Xaml folder):

XAML

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
```

```
    <CheckBox Checked="CheckBox_Checked"
              Grid.Row="0"
              HorizontalAlignment="Center"
              Unchecked="CheckBox_Unchecked"
              VerticalAlignment="Center">
        Unchecked
    </CheckBox>

    <CheckBox Checked="CheckBox_Checked"
              Grid.Row="1"
              HorizontalAlignment="Center"
              IsChecked="True"
              Unchecked="CheckBox_Unchecked"
              VerticalAlignment="Center">
        Checked
    </CheckBox>

    <CheckBox Checked="CheckBox_Checked"
              Grid.Row="2"
              HorizontalAlignment="Center"
              Indeterminate="CheckBox_Indeterminate"
              IsChecked="{x:Null}"
              IsThreeState="True"
              Unchecked="CheckBox_Unchecked"
              VerticalAlignment="Center">
        Indeterminate
    </CheckBox>
</Grid>
```

The result of the above markup is shown in Fig. 38.

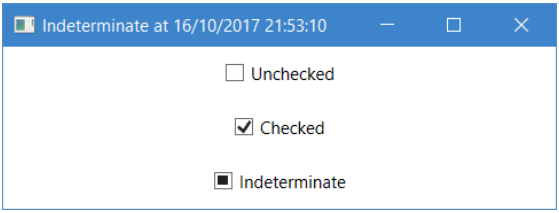


Fig. 38. The CheckBox control

In this example, the markup extension is used to assign null to one of the properties: IsThreeState="{x:Null}".

The following code corresponds to the markup described above (the complete example is in Wpf.Controls.CheckBox.CSharp):

C#

```
var checkBox1 = new CheckBox
{
    Content = "Unchecked",
    HorizontalAlignment = HorizontalAlignment.Center,
    VerticalAlignment = VerticalAlignment.Center
};
checkBox1.Checked += CheckBox_Checked;
checkBox1.Unchecked += CheckBox_Unchecked;

var checkBox2 = new CheckBox
{
    Content = "Checked",
    HorizontalAlignment = HorizontalAlignment.Center,
    IsChecked = true,
    VerticalAlignment = VerticalAlignment.Center
};
checkBox2.Checked += CheckBox_Checked;
checkBox2.Unchecked += CheckBox_Unchecked;

var checkBox3 = new CheckBox
{
    Content = "Indeterminate",
    HorizontalAlignment = HorizontalAlignment.Center,
    IsChecked = null,
    IsThreeState = true,
    VerticalAlignment = VerticalAlignment.Center
};
checkBox3.Checked += CheckBox_Checked;
checkBox3.Indeterminate += CheckBox_Indeterminate;
```

```
checkBox3.Unchecked += CheckBox_Unchecked;

var grid = new Grid();
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());

grid.Children.Add(checkBox1);
grid.Children.Add(checkBox2);
grid.Children.Add(checkBox3);

Grid.SetRow(checkBox1, 0);
Grid.SetRow(checkBox2, 1);
Grid.SetRow(checkBox3, 2);
```

13.4. The RadioButton Control

The System.Windows.Controls.RadioButton control allows the user to select one option (item) from an available set (group). When one item is activated from a group, all other elements of this group are automatically deactivated, providing the functionality of selecting only one of the elements. At the same time, there may be several unrelated groups of radio buttons within the interface.

Properties of the System.Windows.Controls.RadioButton class:

- **GroupName** (System.String). Gets or sets the name of the group which the switch belongs to. The default value is System.String.Empty.

The following markup fragment demonstrates the control (the complete example is in the Wpf.Controls.RadioButton.Xaml folder):

XAML

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <StackPanel Grid.Column="0"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
        <RadioButton Click="RadioButton_Click"
            Margin="5">Option 1</RadioButton>
        <RadioButton Click="RadioButton_Click"
            Margin="5">Option 2</RadioButton>
        <RadioButton Click="RadioButton_Click"
            Margin="5">Option 3</RadioButton>
    </StackPanel>
    <StackPanel Grid.Column="1"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
        <RadioButton Click="RadioButton_Click"
            Margin="5">Option 4</RadioButton>
        <RadioButton Click="RadioButton_Click"
            Margin="5">Option 5</RadioButton>
        <RadioButton Click="RadioButton_Click"
            Margin="5">Option 6</RadioButton>
    </StackPanel>
</Grid>
```

The result of the above markup is shown in Fig. 39.

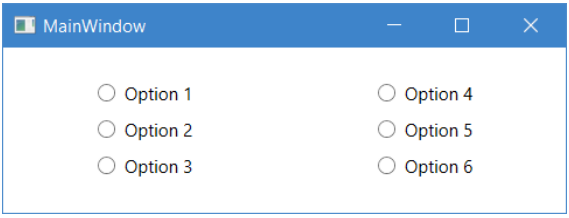


Fig. 39. The RadioButton control

The following code corresponds to the markup described above (the complete example is in Wpf.Controls.RadioButton.CSharp):

C#

```
var margin = new Thickness(5.0);

var radioButton1 = new RadioButton {
    Content = "Option 1",
    Margin = margin
};
radioButton1.Click += RadioButton_Click;
var radioButton2 = new RadioButton {
    Content = "Option 2",
    Margin = margin
};
radioButton2.Click += RadioButton_Click;
var radioButton3 = new RadioButton {
    Content = "Option 3",
    Margin = margin
};
radioButton3.Click += RadioButton_Click;
var radioButton4 = new RadioButton {
    Content = "Option 4",
    Margin = margin
};
radioButton4.Click += RadioButton_Click;
var radioButton5 = new RadioButton {
    Content = "Option 5",
    Margin = margin
};
radioButton5.Click += RadioButton_Click;
var radioButton6 = new RadioButton {
    Content = "Option 6",
    Margin = margin
};
radioButton6.Click += RadioButton_Click;
```

```
var stackPanel1 = new StackPanel
{
    HorizontalAlignment = HorizontalAlignment.Center,
    VerticalAlignment = VerticalAlignment.Center
};
stackPanel1.Children.Add(radioButton1);
stackPanel1.Children.Add(radioButton2);
stackPanel1.Children.Add(radioButton3);
var stackPanel2 = new StackPanel
{
    HorizontalAlignment = HorizontalAlignment.Center,
    VerticalAlignment = VerticalAlignment.Center
};
stackPanel2.Children.Add(radioButton4);
stackPanel2.Children.Add(radioButton5);
stackPanel2.Children.Add(radioButton6);
var grid = new Grid();
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.Children.Add(stackPanel1);
grid.Children.Add(stackPanel2);
Grid.SetColumn(stackPanel1, 0);
Grid.SetColumn(stackPanel2, 1);
```

You can group radio buttons in two ways:

- All radio buttons within a single panel or grouping control are automatically placed in the same group.
- All radio buttons with the same value of the System.Windows.Controls.RadioButton.GroupName property are automatically placed in the same group. This grouping option does not take into account the location of the element in the element tree. In other words, using this method, you can create more than one group within one panel or create one group whose elements will be in different panels.

14. Text Boxes

The most common control for getting data from a user is the text box. In WPF, there are several kinds of such fields that will be discussed in this section.

- System.Windows.Controls.TextBox. A common field for text input.
- System.Windows.Controls.RichTextBox. A text input field that supports a variety of text formatting: alignment, font, color, etc. This element will be considered in the section related to flow documents, as it applies them to display its contents.
- System.Windows.Controls.PasswordBox. Text field for entering and processing passwords.

Different text boxes, like buttons, have a part of the general functionality that is rendered in the base classes. Fig. 40 shows a diagram of classes describing text boxes.

The diagram above shows the base class System.Windows.Controls.Primitives.TextBoxBase, from which the System.Windows.Controls.TextBox and System.Windows.Controls.RichTextBox classes are inherited.

Properties of the System.Windows.Primitives.TextBoxBase class:

- **AcceptsReturn** (System.Boolean). Gets or sets a value that indicates how the element responds to the Enter key. true if a line break is to be insert at the current position of the input cursor when you press Enter; otherwise, pressing Enter is ignored. The default value is false for the System.

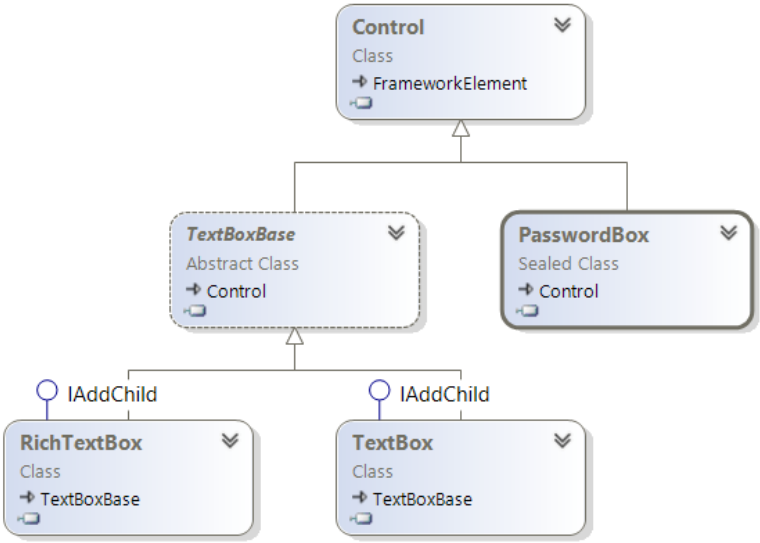


Fig. 40. Diagram of classes describing text boxes

Windows.Controls.TextBox class and true for the System.Windows.Controls.RichTextBox class.

- **AcceptsTab** (System.Boolean). Gets or sets a value that indicates how the item responds to the Tab key. true if a horizontal tab character is to be inserted at the current position of the input cursor when you press the Tab key; otherwise, you need to move the focus to the next element and do not insert a horizontal tab character. The default value is false.
- **AutoWordSelection** (System.Boolean). Gets or sets a value that specifies whether to select the entire word when the user selects part of the word and moves the mouse cursor along it. true if automatic word selection is enabled; otherwise, false. The default value is false.

- **CanRedo** (System.Boolean). Gets a value that indicates whether the last undone action can be repeated. true if possible; otherwise, false.
- **CanUndo** (System.Boolean). Gets a value that indicates whether you can undo the last action. true if possible; otherwise, false.
- **CaretBrush** (System.Windows.Media.Brush). Gets or sets the brush that is used to display the insertion point.
- **HorizontalOffset** (System.Double). Gets the value of the horizontal scroll position in hardware-independent units.
- **HorizontalScrollBarVisibility** (System.Windows.Controls.ScrollBarVisibility). Gets or sets the horizontal scroll bar display mode. The default value is System.Windows.Controls.SrollBarVisibility.Hidden.
- **IsInactiveSelectionHighlightEnabled** (System.Boolean). Gets or sets a value that indicates whether to display text selection when the item does not have a focus. true if it is needed to display the selection; otherwise, false. The default value is false.
- **IsReadOnly** (System.Boolean). Gets or sets a value that indicates whether the user can edit the text in the element. true if it cannot; otherwise, false. The default value is false.
- **IsReadOnlyCaretVisible** (System.Boolean). Gets or sets a value that indicates whether to display the insertion point in the element in which the user cannot edit a text. true if the input cursor needs to be displayed; otherwise, false. The default value is false.

- **IsSelectionActive** (System.Boolean). Gets a value that indicates whether the element has a focus and whether there is a selected text in it. true if there is focus and selection; otherwise, false.
- **IsUndoEnabled** (System.Boolean). Gets or sets a value that indicates whether it is required to support undoing the last action in the element. true if the last action undo support is required; otherwise, false. The default value is true.
- **SelectionBrush** (System.Windows.Media.Brush). Gets or sets the brush that is used to display the selected text.
- **SelectionOpacity** (System.Double). Gets or sets the transparency level of the brush used to display the selected text. This property can take values from 0.0 (transparent) to 1.0 (opaque). The default value is 0.4.
- **SpellCheck** (System.Windows.Controls.SpellCheck). Gets an object that provides access to spelling errors made in the text of the element.
- **UndoLimit** (System.Int32). Gets or sets the maximum number of actions that are in the action queue and are available for cancellation. The default value is -1, which means an unlimited queue size.
- **VerticalOffset** (System.Double). Gets the value of the vertical scroll position in hardware-independent units.
- **VerticalScrollBarVisibility** (System.Windows.Controls.ScrollBarVisibility). Gets or sets the vertical scroll bar display mode. The default value is System.Windows.Controls.SrollBarVisibility.Hidden.

Events of the System.Windows.Controls.Primitives.TextBoxBase class:

- **SelectionChanged** (System.Windows.RoutedEventHandler). It works when the selection of the text changes.
- **TextChanged** (System.Windows.Controls.TextChangedEventHandler). It works when the text changes.

Methods of the System.Windows.Controls.Primitives.TextBoxBase class:

- **AppendText**(textData). Adds the transmitted string to the end of the text.
- **Copy**(). Copies the current selection of text to the clipboard.
- **Cut**(). Copies the current selection of text to the clipboard and removes it from the text.
- **LineDown**(). Scrolls the content of the element one line down.
- **LineLeft**(). Scrolls the contents of the element one line to the left.
- **LineRight**(). Scrolls the contents of the element one line to the right.
- **LineUp**(). Scrolls the contents of the element one line up.
- **LockCurrentUndoUnit**(). Ends the most recent undo block. This prevents the completed block from merging with subsequent action blocks that will be added subsequently.
- **PageDown**(). Scrolls the content of the element one page down.
- **PageLeft**(). Scrolls the content of the element one page to the left.
- **PageRight**(). Scrolls the content of the element one page to the right.

- **PageUp**(). Scrolls the content of the element one page up.
- **Paste**(). Inserts content from the clipboard instead of the current selection of text.
- **Redo**(). Repeats the last undone action. Returns true if the operation succeeded; otherwise — false (no command to repeat).
- **ScrollToEnd**(). Scrolls the content of the element to the end.
- **ScrollToHome**(). Scrolls the content of the element to the beginning.
- **ScrollToHorizontalOffset**(offset). Scrolls the contents of the element to the transmitted value of the horizontal scroll.
- **ScrollToVerticalOffset**(offset). Scrolls the contents of the element to the transferred value of the vertical scroll.
- **SelectAll**(). Selects all text.
- **Undo**(). Undoes the last action. Returns true if the operation succeeded; otherwise — false (no command to undo).

To specify the display mode for a horizontal or vertical scrollbar, you must use the System.Windows.Controls.ScrollBarVisibility enumeration, which contains the following options:

- Auto. The scroll bar appears automatically if the content does not fit in the client area of the element.
- Disabled. The scroll bar does not appear, even if the content does not fit in the client area of the element. In this case, the content uses the limited size of the free area.
- Hidden. The scroll bar does not appear, even if the content does not fit in the client area of the element. In this case, the content behaves the same as with the scroll bar.
- Visible. The scrollbar is always visible.

14.1. The TextBox Control

The System.Windows.Controls.TextBox control is the main kind of text input field. It supports single-line and multi-line text input/output.

Properties of the System.Windows.Controls.TextBox class:

- **CaretIndex** (System.Int32). Gets or sets the input cursor position. This property can take values from 0 and greater.
- **CharacterCasing** (System.Windows.Controls.CharacterCasing). Gets or sets the mode for changing the case of user-entered characters. The default value is System.Windows.Controls.CharacterCasing.Normal.
- **LineCount** (System.Int32). Gets the number of lines in the text.
- **MaxLength** (System.Int32). Gets or sets the maximum number of characters entered. The default value is 0, which means an unlimited number.
- **MaxLines** (System.Int32). Gets or sets the maximum number of visible lines. The default value is System.Int32.MaxValue.
- **MinLines** (System.Int32). Gets or sets the minimum number of visible lines. The default value is 1.
- **SelectedText** (System.String). Gets or sets the contents of the current text selection.
- **SelectionLength** (System.Int32). Gets or sets the number of characters of the current text selection. The default value is 0.
- **SelectionStart** (System.Int32). Gets or sets the index of the first character of the current text selection.
- **Text** (System.String). Gets or sets the text of the element.

The default value is System.String.Empty.

- **TextAlignment** (System.Windows.TextAlignment). Gets or sets the horizontal alignment of the text. The default value is System.Windows.TextAlignment.Left.
- **TextWrapping** (System.Windows.TextWrapping). Gets or sets the text wrap mode. The default value is System.Windows.TextWrapping.NoWrap.
- Methods of class System.Windows.Controls.TextBox:
- **Clear**(). Clears the contents (text) of the element.
- **GetCharacterIndexFromIndex**(lineIndex). Returns the index of the first character in the lne with the specified index.
- **GetFirstVisibleLineIndex**(). Returns the index of the first visible line.
- **GetLastVisibleLineInde**(). Returns the index of the last visible line.
- **GetLineIndexFromCharacterIndex**(charIndex). Returns the index of the line that contains the character with the specified index.
- **GetLineLength**(lineIndex). Returns the number of characters in a line with the specified index.
- **GetLineText**(lineIndex). Returns the contents of a line with the specified index.
- **GetNextSpellingErrorCharacterIndex**(charIndex, direction). Returns the index of the first character of the next spelling error. In addition, the index of the character from which you want to start the search and the direction of the search is indicated.
- **GetSpellingError**(charIndex). Returns an object that describes the spelling error at the specified index.

- **GetSpellingErrorLength**(charIndex). Returns the number of characters of the spelling error that include the specified index.
- **GetSpellingErrorStart**(charIndex). Returns the index of the first character of the spelling error that includes the specified index.
- **ScrollToLine**(lineIndex). Scrolls the contents of the element to the line with the specified index.
- **Select**(start, length). Selects text in the specified range.

In order to specify the mode of changing the case of the characters entered by the user, you must use the System.Windows.Controls.CharacterCasing enumeration, which contains the following options:

- Lower. The entered characters are converted to lowercase.
- Normal. The character case does not change when you type.
- Upper. The entered characters are converted to uppercase.

In order to specify the horizontal alignment of the text, you must use the System.Windows.Controls.TextAlignment enum, which contains the following options:

- Center. Centering the text.
- Justify. Stretching the text.
- Left. Align text to the left.
- Right. Align text to the right.

In order to specify the text wrapping mode, you must use the System.Windows.TextWrapping enumeration, which contains the following options:

- NoWrap. The wrapping is disabled.

- Wrap. The words are moved to the next line if they do not fit. If the transfer algorithm cannot determine the best transfer option (for example, a very long word), the transfer is still carried out.
- WrapWithOverflow. The words are moved to the next line if they do not fit. If the transfer algorithm cannot determine the best transfer option (for example, a very long word), the transfer is not performed, and the word can go beyond the borders of the element.

The following markup fragment demonstrates the control (the complete example is in the Wpf.Controls.TextBox.Xaml folder):

XAML

```
<TextBox HorizontalAlignment="Center"
         Text="Text" VerticalAlignment="Center"
         Width="200"/>
```

The result of the above markup is shown in Fig. 41.

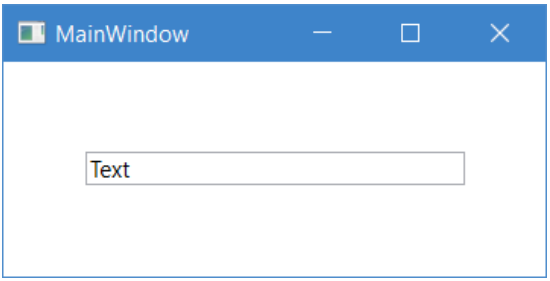


Fig. 41. The TextBox control

The following code corresponds to the markup described above (the complete example is in Wpf.Controls.TextBox.CSharp):

C#

```
var textBox = new TextBox
{
    HorizontalAlignment = HorizontalAlignment.Center,
    Text = "Text",
    VerticalAlignment = VerticalAlignment.Center,
    Width = 200.0
};
```

14.2. The PasswordBox Control

The System.Windows.Controls.PasswordBox control is for entering passwords.

Properties of the System.Windows.Controls.PasswordBox class:

- **CaretBrush** (System.Windows.Media.Brush). Gets or sets the brush that is used to display the insertion point.
- **IsInactiveSelectionHighlightEnabled** (System.Boolean). Gets or sets a value that indicates whether to display text selection when the element does not have a focus. true if it is needed to display the selection; otherwise, false. The default value is false.
- **IsSelectionActive** (System.Boolean). Gets a value that indicates whether the element has a focus and whether there is a selected text in it. true if there is focus and selection; otherwise, false.
- **MaxLength** (System.Int32). Gets or sets the maximum number of characters entered. The default value is 0, which means an unlimited number.
- **Password** (System.String). Gets or sets the password. The default value is System.String.Empty.

- **PasswordChar** (System.Char). Gets or sets a character that uses for substitution and display of password characters. The default value is the bullet character (•).
- **SecurePassword** (System.Security.SecureString). Gets the password as a secure string.
- **SelectionBrush** (System.Windows.Media.Brush). Gets or sets the brush that is used to display the selected text.
- **SelectionOpacity** (System.Double). Gets or sets the transparency level of the brush used to display the selected text. This property can take values from 0.0 (transparent) to 1.0 (opaque). The default value is 0.4.
- Events of the System.Windows.Controls.PasswordBox class:
- **PasswordChanged** (System.Windows.RoutedEventHandler). It works when the password is changed.

Methods for the System.Windows.Controls.PasswordBox class:

- **Clear**(). Clears the contents (text) of the element.
- **Paste**(). Inserts content from the clipboard instead of the current selection of text.
- **SelectAll**(). Sellects all text.

The following markup fragment demonstrates the control (the complete example is in the Wpf.Controls.PasswordBox.Xaml folder):

XAML

```
<PasswordBox HorizontalAlignment="Center"
             Password="Text"
             VerticalAlignment="Center"
             Width="200"/>
```

The result of the above markup is shown in Fig. 42.

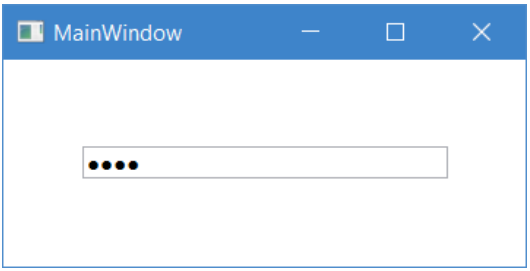


Fig. 42. The PasswordBox control

The following code corresponds to the markup described above (the complete example is in Wpf.Controls.PasswordBox.CSharp):

C#

```
var passwordBox = new PasswordBox
{
    HorizontalAlignment = HorizontalAlignment.Center,
    Password = "Text",
    VerticalAlignment = VerticalAlignment.Center,
    Width = 200.0
};
```

15. Home Assignment

15.1. Task 1

It is necessary to develop an application containing a set of buttons that occupy 2/3 of the width of the window for any size (Fig. 43). Each button should display the color name as content and have an outer indent equal to 2.0. Also, the corresponding color should be used as the foreground color of the button. You must use the following color set: Navy, Blue, Aqua, Teal, Olive, Green, Lime, Yellow, Orange, Red, Maroon, Fuchsia, Purple, Black, Silver, Gray, White.

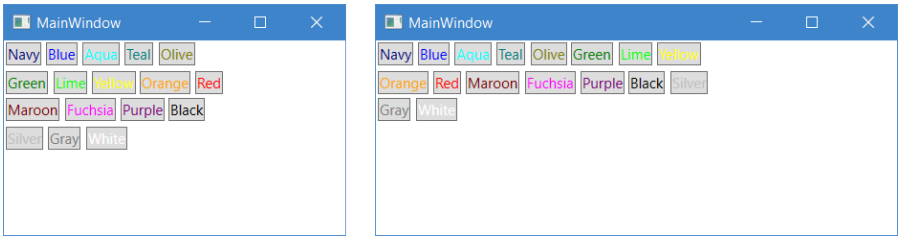


Fig. 43. Task 1

15.2. Task 2

You need to develop an application "Calculator" (Fig. 44). At the top of the application, you need to use two text boxes. The first is used to display the previous operations, and the second is used to enter the current number. The contents of both boxes should be prohibited from editing via keyboard input. These boxes will be filled automatically by clicking on the corresponding buttons below.

The buttons "0" — "9" add the corresponding digit to the end of the current number. Thus checks should be carried out, not allowing an incorrect input. For example, you cannot enter numbers starting with zero, after which there is no decimal point.

The "." button adds (if possible) a decimal point to the current number.

The buttons "/", "*", "+", "-" perform the corresponding operation on the result of the previous operation and the current number.

The "=" button calculates the expression and displays the result.

The "CE" button clears the current number.

The "C" button clears the current number and the previous expression.

The "<" button clears the last entered character in the current number.

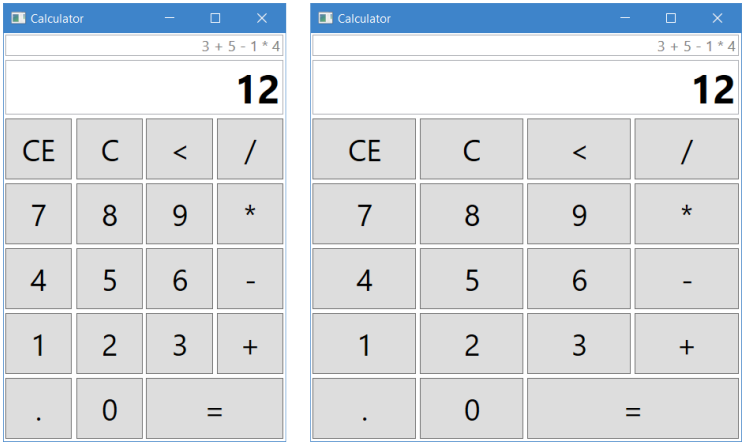


Fig. 44. Task 2

Lesson 1

Introduction to WPF, containers, introduction to controls

© Pavel Dubskiy.
© STEP IT Academy.
www.itstep.org

All rights to protected pictures, audio, and video belong to their authors or legal owners.
Fragments of works are used exclusively in illustration purposes to the extent justified by the purpose as part of an educational process and for educational purposes in accordance with Article 1273 Sec. 4 of the Civil Code of the Russian Federation and Articles 21 and 23 of the Law of Ukraine "On Copyright and Related Rights". The extent and method of cited works are in conformity with the standards, do not conflict with a normal exploitation of the work, and do not prejudice the legitimate interests of the authors and rightholders. Cited fragments of works can be replaced with alternative, non-protected analogs, and as such correspond the criteria of fair use.

All rights reserved. Any reproduction, in whole or in part, is prohibited. Agreement of the use of works and their fragments is carried out with the authors and other right owners. Materials from this document can be used only with resource link.

Liability for unauthorized copying and commercial use of materials is defined according to the current legislation of Ukraine.