

**Рязанский станкостроительный колледж РГРТУ**

**МДК.01.01 Разработка программных модулей**

**Тема 3. Разработка интерфейса пользователя**

**Рязань 2023**

## Оглавление

Событийно-управляемое программирование. Обработчики событий.....	4
Общие понятия.....	4
События визуальных классов.....	4
Правила разработки интерфейсов пользователя. Элементы управления .....	6
Принципы построения интерфейсов.....	6
Золотое сечение.....	6
Кошелек Миллера.....	6
Принцип группировки .....	7
Бритва Оккама или KISS .....	7
Видимость отражает полезность .....	7
Умное заимствование .....	7
Создание профессионального интерфейса .....	7
Стандартные элементы интерфейса .....	7
Небольшая палитра инструментов .....	8
Одинаковое расстояние между элементами управления .....	8
TabIndex. "Правильный" порядок .....	8
Выбор шрифтов.....	8
Выбор цветов.....	9
Альтернативное управление .....	9
Кирпичики интерфейса .....	10
Заголовок окна (окна) .....	10
Командные кнопки.....	11
Меню .....	11
Списки.....	12
Флажки и переключатели .....	12
Панели инструментов .....	12
Вкладки .....	13
Всплывающие подсказки .....	13
Элементы интерфейса .....	14
Групповые панели .....	14
Компонент Border – панель.....	14
Компонент GroupBox – групповая панель.....	15
Меню.....	15
Компонент Menu – главное меню .....	15
Компонент MenuItem – элемент меню .....	16
Контекстное меню.....	17
Компонент ContextMenu – контекстное меню .....	17
Панели инструментов.....	18
Компонент ToolBar – панель инструментов .....	18
Строка статуса.....	19
Компонент StatusBar – строка статуса.....	19
Размещение элементов в контейнере Grid.....	20
Вкладки.....	21
Компонент TabControl – многостраничная панель .....	21
Всплывающие подсказки.....	22
Разработка масштабируемого интерфейса приложения .....	24
Основы компоновки WPF .....	24
Контейнер Grid .....	25
Установка размеров .....	27

Контейнер UniformGrid .....	28
Контейнер StackPanel.....	28
Контейнер DockPanel.....	28
Компонет GridSplitter.....	28
Свойства компоновки элементов.....	29
Примеры компоновки элементов.....	30
Компоновка с применением контейнера Grid.....	30
Смешанная компоновка с применением контейнеров Grid и StackPanel.....	31
Разработка масштабируемого интерфейса приложения.....	32
Рекомендации при разработке масштабируемого интерфейса приложения.....	32
Практическая работа №12 .....	35
Практическая работа №13 .....	38
Многооконные приложения. Диалоговые окна.....	40
Общие положения.....	40
Модальные окна на базе класса MessageBox.....	40
Произвольное окно Window .....	42
Передача параметров между окнами .....	43
<i>Используем отдельный класс .....</i>	43
Концепция ресурсов в MAUI.....	45
Уровни ресурсов .....	45
Ресурсы приложения .....	46
Динамические ресурсы .....	46
Стили.....	47
TargetType .....	47
Переопределение стилей .....	47
Встроенные стили .....	Ошибка! Закладка не определена.
Визуальные состояния элемента .....	Ошибка! Закладка не определена.
Доступные визуальные состояния .....	Ошибка! Закладка не определена.
Триггеры.....	48
Триггеры свойств .....	48
Триггеры событий .....	Ошибка! Закладка не определена.
Практическая работа №14 .....	50
Контроль ввода данных. Исключения.....	51
Введение .....	51
Ограничение ввода .....	51
Контроль введенных параметров .....	52
Исключения.....	55
<i>Как возникают исключительные ситуации?.....</i>	55
<i>Обработка исключений.....</i>	56
Практическая работа №15 .....	59

## Событийно-управляемое программирование. Обработчики событий

### Общие понятия

В событийном программировании вводят такое понятие, как событие. Чаще всего события вызываются пользователем, но могут быть вызваны также приложением, другой программой и т.д. У объекта в событийном программировании наряду со свойствами и методами появляется новая характеристика – набор событий, на которые он может реагировать. Этот набор для данного объекта неизменяем, однако то, каким именно образом будет происходить эта реакция, зависит от программиста. Именно он должен написать процедуру-ответ на событие. Такая процедура называется обработчиком события.

Приведем пример. При нажатии на какую-то кнопку **Button** окна мы вызываем для нее событие **Click**. Для того чтобы описать ответ на это событие, программист должен создать процедуру обработки события (обработчик). Эта процедура будет выполняться каждый раз при нажатии на кнопку.

Вы можете использовать интегрированную среду разработки (IDE) Visual C#, чтобы просмотреть события, которые поддерживаются элементом управления, и выбрать те из них, которые необходимо обрабатывать. IDE позволяет автоматически добавлять пустой метод обработчика событий и код для подписки на событие.

Создание события в интегрированной среде разработки Visual Studio

1. Выберите элемент управления, для которого требуется создать обработчик событий, и перейдите в окно **Свойства**.
2. Вверху окна **Свойства** щелкните значок **События**.
3. Дважды щелкните событие, которое требуется создать, например событие **Loaded**. Visual C# создаст пустой метод обработчика событий и добавит его в код. Например, приведенные ниже строки кода объявляют метод обработчика событий, который будет выполнен при вызове классом **Window** события **Loaded**.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    //Добавьте тут код для реализации события
}
```

В разметке интерфейса XAML в соответствующем элементе так же добавится соответствующая запись, пример смотри ниже:

```
Title="MainWindow" Height="450" Width="800" Loaded="Window_Loaded">
```

Реализации механизма событий в языке программирования C# основана на механизме делегатов.

### События визуальных классов

Рассмотрим некоторые базовые события визуальных классов.

Таблица 1. События окна

События	Комментарий
Activated	Событие наступает, когда окно становится активной, т.е. получает фокус, например, при щелчке на ней.
Deactivate	Событие наступает, когда окно перестает быть активной, т.е. теряет фокус.
Initialized	Событие наступает, когда происходит инициализации окна, когда у него устанавливаются все свойства, но до применения к нему стилей и привязки данных. Это общее

	событие для всех элементов управления в WPF, поэтому следует учитывать, что сначала возникают события вложенных элементов, а затем их контейнеров.
Closed	Событие наступает, при закрытии окна. После события <b>Closing</b> .
Closing	Первое из последовательности событий, наступающих при закрывании окна. Передаваемый по параметр <b>CancelEventArgs e</b> определяет, разрешается ли закрытие окна. По умолчанию закрытие окна разрешено. Но если закрывать окно не надо, параметру <b>e.Cancel</b> должно быть присвоено значение <b>true</b> ( <b>e.Cancel=true</b> ). Тогда последующих событий, связанных с закрытием окна, не будет.
Loaded	Возникает после полной инициализации окна и применения к нему стилей и привязки данных. После генерации этого события происходит визуализация элемента, и окно отображается на экране и становится видимым для пользователя

Таблица 2. События характерные для всех визуальных классов

События	Комментарий
Click	Соответствует щелчку мыши на компоненте или клавиши <b>Enter</b> , если компонент в фокусе. Параметр <b>Sender</b> содержит ссылку на объект, в котором произошло событие, и может использоваться для дифференцированной реакции на события в разных компонентах.
MouseDoubleClick	Соответствует двойному щелчку мыши на компоненте.
MouseDown	Происходит при нажатии кнопки мыши, если указатель мыши находится на элементе управления. Также можно узнать какая кнопка мыши нажата.
MouseEnter	Происходит, когда указатель мыши входит в пределы элемента управления.
MouseUp	Происходит при отпускании кнопки мыши, когда указатель мыши находится на элементе управления.

Таблица 3. События характерные для оконных классов

События	Комментарий
TextChanged	Данное событие возникает в том случае, если свойство <b>Text</b> изменено программой или в результате действий пользователя.
GotFocus	Событие наступает в момент получения элементом фокуса.
LostFocus	Происходит, когда фокус ввода покидает элемент управления.
KeyDown	Происходит при нажатии клавиши, если элемент управления имеет фокус. Параметр <b>KeyEventArgs</b> позволяет узнать нажатую клавишу и содержит свойства: <b>Key Key</b> - Получает код клавиши для события <b>KeyDown</b> или события <b>KeyUp</b> . Например, при нажатии клавиш F5 или Z вывести сообщение <pre>private void TextBox_KeyDown(object sender, KeyEventArgs e) {     if (e.Key == Key.F5    e.Key == Key.Z) MessageBox.Show("Нажата</pre>

События	Комментарий
	клавиша"); }
KeyUp	Событие наступает при отпускании пользователем нажатой ранее клавиши. В обработчике можно распознать отпускаемую клавишу. Имеет те же параметры, что и событие <b>KeyDown</b> .

## Правила разработки интерфейсов пользователя. Элементы управления

### Принципы построения интерфейсов

#### Золотое сечение

Золотое сечение - это самая комфортная для глаза пропорция, окно, в основе построения которой лежит сочетание симметрии и золотого сечения, способствует наилучшему зрительному восприятию и появлению ощущения красоты и гармонии.

#### Отношение отрезков а и в составляет 1,618.

Золотое сечение не является искусственным явлением. Оно очень широко распространено в природе: золотое сечение можно найти в пропорциях тел многих растений и животных, а также морских раковин и птичьих яиц. Но наиболее впечатляющий пример "применения" природой принципа золотого сечения - человеческое тело. Оно целиком и его части (лицо, руки, кисти рук и т. п.) насквозь пронизаны пропорцией 1,618.

Принцип золотого сечения был открыт людьми еще в глубокой древности. Знаменитые египетские пирамиды в Гизе, например, основаны на пропорциях золотого сечения. Более молодые мексиканские пирамиды и античный храм Парфенон также содержат в себе пропорцию 1,618.

С развитием дизайна и технической эстетики действие закона золотого сечения распространилось на конструирование машин, мебели и т. д. Проектирование компьютерных интерфейсов - не исключение. Окна диалоговых окон и элементов управления, стороны которых относятся как 1,618, очень привлекательны для пользователей.

#### Кошелек Миллера

Этот принцип назван так в честь ученого-психолога Г. А. Миллера, который исследовал кратковременную память. Он пытался выяснить, сколько инокноции может запомнить человек без каких-либо специальных мнемонических приемов. Оказалось, что емкость памяти ограничена семью цифрами, семью буквами или названиями семи предметов. Это "магическое число" семь, служащее своего рода меркой памяти, и было проверено Миллером, который показал, что память действительно в среднем не может хранить более семи элементов; в зависимости от сложности элементов это число может колебаться в пределах от пяти до девяти.

Если необходимо в течение короткого времени сохранить инокноцию, включающую больше семи элементов, мозг почти бессознательно группирует эту инокноцию таким образом, чтобы число запоминаемых элементов не превышало предельно допустимого. Например, номер телефона 8 910 900 80 24, состоящий из десяти элементов, будет, скорее всего, запоминаться как четыре числовых элементов.

Применяя принцип кошелька Миллера в дизайне интерфейсов, следует группировать элементы в программе (кнопки на панелях инструментов, пункты меню,

закладки, опции на этих закладках и т. п.) с учетом этого правила - т. е. не более семи в группе, в крайнем случае - девяти.

### ***Принцип группировки***

Согласно этому правилу, экран программы должен быть разбит на ясно очерченные блоки элементов, может быть, даже с заголовком для каждого блока. При этом группировка, естественно, должна быть осмысленной: как расположение элементов в группах, так и расположение самих групп друг от друга должны быть продуманы.

Примеров реализации этого принципа очень много: это уже упоминавшиеся при разговоре о кошельке Миллера пункты меню, кнопочные панели инструментов, а также сгруппированные по назначению флажки и переключатели, с помощью которых настраиваются параметры работы программы в диалоговых окнах Свойства, Настройка и т. п..

### ***Бритва Оккама или KISS***

Философский принцип, носящий название "Бритва Оккама", гласит: "Не множить сущности без надобности". Или, как говорят американцы, KISS ("Keep It Simple, Stupid" - "Не усложняй, болван").

На языке интерфейсов это означает, что:

- любая задача должна решаться минимальным числом действий;
- логика этих действий должна быть очевидной для пользователя;
- движения курсора и даже глаз пользователя должны быть оптимизированы.

### ***Видимость отражает полезность***

Смысл этого принципа состоит в том, чтобы вынести самую важную информацию и элементы управления на первый план и сделать их легкодоступными пользователю, а менее важную - переместить, например, в меню.

Отличие принципа "Видимость отражает полезность" как раз и состоит в том, что интерфейс программы должен быть построен вокруг объектов, с которыми манипулирует пользователь, и отражать состояние текущего объекта. Реализацию этого принципа вы видите каждый раз, когда пользуетесь компьютером: контекстные панели инструментов в программах пакета *Microsoft Office*, которые меняются в зависимости от того, с какой частью программы (редактором, предварительным просмотром, рисованием и т. п.) и данный момент работает пользователь. Еще один пример - уже упоминавшееся меню Пуск в *Windows 7* и *Windows 10*, стартовый экран *Windows 8* по умолчанию в них видимы наиболее часто используемые, т. е. полезные для пользователя, пункты.

### ***Умное заимствование***

Заимствование широко распространенных приемов дизайна интерфейсов и удачных находок авторов конкурирующих программ позволяет резко сократить время обучения и повысить комфорт пользователя. При работе он будет использовать уже приобретенные навыки - этот вопрос затрагивает и принцип равенства между системой и реальным миром.

## ***Создание профессионального интерфейса***

### ***Стандартные элементы интерфейса***

Постарайтесь не использовать в своей программе нестандартные элементы интерфейса - например, командные кнопки не только с текстом, но и с рисунком, или комбинированные списки со сложной рамкой, которые в изобилии можно найти в oinline-коллекциях VCL и ActiveX-компонентов. Хотя бы потому, что именно так поступают профессиональные разработчики интерфейсов. Применение стандартных элементов позволяет пользователю быстрее адаптироваться к работе в новой программе и поэтому вероятность, что пользователь будет использовать программу более высокая.

Нестандартные подходы могут вызвать затруднения в использовании программы и как следствие программа будет менее востребована.

### **Замечание**

Обратите внимание, что применение стандартных элементов настоятельно рекомендуется только для оформления стандартных функций. В программах, решающих какие-то специфические задачи, одними стандартными элементами обойтись очень нелегко. Самый простой пример - графический редактор: организовать выбор цвета или просмотр цветовых каналов при помощи традиционных списков или панелей довольно затруднительно.

### ***Небольшая палитра инструментов***

Логическое развитие правила применения стандартных элементов: не используйте слишком большое их количество. Например, если где-то в одном из диалоговых окон программы вы поместили командную кнопку стандартного вида, то не нужно в другом месте программы использовать кнопку, отличающуюся от нее по оформлению.

### ***Одинаковое расстояние между элементами управления***

Если элементы управления на форме приложения располагаются на разном расстоянии между ними, то это сразу придает интерфейсу непрофессиональный вид. Наоборот, аккуратно выстроенные на форме кнопки, переключатели, флажки и другие элементы - признак качественной работы.

### ***TabIndex. "Правильный" порядок***

**TabIndex** - это порядок, в котором экранный курсор перемещается по элементам управления в форме при нажатии клавиши **<Tab>** на клавиатуре компьютера. На стадии разработки программы, при размещении элементов управления на форме, **TabIndex** эквивалентен тому порядку, в котором создаются эти элементы. Однако в процессе проектирования программы автор многократно меняет расположение элементов на форме, какие-то из них удаляет, добавляет новые компоненты. В результате почти всегда оказывается, что **TabIndex** не соответствует тому порядку, в котором визуальны расположены элементы, и при нажатии клавиши **<Tab>** курсор хаотично скачет по экрану вместо последовательного перемещения по компонентам.

Опытные разработчики по окончании проектирования окна обязательно проверяют **TabIndex** и, при необходимости, корректируют его. А вот начинающие авторы, увы, часто забывают об этом.

### ***Выбор шрифтов***

Здесь все просто - автор не должен выбирать никаких шрифтов. Оставьте их такими, какими они определены по умолчанию, а лучше - укажите в свойстве Шрифт (**Font**) соответствующие глобальные переменные **Windows: WindowText, MenuText** и т. д. В этом случае смена пользователем стандартных шрифтов Windows по своему вкусу с помощью Панели управления отразится и на внешнем виде вашей программы. Таким образом, пользователь, запустив ваш продукт, окажется в знакомом ему окружении.

Еще один вопрос - предпочтительное начертание шрифтов. Современные системы программирования допускают указание для свойства Шрифт, помимо обычного (**normal**) начертания еще и полужирное (**bold**), курсивное (**italic**) и подчеркнутое (**underlined**), а также их сочетания. Многие авторы, обрадовавшись предоставленным возможностям, охотно ими пользуются, применяя различные комбинации шрифтовых начертаний. На самом же деле в интерфейсах Windows-программ принято использовать всего два начертания: нормальное и полужирное (последнее - для выделения какой-либо важной инокноции, заголовков и т. п.). Применение курсива или подчеркивания, которое, к тому же, пользователь ошибочно может принять за гиперссылку - это дурной тон.



### ***Выбор цветов***

Здесь ситуация в точности такая же, как и со шрифтами: никакого выбора. При проектировании интерфейса нужно вообще забыть о свойстве Цвет (***Color***) элементов управления. Оставьте цвета стандартными, и пусть ваша программа выглядит так, как этого хочет ее пользователь, а не автор (хорошая идея - предусмотреть в программе возможность изменения цветовой гаммы различных частей интерфейса: многие пользователи любят настраивать цвета "под себя", причем так, что другому человеку такое "сочетание" цветов может показаться совершенно неудобоваримым). И в этом, нужно сказать, авторам программ повезло: они лишены одной из самых главных забот дизайнеров и художников, какие же цвета выбрать для своего нового творения.

Многие программисты все-таки жестко прописывают в своей программе используемые цвета, и это может служить причиной возникновения одного неприятного эффекта. Дело в том, что, как вы знаете, с помощью Панели управления можно легко изменить цветовую гамму Windows. Жестко фиксируя в своей программе выбранные цвета, автор не учитывает, что его программа выглядит хорошо только до тех пор, пока она работает на компьютере с такой же цветовой гаммой, как и на компьютере разработчика. Если же ее запускают в системе с другим цветовым оформлением, то результат может выглядеть, мягко говоря, не очень хорошо. Для предотвращения таких досадных ошибок в процессе разработки программы нужно время от времени переключаться на другие цветовые "схемы" Windows, проверяя, как ваша программа будет выглядеть на компьютере с нестандартной цветовой гаммой.

### ***Альтернативное управление***

Ваша программа должна одинаково хорошо управляться как с помощью мыши, так и клавиатуры. Не должно быть функций, которые можно выполнить только мышью (за исключением традиционно "мышинных" операций - например, рисования в графических редакторах). Наиболее популярные функции следует снабдить "горячими клавишами" для их быстрого вызова. При выборе комбинаций клавиш не забывайте о привычках и навыках пользователей: остановитесь на тех комбинациях, которые обычно используются в программах такого рода. Например, если вы разрабатываете файловый менеджер в стиле Проводника Windows, то лучше создавать комбинации, традиционные для Windows-программ.

#### **Стандартные комбинации клавиш в Windows**

Новое (окно, письмо, файл и т. п.)	<Ctrl>+<N>
Открыть	<Ctrl>+<O>
Сохранить	<Ctrl>+<S>
Печать	<Ctrl>+<P>
Отменить	<Ctrl>+<Z>
Повторить	<Ctrl>+<Y>
Вырезать	<Ctrl>+<X>, <Shift>+<Del>
Копировать	<Ctrl>+<C>, <Ctrl>+<Ins>
Вставить (из буфера обмена)	<Ctrl>+<V>, <Shift>+<Ins>
Выделить все	<Ctrl>+<A>
Найти	<Ctrl>+<F>
Обновить	<F5>
Справка	<F1>

## ***Кирпичики интерфейса***

Итак, вы познакомились с теорией проектирования интерфейсов и практическими рекомендациями по выбору типа интерфейса и приданию ему профессионального вида. Настало время поговорить о самых мелких частицах интерфейса, из которых, как дом из кирпичей, строится внешний вид программного продукта, - элементах управления (компонентах).

В графическом пользовательском интерфейсе элемент управления - это средство, при помощи которого пользователь взаимодействует с компьютерной программой. Качество этого взаимодействия зависит от двух аспектов:

- соответствия элемента управления выполняемой им задаче;
- от последовательности правил, по которым функционирует элемент управления.

Достаточно выбрать не тот инструмент работы или изменить правила, по которым он действует, и вы создадите проблемы для пользователей своей программы. Далее рассмотрим некоторые элементы управления и о наиболее часто возникающие проблемы, связанных с интеграцией компонентов и интерфейса, а также о том, как можно решить эти проблемы.

### ***Заголовок окна (окна)***

Хотя в палитрах доступных программисту компонентов современных систем создания приложений отсутствует такой элемент управления, как заголовок окна, он определяется свойством Заголовок (Title) объекта Окно (Window), ему нужно уделять не меньшее внимание, чем кнопкам, спискам и тому подобным элементам.

Заголовок главного окна программы традиционно используется для вывода информации о двух вещах: названии программы и названии документа, с которым в данный момент работает пользователь (если, конечно, в программе вообще предусмотрена обработка каких-либо документов). Так вот, вопрос в том, в каком порядке должна помещаться в окне программы такая информация?

Вопрос вовсе не праздный, как кажется на первый взгляд. Традиционно в программах с мультидокументным интерфейсом в заголовке окна сначала помещается название программы, а затем - название открытого документа. Однако для пользователя более удобным является другой порядок расположения информации в заголовке окна, когда первым идет название открытого документа, а после него - название программы. Такой подход к организации заголовка окна имеет следующие преимущества:

- если название документа помещается в заголовке окна первым, то оно всегда видимо на кнопке, представляющей соответствующую программу на Панели задач Windows, и узнать, какой в данный момент открыт документ, не представляет никакого труда. А вот если в заголовке окна сначала идет название программы, то, следовательно, оно и видно на Панели задач, и для того, чтобы выяснить, с каким именно документом работает данное окно, нужно переключиться в него (или навести курсор на кнопку Панели задач и подождать секунду, пока не появится всплывающая подсказка);
- т. к. при чтении взгляд человека скользит слева направо, то идущее в заголовке окна первым название документа (а именно для того, чтобы выяснить название текущего документа, и смотрит в заголовок окна пользователь чаще всего) читается наиболее легко. Если же применяется традиционная схема (Название программы - название документа), то пользователю сначала нужно пробежать глазами название программы, т. е. восприятие важной информации затрудняется.

### **Командные кнопки**

Наиболее частая ошибка начинающих разработчиков интерфейсов - использование в проекте нестандартных кнопок, включающих, помимо текста, также и графику. Во-первых, из-за обычно невысокого качества графики выглядят такие кнопки очень непрофессионально; во-вторых, почти всегда у автора программы не хватает рисунков на все кнопки, имеющиеся в программе, и часть кнопок приходится делать обычными, с текстовыми подписями и без рисунков. Нарушается принцип последовательности, что, сами понимаете, не добавляет интерфейсу привлекательности.

Пожалуй, единственная категория программ, в интерфейс которых можно включать кнопки с графикой, - это игры. Здесь такие кнопки могут оживить стандартный интерфейс Windows-программ, придать ему более праздничный и нарядный вид.

Командная кнопка - один из тех элементов управления, для которых наиболее часто применяется динамически изменяемое свойство *isEnabled*, т. е. отключение кнопки, когда она перестает реагировать на нажатия, и ее включение, в зависимости от текущего состояния программы. Для индикации состояния отключения граница вокруг кнопки и буквы текста на ней становятся светло-серыми.

Динамическое отключение и включение кнопки выглядит очень эффектно и производит впечатление высокого искусственного интеллекта программы (в большинстве случаев так и есть). Например, в диалоговых окнах кнопка ОК остается недоступной до тех пор, пока пользователь не введет необходимые данные, т. е. при каждом изменении информации происходит ее проверка. И вот здесь нужно быть очень осторожным. Дело в том, что пользователь при вводе сложных данных может не иметь понимания о том, что именно он делает неправильно. "Серая" отключенная кнопка только говорит ему о том, что он что-то сделал не так, как нужно - например, ввел неверные или неполные данные, но вот в чем конкретно состоит проблема... И никто не может подсказать пользователю, что необходимо сделать, чтобы заветная кнопка наконец стала реагировать на нажатия мышью. Было бы гораздо лучше, если бы кнопка ОК была доступна, а после ее нажатия выдавалось бы сообщение об ошибке.

### **Меню**

Список команд по работе с программой, предлагаемых на выбор пользователя - одно из самых старых и универсальных средств организации интерфейса компьютерных программ.

Присутствия меню в главном окне программы, является правилом хорошего тона. Отказываясь от включения меню в проект своей программы, автор игнорирует опыт и навыки пользователей, заставляет их отказываться от стиля работы, к которому они привыкли. Хорошее меню - это не просто список команд, это еще и многофункциональная "шпаргалка" по работе с программой, "шпаргалка", которая всегда под рукой (как известно, читать справочные файлы пользователи не очень любят). Достаточно провести мышью по строке меню вверху экрана, и можно выяснить набор функций программы, комбинации "горячих клавиш" и даже - объяснение назначения кнопок на панели инструментов.

И, конечно же, не стоит забывать о контекстных меню - меню, появляющихся при щелчке правой кнопкой мыши на каком-либо объекте и содержащими команды для работы с этим объектом. Разработка таких меню - дело, хотя технически очень простое, но при этом довольно кропотливое. Тем не менее, затраты времени и сил обязательно окупятся. Операционная система Windows приучила пользователей к мысли, что при щелчке правой кнопкой мыши появляется контекстное меню. Щелчок правой кнопкой мыши стал уже настолько естественным действием, что программа, не "понимающая" его, напоминает человека, который не слышит, что ему говорят. Контекстное меню - одна из самых сильных привычек пользователей и навыков, который имеют даже малоопытные

новички. Нужно обязательно учитывать это при разработке интерфейсов собственных программ.

### ***Списки***

Элемент управления Список (ListBox) один из самых популярных во всей палитре компонентов для создания интерфейса. Он позволяет легко просматривать большие объемы информации и осуществлять выделение нужных строк. Однако у него есть неприятная особенность: отсутствие горизонтальной линейки прокрутки. Из-за этого слишком длинные строки обрезаются на границе элемента, что особенно неприятно, если таким образом становится недоступной какая-либо важная информация

Во избежание возникновения подобных проблем нужно тщательно протестировать работу программы, чтобы выяснить, возможна ли ситуация, что в список будут выведены слишком длинные строки, чтобы уместиться в нем полностью. Если это не исключается, то можно предусмотреть средства, позволяющие все-таки полностью просмотреть "обрезанную" строку, например, при двойном щелчке мышью на интересующей пользователя строке выводить на экран небольшое окошко, где требуемый текст отображается полностью.

### ***Флажки и переключатели***

Флажки (CheckBox) и переключатели (RadioButton) используются для одной цели: для выбора из группы предложенных вариантов. Разница между ними, как вам, наверное, известно, состоит в том, что флажки используются для выбора одновременно нескольких вариантов из группы, а переключатели позволяют сделать выбор в пользу только одного варианта.

### ***Панели инструментов***

Кнопочные панели инструментов (ToolBar) - излюбленный компонент многих разработчиков. С ним окно программы сразу приобретает более привлекательный, солидный и профессиональный вид. Любовь программистов к панелям инструментов настолько велика, что они, как я уже рассказывал чуть выше, даже отказываются в пользу них от святой святых пользовательского интерфейса - меню!

Конечно, популярность компонента приводит к тому, что многие начинающие разработчики при включении панелей инструментов в свой проект сталкиваются с некоторыми "подводными камнями".

Часто задают вопрос: "Можно ли для панелей инструментов своих программ брать пиктограммы (иконки) из чужих продуктов?" Если кратко ответить на этот вопрос, то такое "заимствование" запрещено. Пиктограммы, как произведение изобразительного искусства (пусть и очень миниатюрное), охраняется авторским правом, и только автор (или лицо, которому он передал свои права) может использовать эти пиктограммы, а также разрешать или запрещать их использование.

Однако все зависит от самих владельцев прав на так понравившиеся вам пиктограммы. Некоторые из них, закрыв глаза, смотрят на нарушения своих авторских прав, справедливо полагая, что копия никогда не превзойдет оригинал. Например, считается незастыдным заимствовать для своих проектов пиктограммы из продуктов пакета Microsoft Office (а их там огромное количество - подойдет, наверное, для любой программы). Сама корпорация Microsoft не считает это злом, все это способствует унификации интерфейсов программ, которая является одним из положений идеологии Windows.

А вот например, если вы решите позаимствовать для своей программы пиктограммы популярного менеджера загрузок ReGet (<http://www.reget.com>), то вряд ли это окажется незамеченным. Эти пиктограммы были созданы специально для ReGet в одной из самых крупных студий дизайна в России - Студии Артемия Лебедева (<http://www.design.ro>), и разработчик ReGet - компания ReGet Software, конечно же,

заинтересована в том, чтобы никто, кроме нее, не пользовался этими уникальными пиктограммами.

Но, если уж вы решили позаимствовать для своей программы пиктограммы из другого продукта, постарайтесь, чтобы у вас они обозначали те же функции, что и в оригинальной программе. Иначе будет нарушен принцип последовательности, и пользователи будут испытывать затруднения с освоением вашего программного продукта.

### ***Вкладки***

Вкладки (TabControl) широко используются при проектировании интерфейсов современных программ. Нужно отдать должное этому элементу управления: он не только выглядит не менее эффектно, чем кнопочная панель инструментов, но и очень эффективен, позволяя логически группировать большое количество информации, позволяя пользователю комфортно воспринимать ее.

Да, вкладки позволяют упорядочить большие объемы данных, однако это полезное свойство вкладок сходит на нет, если число самих вкладок становится слишком большим. Здесь явно наблюдается противоречие с правилом кошелька Миллера (см. разд. "Другие принципы построения интерфейсов" данной главы), определяющим, что человек может удерживать в своей кратковременной памяти семь плюс-минус две сущности. Поэтому в нескольких рядах вкладок, которые уже перестают уместиться в рамках диалогового окна, очень легко запутаться. Даже тот десяток вкладок, которые находятся в окне Параметры Microsoft Word, вызывает многочисленные нарекания со стороны пользователей. Действительно, мало кому не приходилось беспорядочно щелкать по вкладкам окна настроек Microsoft Word, чтобы отыскать нужную опцию.

Но что же тогда делать авторам функционально очень сложных программ, где диалоговые окна заполнены самой разной информацией? Без вкладок все окажется словно сваленным в одну большую кучу, в которой разобраться будет гораздо труднее, чем в десятке вкладок.

### ***Всплывающие подсказки***

Всплывающие подсказки (ToolTip) - это, конечно, не самостоятельные элементы управления, хотя в коллекциях компонентов и можно найти модули для создания сложных (многострочных, с разными типами шрифтов, с графикой и т. п.) подсказок. На практике они применяются как дополнение к другим элементам управления, выдавая пояснение относительно назначения соответствующего элемента. Стоит пользователю на секунду задержать курсор мыши над интересующим его элементом, как появляется небольшой желтый прямоугольник с поясняющим текстом.

Нужно помнить, что всплывающие подсказки - спутник далеко не любого компонента на форме приложения.

## Элементы интерфейса

### Групповые панели

#### Принцип группировки

Согласно этому правилу, экран программы должен быть разбит на ясно очерченные блоки элементов, может быть, даже с заголовком для каждого блока. При этом группировка, естественно, должна быть осмысленной: как расположение элементов в группах, так и расположение самих групп друг от друга должны быть продуманы.

Примеров реализации этого принципа очень много: это уже упоминавшееся при разговоре о кошельке Миллера пункты меню, кнопочные панели инструментов, а также сгруппированные по назначению флажки и переключатели, с помощью которых настраиваются параметры работы программы в диалоговых окнах Свойства, Настройка и т. п..

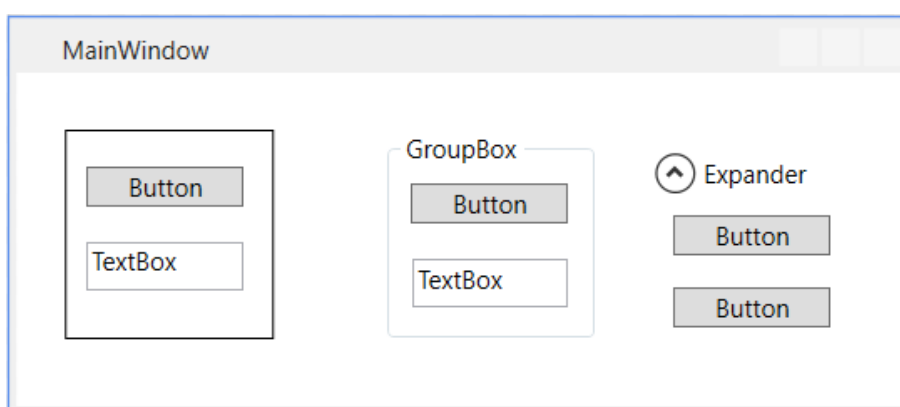


рис. 1. Группировка – элементы **Border**, **GroupBox** и **Expander**.

Для группировки используются элементы **Border**, **GroupBox** и **Expander**. Основное отличие у **Border** нет поясняющей надписи у группового окна есть, **Expander** может скрывать свое содержимое.

#### Компонент **Border** – панель

Компонент **Border** представляет собой контейнер для других компонентов и позволяет легко управлять ими. Компоненты, которые находятся на панели, наследуют свойства компонента **Border**. Например, чтобы сделать недоступными все компоненты на панели, достаточно присвоить значение **False** свойству **Enabled** панели. Свойства компонента **Border** приведены в таблице.

Таблица 1. Основные свойства компонента **Border**

Свойство	Комментарий
Brush Background	Получает или задает цвет фона для элемента управления.
Brush BorderBrush	Получает или задает цвет рамки для элемента управления.
bool IsEnabled	Свойство позволяет сделать доступными ( <b>True</b> ) или недоступными ( <b>False</b> ) все компоненты, которые размещены на панели.
Thickness BorderThickness	Толщина рамки компонента.
bool IsVisible	Свойство позволяет отображать ( <b>True</b> ) и скрывать ( <b>False</b> ) панель.

### Компонент *GroupBox* – групповая панель

Этот компонент представляет собой контейнер с рамкой и надписью, объединяющий группу компонентов. Обычно он используется для объединения компонентов в группы по функциональным признакам. Основные свойства компонента приведены в таблице.

Таблица 2. Основные свойства компонента *GroupBox*

Свойство	Комментарий
Brush Background	Получает или задает цвет фона для элемента управления.
Brush BorderBrush	Получает или задает цвет рамки для элемента управления.
bool IsEnabled	Свойство позволяет сделать доступными ( <i>True</i> ) или недоступными ( <i>False</i> ) все компоненты, которые размещены на панели.
Thickness BorderThickness	Толщина рамки компонента.
Object Header	Текст связанный с этим элементом (надпись на элементе).
bool IsVisible	Свойство позволяет отображать ( <i>True</i> ) и скрывать ( <i>False</i> ) панель.

### Меню

Список команд по работе с программой, предлагаемых на выбор пользователя - одно из самых старых и универсальных средств организации интерфейса компьютерных программ.

Присутствия меню в главном окне программы, является правилом хорошего тона. Отказываясь от включения меню в проект своей программы, автор игнорирует опыт и навыки пользователей, заставляет их отказываться от стиля работы, к которому они привыкли. Хорошее меню - это не просто список команд, это еще и многофункциональная "шпаргалка" по работе с программой, "шпаргалка", которая всегда под рукой (как известно, читать справочные файлы пользователи не очень любят). Достаточно провести мышью по строке меню вверху экрана, и можно выяснить набор функций программы, комбинации "горячих клавиш".

Теперь - о том, что же можно помещать в меню. Некоторые авторы считают, что для простых программ меню вовсе не нужно, т. к. не стоит делать меню ради двух-трех пунктов. Но даже программа, которая ничего не делает, достойна меню как минимум из двух пунктов: **Файл**, и **Справка**, включающий подпункт **О программе** (ведь потенциальные пользователи должны знать, куда отправлять свои денежки!).

Желательна минимальная конфигурация меню. Пункт **Файл**, где находится команда **Выход**, даже если программа непосредственно с файлами не работает, очень желателен, я бы сказал - необходим. Просто **Файл** - настолько привычный элемент меню, что все основные команды (начать новую работу, сохранить результаты, выйти из программы) пользователи ищут прежде всего там. Пункт **Справка**, включающий подпункт **О программе** я рекомендую включать в любое меню по двум причинам: во-первых, пользователь сразу видит, что в программе есть справочная система и есть куда обратиться в случае возникновения затруднений; во-вторых, если в строке меню содержится только один пункт - Файл, то это выглядит не очень красиво. Добавление еще одного пункта уравновешивает картину.

### Компонент *Menu* – главное меню

Этот компонент представляет собой главное меню программы. Этот компонент обычно размещается в верхней части формы. Обычно на форму помещается один компонент *Menu*.

Разместить заготовку меню можно так, в данном случае с привязкой к верхней части окна:

```
<Menu VerticalAlignment="Top">
...
</Menu>
```

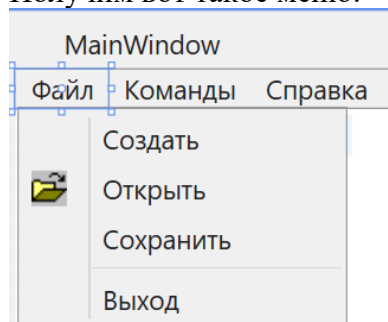
### Компонент *MenuItem* – элемент меню

После добавления к форме программы компонента *Menu* его необходимо настроить. Т.е. добавить элементы меню. Для добавления пунктов меню используется элемент *MenuItem*. Путем использования элемента *MenuItem* и вложения его друг в друга строится иерархическая система меню. **Обратите внимание, что меню строится вручную в окне разметки интерфейса.**

Построим простое меню, используя окно разметки интерфейса:

```
<Menu VerticalAlignment="Top" Grid.Row="0" >
  <MenuItem Header="Файл">
    <MenuItem Header="Создать"/>
    <MenuItem Header="Открыть">
      <MenuItem.Icon>
        <Image Source="image\Open.bmp"/></Image>
      </MenuItem.Icon>
    </MenuItem>
    <MenuItem Header="Сохранить"/></MenuItem>
    <Separator/></Separator>
    <MenuItem Header="Выход"/></MenuItem>
  </MenuItem>
  <MenuItem Header="Команды">
  </MenuItem>
  <MenuItem Header="Справка"/></MenuItem>
</Menu>
```

Получим вот такое меню:



Пункты меню можно отделять друг от друга с помощью разделителей. Для этого используйте элемент *Separator*.

```
<MenuItem Header="Сохранить"/></MenuItem>
<Separator/></Separator>
<MenuItem Header="Выход"/></MenuItem>
```

Рассмотрим основные свойства элемента *MenuItem*.

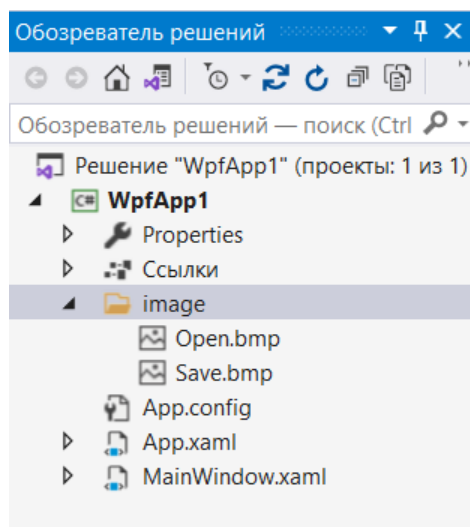
Таблица 3. Основные свойства элементов меню (*MenuItem*)

Свойство	Комментарий
<code>object</code> Header	Получает или задает элемент, задающий подпись элементу управления.
<code>object</code> Icon	Возвращает или задает значок, появляющийся в элементе меню.



<code>bool</code> IsCheckable	Вызывает значение показывающее, можно ли отметить флажком элемент меню.
<code>bool</code> IsChecked	Возвращает или задает значение, показывающее, установлен ли флажок для элемента меню.
<b>События</b>	<b>Комментарий</b>
<code>Click</code>	Происходит при нажатии элемента управления

Картинки для меню удобно добавлять в структуру проекта. Выделяем в *Обозревателе решений* главный узел проекта и чтобы сформировать правильную файловую структуру проекта сначала добавляем папку *image* командой **Проект – Создать папку**. Выделяем папку *image* и добавляем в нее картинки командой **Проект – Добавить существующий элемент**. Картинки можно найти в папке *C:\tools\Иконки и глифы*.



### Контекстное меню

И, конечно же, не стоит забывать о контекстных меню - меню, появляющихся при щелчке правой кнопкой мыши на каком-либо объекте и содержащими команды для работы с этим объектом. Разработка таких меню - дело, хотя технически очень простое, но при этом довольно кропотливое. Тем не менее, затраты времени и сил обязательно окупятся. Щелчок правой кнопкой мыши стал уже настолько естественным действием, что программа, не "понимающая" его, напоминает человека, который не слышит, что ему говорят. Контекстное меню - одна из самых сильных привычек пользователей и навыков, который имеют даже малоопытные новички. Нужно обязательно учитывать это при разработке интерфейсов собственных программ.

Некоторые авторы программ забывают о значении термина "контекстное меню". Такое меню должно содержать только те команды, которые относятся к объекту, которому принадлежит меню. Вызывать по щелчку правой кнопкой мыши меню из пары десятков пунктов и уж тем более дубликат главного меню программы не имеет смысла и практическая ценность такого "контекстного" меню равна нулю.

### Компонент *ContextMenu* – контекстное меню

Класс *ContextMenu* служит для создания контекстных всплывающих меню, отображающихся после нажатия на правую кнопку мыши. Этот элемент также содержит коллекцию элементов *MenuItem*. Однако сам по себе *ContextMenu* существовать не может и должен быть прикреплен к другому элементу управления. Для этого у элементов есть свойство *ContextMenu*. Создается контекстное меню также как и обычное вручную в окне

разметки интерфейса. Например, создадим контекстное меню для главного окна программы:

```
<Window.ContextMenu>
  <ContextMenu>
    <MenuItem Header="Копировать"></MenuItem>
    <MenuItem Header="Вставить"></MenuItem>
    <MenuItem Header="Вырезать"></MenuItem>
    <MenuItem Header="Удалить"></MenuItem>
  </ContextMenu>
</Window.ContextMenu>
```

### Панели инструментов

Кнопочные панели инструментов (*ToolBar*) - излюбленный компонент многих разработчиков. С ним окно программы сразу приобретает более привлекательный, солидный и профессиональный вид.

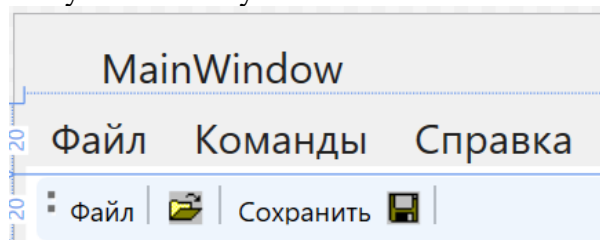
#### Компонент *ToolBar* – панель инструментов

После добавления к форме программы компонента *ToolBar* его необходимо настроить, т.е. добавить элементы панели инструментов. Добавить можно практически любые простые элементы, которые мы уже использовали ранее. **Обратите внимание, что панель инструментов строится вручную в окне разметки интерфейса.**

Создадим простую панель элементов: кнопка с надписью, кнопка с картинкой и кнопки с картинкой и надписью:

```
<ToolBar VerticalAlignment="Top" Grid.Row="1" >
  <Button Content="Файл"></Button>
  <Separator></Separator>
  <Button>
    <Image Source="image\Open.bmp"></Image>
  </Button>
  <Separator></Separator>
  <Button>
    <StackPanel Orientation="Horizontal" ToolTip="Сохранить документ">
      <TextBlock>Сохранить</TextBlock>
      <Separator Opacity="0" Width="5"></Separator>
      <Image Source="image\Save.bmp"></Image>
    </StackPanel>
  </Button>
  <Separator></Separator>
</ToolBar>
```

Получим вот такую панель элементов:



## Строка статуса

В нижней части окна обычно располагается строка состояния. Она используется для отображения различной информации о текущем состоянии приложения, например о положении курсора, о количестве слов, о прогрессе заданий и т.д.

### Компонент *StatusBar* – строка статуса

Компонент *StatusBar* представляет собой область (панель) для вывода служебной информации. Обычно такая панель располагается в нижней части программы и может разбиваться на несколько частей. Строка статуса строится из компонентов уже известных нам ранее, в которые размещается нужная информация. **Обратите внимание, что строка статуса строится вручную в окне разметки интерфейса.**

Например, создадим строку статуса для размещения даты и времени.


В окне разметки интерфейса создаем соответствующую разметку для строки статуса:

```
<StatusBar Grid.Row="3">
    <TextBlock x:Name="time" Text="12:12"></TextBlock>
    <Separator></Separator>
    <TextBlock x:Name="data" Text="20.07.2021"></TextBlock>
</StatusBar>
```

Далее надо получать текущую дату и время и выводить в строку статуса. Для обновления времени используем таймер:

```
DispatcherTimer timer;//Описываем таймер
ссылка: 1
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    //Добавляем таймер
    timer = new DispatcherTimer();
    timer.Tick += Timer_Tick;
    timer.Interval = new TimeSpan(0,0,0,1,0);
    timer.IsEnabled = true;
}
//Создаем вручную событие таймера
ссылка: 1
private void Timer_Tick(object sender, EventArgs e)
{
    DateTime d = DateTime.Now;//Создание объекта
    time.Text = d.ToString("HH:mm");//Время
    data.Text = d.ToString("dd.MM.yyyy");//Дата
}
```

Получим вот такую строку статуса:



12:12 | 20.07.2021

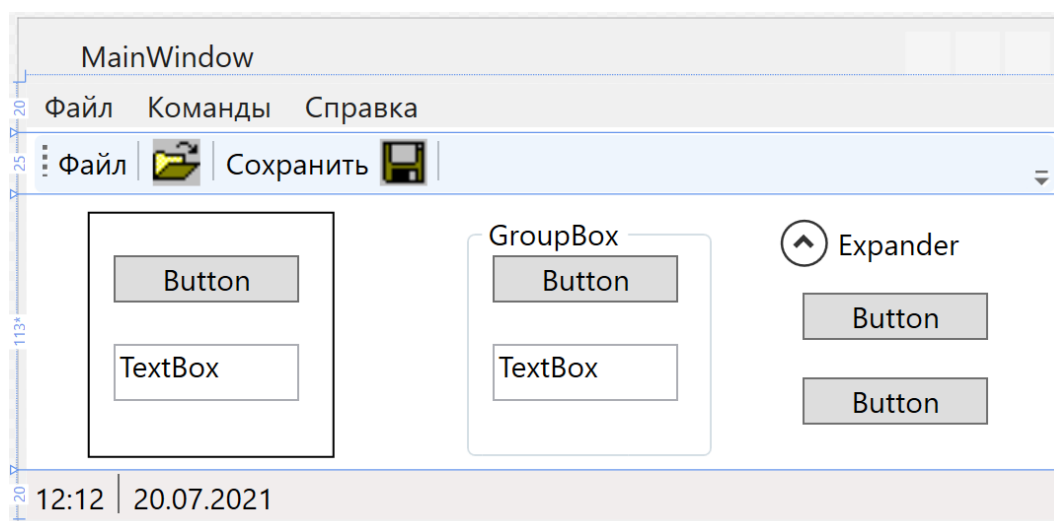
## Размещение элементов в контейнере *Grid*

При размещении различных элементов в окне программы могут возникнуть проблемы с их позиционированием. Например, при одновременном использовании меню и панели инструментов с относительной привязкой к верхнему краю контейнера *Grid* эти элементы будут накладываться друг на друга. Можно, конечно, использовать абсолютную привязку через свойство *Margin*, но такой подход является не самым лучшим. Ниже приведен пример разметки меню и панели инструментов с относительной привязкой.

```
<ToolBar VerticalAlignment="Top"
<Menu VerticalAlignment="Top" G
```

Чтобы интерфейс построить более правильно, контейнер *Grid* можно разбить на ячейки и информацию размещать по ячейкам. Для элементов, которые были рассмотрены выше, можно создать четыре строки, в которые поместить интерфейс программы. Первая строка будет содержать меню, вторая строка – панель инструментов, третья строка – окно программы и четвертая – строка статуса.

Чтобы создать строки в контейнере *Grid* выделите его и щелкните на левой границе элемента, указатель мыши при этом имеет вид белой стрелки с плюсиком. Горизонтальные линии покажут строки в контейнере *Grid*, см. рисунок ниже.



При добавлении строк в контейнер *Grid* автоматически меняется разметка этого элемента в XAML, в которой можно задать высоту каждой строки. Если высота строки не должна менять при масштабировании окна размер задается числовой константой, если высота строки будет изменять при масштабировании окна, к размеру добавляют знак \*.

```
<Grid.RowDefinitions>
    <RowDefinition Height="20"/>
    <RowDefinition Height="25"/>
    <RowDefinition Height="113*"/>
    <RowDefinition Height="20"/>
</Grid.RowDefinitions>
```

Для размещения элементов по разным строкам, переносим элемент в нужную строку или в разметке интерфейса вручную указываем позиционирование элемента указав *Grid.Row="номер строки"*, см. пример ниже:

```
<ToolBar VerticalAlignment="Top" Grid.Row="1">  
<Menu VerticalAlignment="Top" Grid.Row="0" >
```

### Вкладки

Вкладки (*TabControl*) широко используются при проектировании интерфейсов современных программ. Нужно отдать должное этому элементу управления: он не только выглядит не менее эффектно, чем кнопочная панель инструментов, но и очень эффективен, позволяя логически группировать большое количество информации, позволяя пользователю комфортно воспринимать ее.

Да, вкладки позволяют упорядочить большие объемы данных, однако это полезное свойство вкладок сходит на нет, если число самих вкладок становится слишком большим. Здесь явно наблюдается противоречие с правилом кошелька Миллера (см. разд. "Другие принципы построения интерфейсов" данной главы), определяющим, что человек может удерживать в своей кратковременной памяти семь плюс-минус две сущности. Поэтому в нескольких рядах вкладок, которые уже перестают уместиться в рамках диалогового окна, очень легко запутаться. Даже тот десяток вкладок, которые находятся в окне Параметры Microsoft Word, вызывает многочисленные нарекания со стороны пользователей. Действительно, мало кому не приходилось беспорядочно щелкать по вкладкам окна настроек Microsoft Word, чтобы отыскать нужную опцию.

Но что же тогда делать авторам функционально очень сложных программ, где диалоговые окна заполнены самой разной информацией? Без вкладок все окажется словно сваленным в одну большую кучу, в которой разобраться будет гораздо труднее, чем в десятке вкладок.

### Компонент *TabControl* – многостраничная панель

Многостраничные панели позволяют экономить пространство окна приложения, размещая на одном и том же месте страницы разного содержания (см. рис. 2).

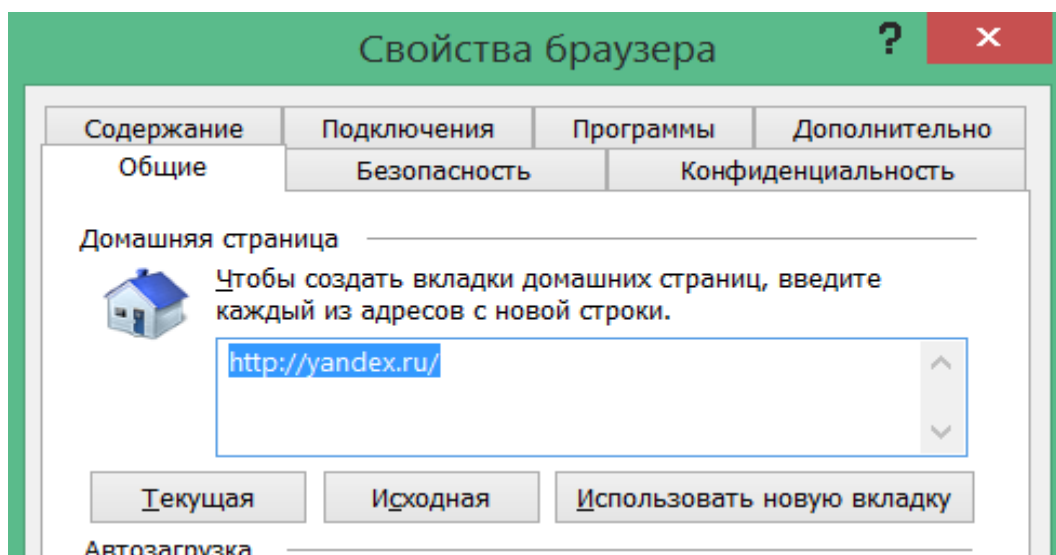


рис. 2. Многостраничная панель

Особенности работы с элементом заключаются в том, что имеется главный элемент *TabControl* и страницы *TabPage*. Щелчок на шапке элемента (красная галочка) выделяет главный элемент *tabControl*. Щелчок внутри элемента или на вкладке *TabPage* (зеленая галочка) выделяет выбранную страницу *TabPage*.

Добавлять и удалять страницы можно через контекстное меню, вызываемое на шапке элемента **TabControl** (красная галочка). В контекстном меню вы можете видеть команды: Добавить **TabItem**; Удалить. Также добавлять и удалять элементы можно в окне разметки интерфейса, см. рисунок 3. Размещение информации в на страницах осуществляется также как и в обычном окне.

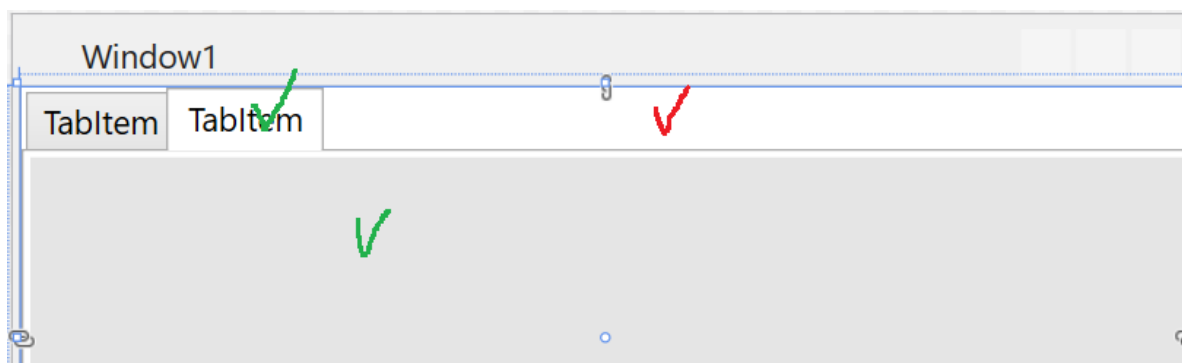


рис. 3. Заготовка многостраничной панели

При добавлении элемента **TabControl** из **Панели элементов** в XAML добавляется заготовка с двумя вкладками, пример см. ниже:

```
<TabControl>
  <TabItem Header="TabItem">
    <Grid Background="#FFE5E5E5">
      <Button Content="Button" HorizontalAlignment="Left" Margin="150,44,0,0"
    </Grid>
  </TabItem>
  <TabItem Header="TabItem">
    <Grid Background="#FFE5E5E5">
      <ComboBox HorizontalAlignment="Left" Margin="159,50,0,0" VerticalAlignm
    </Grid>
  </TabItem>
</TabControl>
```

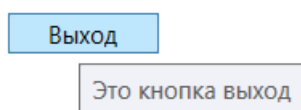
### Всплывающие подсказки

Всплывающие подсказки (ToolTips) - это, конечно, не самостоятельные элементы управления. На практике они применяются как дополнение к другим элементам управления, выдавая пояснение относительно назначения соответствующего элемента. Стоит пользователю на секунду задержать курсор мыши над интересующим его элементом, как появляется небольшой желтый прямоугольник с поясняющим текстом.

Нужно помнить, что всплывающие подсказки - спутник далеко не любого компонента на форме приложения.

Для определения всплывающей подсказки у элементов уже есть свойство **ToolTip**, которому можно задать текст, отображаемый при наведении:

```
<Button Content="Выход" ToolTip="Это кнопка выход"/>
```



Всплывающие подсказки можно применять не только кнопкам, но и ко всем другим элементам управления.

Поскольку ***ToolTip*** является элементом управления содержимого, то в него можно встроить другие элементы для создания более богатой функциональности, например, картинки.



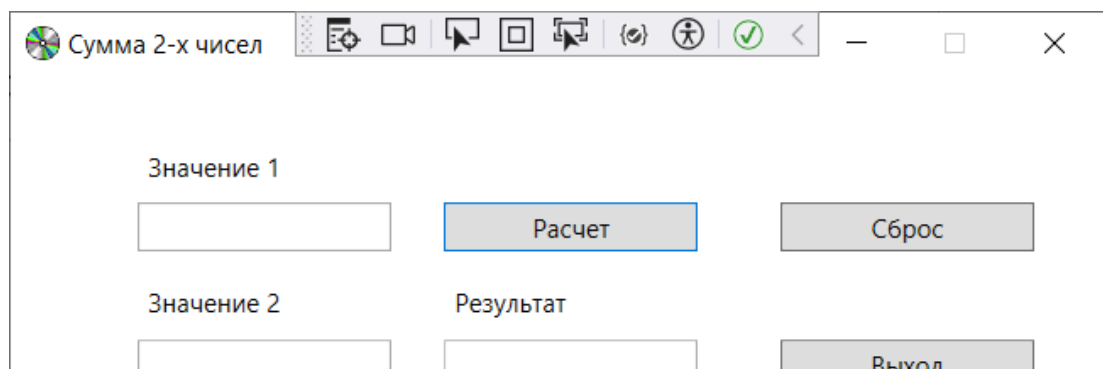
## Разработка масштабируемого интерфейса приложения

### Основы компоновки WPF

Что такое компоновка? Компоновка – это организация содержимого вашего окна. На этом этапе разработчик размечает всё доступное пространство на несколько областей, группируя контент на форме. Чем грамотнее планировка, тем удобнее пользоваться конечным продуктом.

Возможно вами ранее **не применялись правила правильной компоновки**, все элементы имеют абсолютное позиционирование. Для простых приложений компоновка не имела значения, но для разработки профессиональных приложений необходимо использовать правильную компоновку приложения.

При использовании не правильной компоновки возможны различные артефакты при отображении интерфейса приложения. Например, в примере калькулятор, который был рассмотрен ранее при разработке в конструкторе все отображается правильно, а у запущенного приложения могут съезжать элементы и отображаться неправильно см. пример ниже:



```
<Label Content="Значение 1" HorizontalAlignment="Left" Margin="60,30,0,0" VerticalAlignment="Top" Width="110"/>
<Label Content="Значение 2" HorizontalAlignment="Left" Margin="60,94,0,0" VerticalAlignment="Top" Width="110"/>
<TextBox x:Name="tbZn1" HorizontalAlignment="Left" Height="23" Margin="60,60,0,0" TextWrapping="Wrap"
    VerticalAlignment="Top" Width="120" MaxLength="9" TextChanged="tbZn1_TextChanged"/>
<TextBox x:Name="tbZn2" HorizontalAlignment="Left" Height="23" Margin="60,125,0,0" TextWrapping="Wrap"
    VerticalAlignment="Top" Width="120" MaxLength="9"/>
<TextBox x:Name="tbRez" HorizontalAlignment="Left" Height="23" Margin="205,125,0,0" TextWrapping="Wrap" Vertical
<Button Content="Расчет" HorizontalAlignment="Left" Margin="205,60,0,0" VerticalAlignment="Top" Width="120"
    Height="23" Click="btnCalc_Click" IsDefault="True" />
<Button Content="Выход" HorizontalAlignment="Left" Margin="364,125,0,0" VerticalAlignment="Top" Width="120" Height
<Button Content="Сброс" HorizontalAlignment="Left" Margin="364,60,0,0" VerticalAlignment="Top"
    Width="120" Height="23" Click="btnClear_Click" IsCancel="True"/>
<Label Content="Результат" HorizontalAlignment="Left" Margin="205,94,0,0" VerticalAlignment="Top" Width="110"/>
```

Чтобы перейти уже непосредственно к созданию красивых в том числе масштабируемых интерфейсов и их компонентов, сначала необходимо познакомиться с компоновкой. Компоновка (*layout*) представляет собой процесс размещения элементов внутри контейнера. Возможно, вы обращали внимание, что одни программы и веб-сайты на разных экранах с разным разрешением выглядят по-разному: где-то лучше, где-то хуже. В большинстве своем такие программы используют жестко закодированные в коде размеры элементов управления. WPF уходит от такого подхода в пользу так называемого "резинового дизайна", где весь процесс позиционирования элементов осуществляется с помощью компоновки.

Благодаря компоновке мы можем удобным нам образом настроить элементы интерфейса, позиционировать их определенным образом. Например, элементы компоновки в WPF позволяют при изменении размера окна - сжатии или растяжении



масштабировать элементы, что очень удобно, а визуально не создает всяких шероховатостей типа незаполненных пустот на форме.

В WPF компоновка определяется используемыми контейнерами. Окно WPF должно отвечать следующим принципам:

- **Элементы не должны иметь явно установленных размеров.** Вместо этого они растут, чтобы уместить свое содержимое. Например, кнопка увеличивается при добавлении в нее текста. Можно ограничить элементы приемлемыми размерами, устанавливая максимальное и минимальное их значение.
- **Элементы не указывают свою позицию в абсолютных (экранных) координатах.** Вместо этого они упорядочиваются на основе размера, порядка и специфичных настроек контейнера компоновки. Не используйте свойство *Margin* для позиционирования элементов относительно окна! Его единственная истинная функция – отступить от соседнего элемента, чтобы они не выглядели слипшимися.
- **Контейнеры компоновки "разделяют" доступное пространство между своими дочерними элементами.** Они пытаются обеспечить для каждого элемента его предпочтительный размер (на основе его содержимого), если только позволяет свободное пространство. Они могут также выделять дополнительное пространство одному или более дочерним элементам.
- **Контейнеры компоновки поддерживают вложение.** Вкладывайте один контейнер в другой, чтобы разбивать блоки пространства на подблоки, тем самым уточняя разметку вашего окна.

Если вы последуете этим правилам при построении WPF-приложения, то получите лучший и более гибкий пользовательский интерфейс. Если же вы нарушаете эти правила, то получите пользовательский интерфейс, который не очень хорошо подходит для WPF и его будет значительно сложнее сопровождать.

В WPF компоновка осуществляется при помощи специальных контейнеров. Среда разработки предоставляет нам следующие контейнеры: *Grid*, *UniformGrid*, *StackPanel*, *WrapPanel*, *DockPanel* и *Canvas*.

Различные контейнеры могут содержать внутри себя другие контейнеры. Контейнеры компоновки позволяют эффективно распределить доступное пространство между элементами, найти для него наиболее предпочтительные размеры.

WPF при компоновке и расположении элементов внутри окна нам надо придерживаться следующих принципов:

- Нежелательно указывать явные размеры элементов (за исключением минимальных и максимальных размеров). Размеры должны определяться контейнерами.
- Нежелательно указывать явные позицию и координаты элементов внутри окна. Позиционирование элементов всецело должно быть прерогативой контейнеров. И контейнер сам должен определять, как элемент будет располагаться. Если нам надо создать сложную систему компоновки, то мы можем вкладывать один контейнер в другой, чтобы добиться максимально удобного расположения элементов управления.

### **Контейнер *Grid***

Это наиболее мощный и часто используемый контейнер, напоминающий обычную таблицу. Он содержит столбцы и строки, количество которых задает разработчик. Для

определения строк используется свойство **RowDefinitions**, а для определения столбцов – свойство **ColumnDefinitions**:

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

Также разбить на строки и столбцы можно щелкая мышью на левую и верхнюю границу элемента Grid.

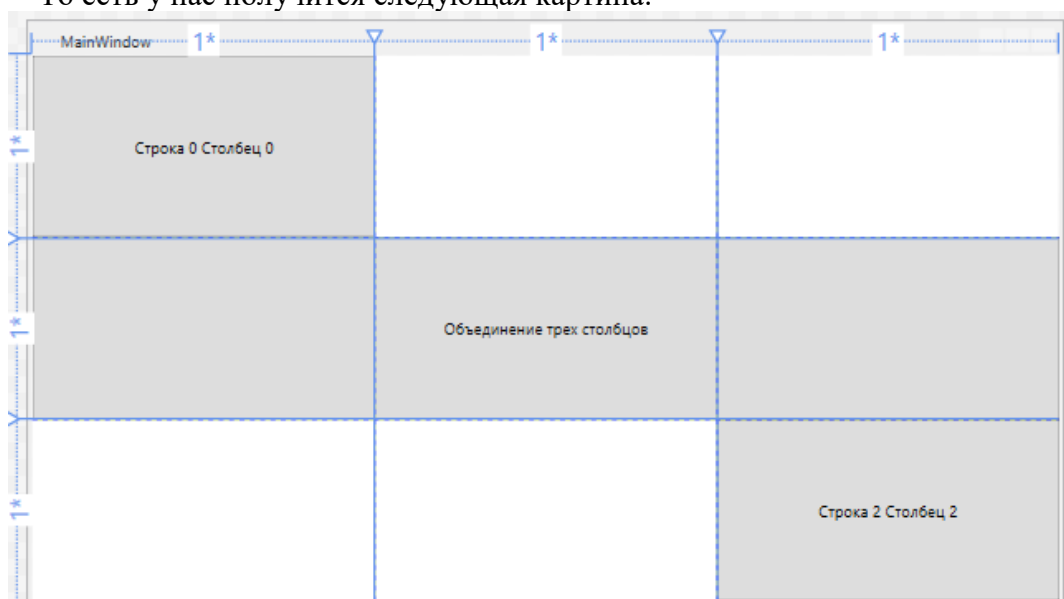
Каждая строка задается с помощью вложенного элемента **RowDefinition**, который имеет открывающий и закрывающий тег. При этом задавать дополнительную информацию необязательно. То есть в данном случае у нас определено в гриде 3 строки.

Каждая столбец задается с помощью вложенного элемента **ColumnDefinition**. Таким образом, здесь мы определили 3 столбца, т.е. в итоге у нас получится таблица 3x3.

Чтобы задать позицию элемента управления с привязкой к определенной ячейке **Grid**, в разметке элемента нужно прописать значения свойств **Grid.Column** и **Grid.Row**, тем самым указывая, в каком столбце и строке будет находиться элемент. Кроме того, если мы хотим растянуть элемент управления на несколько строк или столбцов, то можно указать свойства **Grid.ColumnSpan** и **Grid.RowSpan**, как в следующем примере:

```
<Button Grid.Column="0" Grid.Row="0" Content="Строка 0 Столбец 0" />
<Button Grid.Column="0" Grid.Row="1"
  Content="Объединение трех столбцов" Grid.ColumnSpan="3" />
<Button Grid.Column="2" Grid.Row="2" Content="Строка 2 Столбец 2" />
```

То есть у нас получится следующая картина:



## Установка размеров

Но если в предыдущем случае у нас строки и столбцы были равны друг другу, то теперь попробуем их настроить столбцы по ширине, а строки - по высоте. Есть несколько вариантов настройки размеров.

### Автоматические размеры

Здесь столбец или строка занимает то место, которое им нужно

```
<ColumnDefinition Width="Auto" />
<RowDefinition Height="Auto" />
```

### Абсолютные размеры

В данном случае высота и ширина указываются в единицах, независимых от устройства:

```
<ColumnDefinition Width="100" />
<RowDefinition Height="100" />
```

### Пропорциональные размеры.

Например, ниже задаются два столбца, второй из которых имеет ширину в четверть от ширины первого:

```
<ColumnDefinition Width="1*" />
<ColumnDefinition Width="0.25*" />
```

Если строка или столбец имеет высоту, равную \*, то данная строка или столбец будет занимать все оставшееся место. Если у нас есть несколько строк или столбцов, высота которых равна \*, то все доступное место делится поровну между всеми такими строками и столбцами. Использование коэффициентов (0.25\*) позволяет уменьшить или увеличить выделенное место на данный коэффициент. При этом все коэффициенты складываются (коэффициент \* аналогичен 1\*) и затем все пространство делится на сумму коэффициентов.

Например, если у нас три столбца:

```
<ColumnDefinition Width="*" />
<ColumnDefinition Width="0.5*" />
<ColumnDefinition Width="1.5*" />
```

В этом случае сумма коэффициентов равна  $1* + 0.5* + 1.5* = 3*$ . Если у нас *Grid* имеет ширину 300 единиц, то для коэффициент 1\* будет соответствовать пространству  $300 / 3 = 100$  единиц. Поэтому первый столбец будет иметь ширину в 100 единиц, второй -  $100 * 0.5 = 50$  единиц, а третий -  $100 * 1.5 = 150$  единиц.

Можно комбинировать все типы размеров. В этом случае от ширины/высоты грида отнимается ширина/высота столбцов/строк с абсолютными или автоматическими размерами, и затем оставшееся место распределяется между столбцами/строками с пропорциональными размерами:



### Контейнер *UniformGrid*

Контейнер *UniformGrid* аналогичен контейнеру *Grid*, только в этом случае все столбцы и строки одинакового размера и используется упрощенный синтаксис для их определения. Подробно рассмотрите самостоятельно.

### Контейнер *StackPanel*

Это более простой элемент компоновки. Он располагает все элементы в ряд либо по горизонтали, либо по вертикали в зависимости от ориентации. Используем свойство *Orientation*, если используется значение *Vertical*, то есть *StackPanel* создает вертикальный ряд, в который помещает все вложенные элементы сверху вниз. Мы также можем задать горизонтальный стек. Для этого нам надо указать свойство *Orientation="Horizontal"*.

Например, из кнопок сделаем флаг России:

```
<GroupBox Header="GroupBox" HorizontalAlignment="Left" Height="156">
  <StackPanel Orientation="Vertical">
    <Button Background="White" Height="25" ></Button>
    <Button Background="Blue" Height="25"></Button>
    <Button Background="Red" Height="25"></Button>
  </StackPanel>
</GroupBox>
```



### Контейнер *DockPanel*

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

```
<DockPanel LastChildFill="True">
  <Button DockPanel.Dock="Top" Background="AliceBlue" Content="Верхняя кнопка 1" />
  <Button DockPanel.Dock="Top" Background="AliceBlue" Content="Верхняя кнопка 2" />
  <Button DockPanel.Dock="Bottom" Background="BlanchedAlmond" Content="Нижняя кнопка" />
  <Button DockPanel.Dock="Left" Background="Aquamarine" Content="Левая кнопка1" />
  <Button DockPanel.Dock="Left" Background="Aquamarine" Content="Левая кнопка2" />
  <Button DockPanel.Dock="Right" Background="DarkGreen" Content="Правая кнопка" />
  <Button Background="LightGreen" Content="Центр" />
</DockPanel>
```



### Компонент *GridSplitter*

Элемент *GridSplitter* представляет собой некоторый разделитель между столбцами или строками, путем сдвига которого можно регулировать ширину столбцов и высоту

строк. Например, для таблицы 3\*3 добавим разделитель для изменения размера 1 и 2 столбца;

```
<GridSplitter Grid.Column="0" Grid.RowSpan="3" Width="3" Panel.ZIndex="1"></GridSplitter>
```

### Свойства компоновки элементов

Элементы WPF обладают набором свойств, которые помогают позиционировать данные элементы. Рассмотрим некоторые из этих свойств.

#### Ширина и высота

У элемента можно установить ширину с помощью свойства **Width** и высоту с помощью свойства **Height**. Хотя общая рекомендация состоит в том, что желательно избегать жестко заданных в коде ширины и высоты.

Также мы можем задать возможный диапазон ширины и высоты с помощью свойств **MinWidth/MaxWidth** и **MinHeight/MaxHeight**. И при растяжении или сжатии контейнеров элементы с данными заданными свойствами не будут выходить за пределы установленных значений.

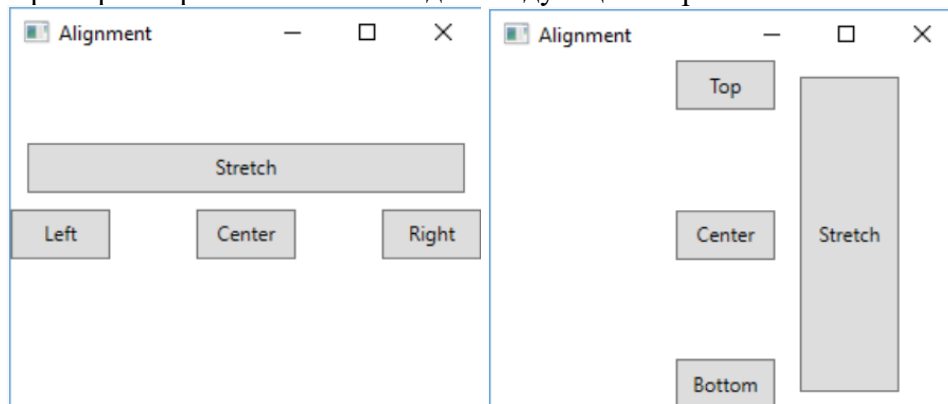
#### Выравнивание **HorizontalAlignment** и **VerticalAlignment**

С помощью специальных свойств мы можем выровнять элемент относительно определенной стороны контейнера по горизонтали или вертикали.

Свойство **HorizontalAlignment** выравнивает элемент по горизонтали относительно правой или левой стороны контейнера и соответственно может принимать значения **Left**, **Right**, **Center** (положение по центру), **Stretch** (растяжение по всей ширине).

Свойство **VerticalAlignment**, которое принимает следующие значения: **Top** (положение вверху контейнера), **Bottom** (положение внизу), **Center** (положение по центру), **Stretch** (растяжение по всей высоте).

Примеры выравнивания выглядят следующим образом:



#### Отступы **margin**

Свойство **Margin** устанавливает отступы вокруг элемента. Синтаксис: **Margin="левый\_отступ верхний\_отступ правый\_отступ нижний\_отступ"**. Если мы зададим свойство таким образом: **Margin="20"**, то сразу установим отступ для всех четырех сторон.

#### **Panel.ZIndex**

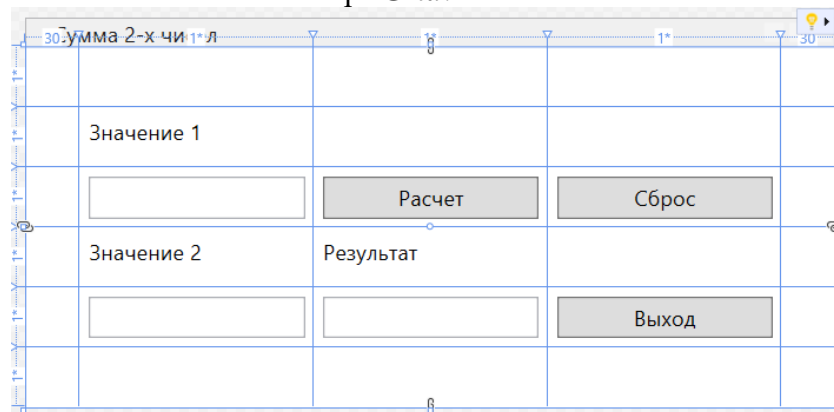
При создании интерфейса возможна ситуация, когда одни элементы будут полностью или частично перекрывать другие. По умолчанию те элементы, которые определены позже, перекрывают те элементы, которые определены ранее. Однако мы можем изменить подобное действие с помощью свойства **Panel.ZIndex**.

По умолчанию для всех создаваемых элементов **Panel.ZIndex="0"**. Однако назначив данному свойству более высокое значение, мы можем передвинуть его на передний план. Элементы с большим значением этого свойства будут перекрывать те элементы, у которых меньшее значение этого свойства.

## Примеры компоновки элементов

### Компоновка с применением контейнера Grid

Рассмотрим пример программы калькулятор, рассмотренный ранее с применением гибкой компоновки и контейнера *Grid*.

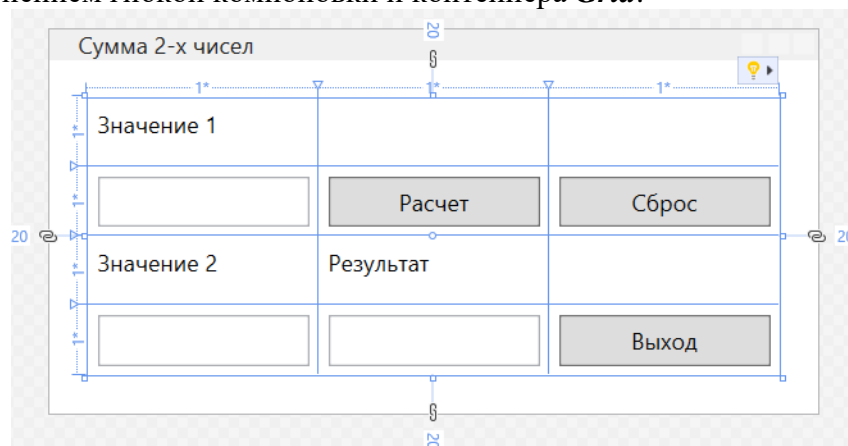


```

Title="Сумма 2-х чисел" Height="216" Width="450" Icon="/cd1
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="30" />
  </Grid.ColumnDefinitions>
  <Label Content="Значение 1" Grid.Row="1" Grid.Column="1" />
  <TextBox x:Name="tbZn1" Grid.Row="2" Grid.Column="1" Margin="5" M
  <Label Content="Значение 2" Grid.Row="3" Grid.Column="1" />
  <Label Content="Результат" Grid.Column="2" Grid.Row="3" />
  <TextBox x:Name="tbZn2" Grid.Row="4" Grid.Column="1" Margin="5" M
  <TextBox x:Name="tbRez" Grid.Column="2" Grid.Row="4" Margin="5" I
  <Button Content="Расчет" Grid.Column="2" Grid.Row="2" Margin="5"
  <Button Content="Выход" Grid.Column="3" Grid.Row="4" Margin="5" C
  <Button Content="Сброс" Grid.Column="3" Grid.Row="2" Margin="5" C
</Grid>

```

Рассмотрим еще пример программы калькулятор, рассмотренный ранее с применением гибкой компоновки и контейнера *Grid*.



```

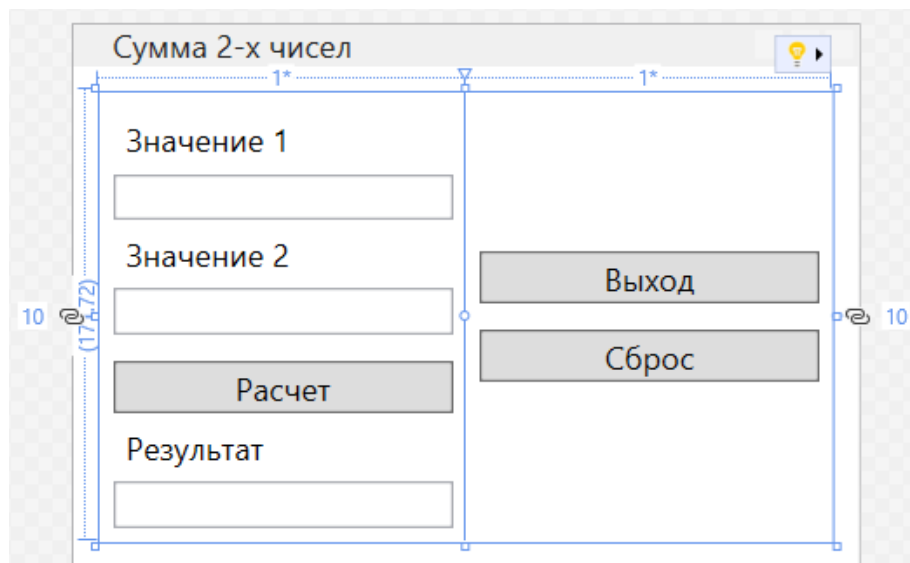
        Title="Сумма 2-х чисел" Height="200" Width="400" Icon="/cd1
<Grid Margin="20">
    <Grid.RowDefinitions>
        <RowDefinition Height="*/>
        <RowDefinition Height="*/>
        <RowDefinition Height="*/>
        <RowDefinition Height="*/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*/>
        <ColumnDefinition Width="*/>
        <ColumnDefinition Width="*/>
    </Grid.ColumnDefinitions>
    <Label Content="Значение 1" Grid.Row="0" Grid.Column="0"/>
    <TextBox x:Name="tbZn1" Grid.Row="1" Grid.Column="0" Margin="5" M
    <Label Content="Значение 2" Grid.Row="2" Grid.Column="0"/>
    <Label Content="Результат" Grid.Column="1" Grid.Row="2"/>
    <TextBox x:Name="tbZn2" Grid.Row="4" Grid.Column="0" Margin="5" M
    <TextBox x:Name="tbRez" Grid.Column="1" Grid.Row="4" Margin="5" I
    <Button Content="Расчет" Grid.Column="1" Grid.Row="1" Margin="5"
    <Button Content="Выход" Grid.Column="2" Grid.Row="3" Margin="5" C
    <Button Content="Сброс" Grid.Column="2" Grid.Row="1" Margin="5" C
</Grid>

```

### Смешанная компоновка с применением контейнеров Grid и StackPanel

Рассмотрим пример программы калькулятор, рассмотренный ранее с применением гибкой смешанной компоновки и применением контейнеров *Grid u StackPanel*.

В качестве основы используем контейнер *Grid* и в отдельных ячейках для последовательного размещения элементов используем вложенный контейнер *StackPanel*.





```
        Title="Сумма 2-х чисел" SizeToContent="Height" Width="300" Icon="/cd10
<Grid Margin="10">
    <Grid.RowDefinitions>
        <RowDefinition Height="auto"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*/>
        <ColumnDefinition Width="*/>
    </Grid.ColumnDefinitions>
    <StackPanel Grid.Column="0" Margin="5">
        <Label Content="Значение 1" />
        <TextBox x:Name="tbZn1" MaxLength="9" TextChanged="tbZn1_TextChanged"/>
        <Label Content="Значение 2"/>
        <TextBox x:Name="tbZn2" MaxLength="9" />
        <Button Content="Расчет" Margin="0,10,0,0" Click="btnCalc_Click" IsDefault
        <Label Content="Результат"/>
        <TextBox x:Name="tbRez" IsReadOnly="True"/>
    </StackPanel>
    <StackPanel Grid.Column="1" VerticalAlignment="Center">
        <Button Content="Выход" Margin="5" Click="btnExit_Click" />
        <Button Content="Сброс" Margin="5" Click="btnClear_Click" IsCancel="True
    </StackPanel>
</Grid>
```

Обратите внимание, так как мы использовали плавающий (резиновый) дизайн, то иногда можно не задавать реальный размер окна, а использовать свойство `SizeToContent="Height"` при котором вертикальный размер окна автоматически подстраивается под содержимое окна.

## Разработка масштабируемого интерфейса приложения

### Рекомендации при разработке масштабируемого интерфейса приложения

При разработке приложения для различных режимов разрешения или с изменяемыми размерами оконных форм, необходимо учитывать следующие рекомендации:

1. Заранее, в самом начале этапа разработки, решите для себя - собираетесь ли вы разрешать масштабировать окно или нет. Преимущество запрета масштабирования в том, что вам ничего не нужно менять во время выполнения приложения. Недостаток запрета масштабирования - во время выполнения приложения никаких изменений не происходит (ваша окно может быть слишком малой или слишком большой для работы в некоторых режимах при отсутствии масштабирования).
2. Если вы НЕ собираетесь масштабировать окно, установите свойство ***ResizeMode*** в ***CanMinimize*** в противном случае, установите свойство окна ***ResizeMode*** в ***CanResize***.
3. При необходимости установите свойство окна ***WindowStartupLocation*** во что-нибудь другое, чем ***Manual***. ***CenterScreen*** - окно находится в центре в текущем отображении и измерения. ***CenterOwner*** - окно находится в центре в пределах границ его родительской окна.
4. При необходимости используйте свойства ***MinWidth*** и ***MaxWidth***, который задают минимальный и максимальный размер окна при ее изменении.



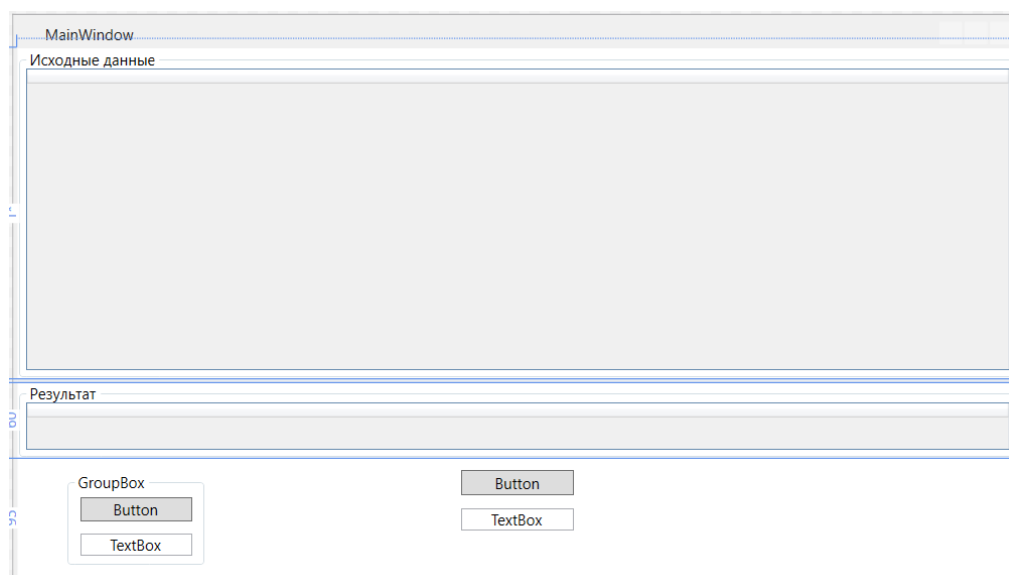
При создании масштабируемого приложения продумывают и реализуют поведение визуальных элементов при изменении размеров окна. Различают два основных вида поведения элемента при изменении размеров окна:

1. **Элемент имеет фиксированное положение и не изменяет размер**, т.е. при изменении размеров окна (родительского контейнера) он остается на первоначально заданном месте. Такой элемент помещают в ячейку **Grid** с фиксированным размером или он имеет абсолютное позиционирование.

2. **Элемент изменяет размер**, т.е. при изменении размеров окна (родительского контейнера) элемент соответственно изменяется в размерах. Такой элемент помещают в ячейку **Grid** с изменяемым размером, и он не имеет абсолютного позиционирования.

Как правило, чаще всего применяют контейнер **Grid**, хотя выше сказанное применимо и для других контейнеров.

Например, рассмотрим пример, в котором будет масштабируемая матрица, одномерный массив с фиксированной высотой, **GridSplitter** между первой и второй таблицей, демонстрирующий возможности элемента и область управляющих кнопок фиксированного размера, образец см. ниже:



```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="1*"/>
    <RowDefinition Height="3"/>
    <RowDefinition Height="60"/>
    <RowDefinition Height="95"/>
  </Grid.RowDefinitions>
  <GroupBox Header="Исходные данные" Grid.Row="0">
    <DataGrid/>
  </GroupBox>
  <GroupBox Header="Результат" Grid.Row="2">
    <Grid>
      <DataGrid/>
    </Grid>
  </GroupBox>
  <GroupBox Header="GroupBox" HorizontalAlignment="Left" Height="75" Margin="38,10.2,0,0" Grid.Row="3"
    VerticalAlignment="Top" Width="110">
    <StackPanel>
      <Button Content="Button" HorizontalAlignment="Stretch" Margin="5" />
      <TextBox HorizontalAlignment="Stretch" Text="TextBox" HorizontalContentAlignment="Center" Margin="5"/>
    </StackPanel>
  </GroupBox>
  <Button Content="Button" Width="90" Height="20" VerticalAlignment="Top" HorizontalAlignment="Left"
    Margin="352,10,0,0" Grid.Row="3" />
  <TextBox Text="TextBox" Width="90" VerticalAlignment="Top" HorizontalAlignment="Left"
    HorizontalContentAlignment="Center" Margin="352,40,0,0" Grid.Row="3" />
  <GridSplitter Grid.Row="1" VerticalAlignment="Center" HorizontalAlignment="Stretch" Height="3" />
</Grid>
```

## Практическая работа №12

### Создание интерфейса приложения.

1. Изучить формулировку задания.
2. Разработать спецификации модулей.
3. Составить паспорта модулей (функций) производящих вычисления по заданию.
4. Написать алгоритмы вычислительных модулей (функций) программы.

#### Практическое занятие в учебном классе

5. Разработать по паспорту модулей функции для решения задачи.
6. Спроектировать интерфейс программы для решения задач по варианту задания.
7. Для размещения двух заданий в одной программе использовать панель закладок.
8. Разбить окно на 2 группы: исходные данные, результат.
9. Предусмотреть в программе две кнопки «**Выход**» и «**О программе**».
10. Использовать меню.
11. Создать строку статуса 3 элемента: номер задачи, дата и время.
12. Использовать контекстное меню для блоков «**Исходные данные**» и «**Результат**».
13. При наличии кнопки «**Рассчитать**», она должна передавать фокус первому элементу блока «Исходные данные».
14. При изменении исходных данных очищать поле результата.
15. При отсутствии исходных данных при расчете вывести уведомление.
16. Оформить: заголовок, подсказки, иконки на окно и значок файла.
17. Обеспечить неизменяемость границ основного окна.
18. Разработать модульные тесты и провести тестирование модулей программы.
19. Оформить программу комментариями

#### Варианты заданий

1. Реализовать расчет задачи:
  - Дана сторона квадрата  $a$ . Найти его площадь и периметр.
  - Дано трехзначное число. Вывести вначале его последнюю цифру (единицы), а затем — его среднюю цифру (десятки).
2. Реализовать расчет задачи:
  - Даны стороны прямоугольника  $a$  и  $b$ . Найти его площадь и периметр.
  - Дано трехзначное число. Вывести число, полученное при прочтении исходного числа справа налево.
3. Реализовать расчет задачи:
  - Дан диаметр окружности  $D$ . Найти ее длину  $L = \pi \cdot D$ . В качестве значения  $\pi$  использовать 3.14.
  - Дано трехзначное число. В нем зачеркнули первую справа цифру и приписали ее слева. Вывести полученное число.
4. Реализовать расчет задачи:
  - Дана длина ребра куба  $A$ . Найти объем куба  $V$  и площадь его поверхности  $S$ .
  - Дана масса  $M$  в килограммах. Используя операцию деления целых чисел, найти количество полных тонн и килограмм (1 тонна = 1000 кг).
5. Реализовать расчет задачи:
  - Даны длины ребер  $a$ ,  $b$ ,  $c$  прямоугольного параллелепипеда. Найти его объем  $V = a \cdot b \cdot c$  и площадь поверхности  $S = 2 \cdot (a \cdot b + b \cdot c + a \cdot c)$ .
  - Дано двузначное число. Найти сумму и произведение его цифр.

6. Реализовать расчет задачи:
  - Даны два неотрицательных числа  $a$  и  $b$ . Найти их среднее геометрическое, то есть квадратный корень из их произведения.
  - Дано трехзначное число. Вывести число, полученное при перестановке цифр сотен и десятков исходного числа (например, 123 перейдет в 213).
7. Реализовать расчет задачи:
  - Даны катеты прямоугольного треугольника  $a$  и  $b$ . Найти его гипотенузу  $c$  и периметр  $P$ .
  - Дано целое число, большее 999. Используя одну операцию деления нацело и одну операцию взятия остатка от деления, найти цифру, соответствующую разряду сотен в записи этого числа.
8. Реализовать расчет задачи:
  - Даны два круга с общим центром и радиусами  $R_1$  и  $R_2$  ( $R_1 > R_2$ ). Найти площади этих кругов  $S_1$  и  $S_2$ , а также площадь  $S_3$  кольца, внешний радиус которого равен  $R_1$ , внутренний радиус равен  $R_2$ :  $S_1 = \pi \cdot (R_1)^2$ ,  $S_2 = \pi \cdot (R_2)^2$ ,  $S_3 = S_1 - S_2$ . В качестве значения  $\pi$  использовать 3.14.
  - Дан номер некоторого года (целое положительное число). Определить соответствующий ему номер столетия, учитывая, что, к примеру, началом 20 столетия был 1901 год.
9. Реализовать расчет задачи:
  - Дана площадь  $S$  круга. Найти его диаметр  $D$  и длину  $L$  окружности, ограничивающей этот круг, учитывая, что  $L = \pi \cdot D$ ,  $S = \pi \cdot D^2 / 4$ . В качестве значения  $\pi$  использовать 3.14.
  - С начала суток прошло  $N$  секунд ( $N$  — целое). Найти количество полных часов, прошедших с начала суток.
10. Реализовать расчет задачи:
  - Даны три точки  $A$ ,  $B$ ,  $C$  на числовой оси. Найти длины отрезков  $AC$  и  $BC$  и их сумму.
  - С начала суток прошло  $N$  секунд ( $N$  — целое). Найти количество полных минут, прошедших с начала последнего часа.
11. Реализовать расчет задачи:
  - Даны координаты двух противоположных вершин прямоугольника:  $(x_1, y_1)$ ,  $(x_2, y_2)$ . Стороны прямоугольника параллельны осям координат. Найти периметр и площадь данного прямоугольника.
  - Дан размер файла в байтах. Используя операцию деления нацело, найти количество полных килобайтов, которые занимает данный файл (1 килобайт = 1024 байта).
12. Реализовать расчет задачи:
  - Найти расстояние между двумя точками с заданными координатами  $(x_1, y_1)$  и  $(x_2, y_2)$  на плоскости. Расстояние вычисляется по формуле  $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ .
  - Дано трехзначное число. Используя одну операцию деления нацело, вывести первую цифру данного числа (сотни).
13. Реализовать расчет задачи:
  - Даны координаты трех вершин треугольника:  $(x_1, y_1)$ ,  $(x_2, y_2)$ ,  $(x_3, y_3)$ . Найти его периметр и площадь, используя формулу для расстояния между двумя точками на плоскости (см. задание 12). Для нахождения площади треугольника со сторонами  $a$ ,  $b$ ,  $c$  использовать формулу Герона:  $S = \sqrt{p(p-a)(p-b)(p-c)}$  где  $p = (a + b + c)/2$  — полупериметр.
  - Дано трехзначное число. Найти сумму и произведение его цифр.
14. Реализовать расчет задачи:

- Даны три точки А, В, С на числовой оси. Точка С расположена между точками А и В. Найти произведение длин отрезков АС и ВС.
  - Дано трехзначное число. Найти сумму и произведение его цифр.
15. Реализовать расчет задачи:
- Найти значение функции  $y = 3 \cdot x^6 - 6 \cdot x^2 - 7$  при данном значении  $x$ .
  - Даны целые положительные числа А и В ( $A > B$ ). На отрезке длины А размещено максимально возможное количество отрезков длины В (без наложений). Используя операцию деления нацело, найти количество отрезков В, размещенных на отрезке А.
16. Реализовать расчет задачи:
- Найти значение функции  $y = 4 \cdot (x-3)^6 - 7 \cdot (x-3)^3 + 2$  при данном значении  $x$ .
  - С начала суток прошло N секунд (N — целое). Найти количество секунд, прошедших с начала последней минуты.
17. Реализовать расчет задачи:
- Дано значение температуры Т в градусах Цельсия. Определить значение этой же температуры в градусах Фаренгейта. Температура по Цельсию ТС и температура по Фаренгейту ТF связаны следующим соотношением:  $TC = (TF - 32) \cdot 9/5$ .
  - Дано двузначное число. Вывести число, полученное при перестановке цифр исходного числа.
18. Реализовать расчет задачи:
- Известно, что X кг шоколадных конфет стоит А рублей, а Y кг ирисок стоит В рублей. Определить, сколько стоит 1 кг шоколадных конфет, 1 кг ирисок, а также во сколько раз шоколадные конфеты дороже ирисок.
  - Дано расстояние L в сантиметрах. Используя операцию деления целых чисел, найти количество полных метров и сантиметров (1 метр = 100 см).

## Практическая работа №13

### Создание масштабируемого интерфейса приложения.

1. Изучить формулировку задания.
2. Разработать спецификации модулей.
3. Составить паспорта модулей (функций) производящих вычисления по заданию.
4. Написать алгоритмы вычислительных модулей (функций) программы.

#### Практическое занятие в учебном классе

1. Разработать по паспорту модулей функции для решения задачи.
2. Спроектировать интерфейс программы для решения задач по варианту задания.
3. Подключить и использовать ранее разработанную библиотеку **libmas** для типовых функций (заполнить, сохранить и т.д.).
4. Разбить окно минимум на 2 группы: исходные данные, результат.
5. Обеспечить динамическое изменение размера таблиц в ходе выполнения программы.
6. Использовать меню.
7. Использовать контекстное меню для блоков «**Исходные данные**» и «**Результат**».
8. Реализовать сохранение и чтение значений из файла для таблицы.
9. При изменении исходных данных очищать поле результата.
10. При отсутствии исходных данных при расчете вывести уведомление.
11. Оформить: заголовок, подсказки, иконки на окно и значок файла.
12. Предусмотреть в программе две кнопки «**Выход**» и «**О программе**».
13. Обеспечить изменяемость границ основного окна.
14. Настроить элементы окна так, чтобы они корректно масштабировались при изменении размеров окна программы.
15. Использовать строку статуса, где отображать размер таблицы и номер выделенной ячейки.
16. Использовать панель инструментов.
17. Разработать модульные тесты и провести тестирование модулей программы.
18. Оформить программу комментариями

#### Варианты заданий

1. Дана матрица размера  $M * N$ . Найти количество ее столбцов, элементы которых упорядочены по убыванию.
2. Дана целочисленная матрица размера  $M * N$ . Найти номер первого из ее столбцов, содержащих максимальное количество одинаковых элементов.
3. Дана целочисленная матрица размера  $M * N$ . Найти номер последней из ее строк, содержащих максимальное количество одинаковых элементов.
4. Дана целочисленная матрица размера  $M * N$ . Найти количество ее столбцов, все элементы которых различны.
5. Дана целочисленная матрица размера  $M * N$ . Найти номер первого из ее столбцов, содержащих только нечетные числа. Если таких столбцов нет, то вывести 0.
6. Дана целочисленная матрица размера  $M * N$ . Найти номер последнего из ее столбцов, содержащих равное количество положительных и отрицательных элементов (нулевые элементы матрицы не учитываются). Если таких столбцов нет, то вывести 0.

7. Дана матрица размера  $M * N$ . Найти номера строки и столбца для элемента матрицы, наиболее близкого к среднему значению всех ее элементов.
8. Дана матрица размера  $M * N$ . В каждом ее столбце найти количество элементов, больших среднего арифметического всех элементов этого столбца
9. Дана матрица размера  $M * N$ . Найти минимальный среди максимальных элементов ее столбцов.
10. Дана матрица размера  $M * N$ . Найти номер ее столбца с наименьшим произведением элементов и вывести данный номер, а также значение наименьшего произведения.
11. Дана матрица размера  $M * N$ . Для каждого столбца матрицы с четным номером (2, 4, ...) найти сумму его элементов. Условный оператор не использовать.
12. Дана матрица размера  $M * N$  и целое число  $K$  ( $1 < K < N$ ). Найти сумму и произведение элементов  $K$ -го столбца данной матрицы.
13. Дана вещественная матрица  $A(M, N)$ . Строку, содержащий максимальный элемент, поменять местами со строкой, содержащей минимальный элемент.
14. По массиву  $A(5,6)$  получить массив  $B(6)$ , присвоив его  $j$ -элементу значение true, если все элементы  $j$ -столбца массива  $A$  нулевые, и значение false иначе.
15. Дана вещественная матрица  $A(N,M)$ . Составить программу замены всех отрицательных элементов матрицы на элемент, имеющий максимальное значение.
16. Дана вещественная матрица  $A(M, N)$ . Составить программу подсчета количества элементов матрицы, которые лежат вне интервала  $[c1, c2]$ .
17. Дана целая матрица  $A(N,N)$ . Составить программу подсчета количества нечетных элементов, расположенных выше побочной диагонали.
18. Дана вещественная матрица  $A(N,M)$ . Составить программу нахождения максимального отрицательного элемента матрицы и нахождения его местоположения.
19. Дана вещественная матрица  $A(N,M)$ . Составить программу замены всех нулевых элементов матрицы на минимальный элемент.

## Многооконные приложения. Диалоговые окна

### Общие положения

До сих пор мы с вами все приложения делали с одним единственным окном. А между тем, в современном программировании редко встречаются программы, имеющие только одно окно. Даже простые стандартные утилиты, вроде Калькулятора `calc.exe` или игры "Сапер" - `winmine.exe` имеют по несколько окон. В этой лекции мы с вами научимся делать многооконные приложения.

Имеется два типа интерфейсов: SDI (Single Document Interface - однодокументный интерфейс) и MDI (Multi Document Interface - многодокументный интерфейс). SDI-приложения работают одновременно с одним документом, MDI-приложения предназначены для одновременной работы с множеством однотипных документов. При этом все документы располагаются внутри одного контейнера, которым служит, как правило, главное окно. В нашей лекции подробно рассматривать MDI-интерфейсы мы не будем, а рассмотрим интерфейс - SDI.

В SDI-приложениях окна могут быть двух видов - **модальные** и **немодальные**. Создаются они одинаково, разница заключается только в способе вывода этих окон на экран. Модальное окно блокирует программу, не даёт с ней работать, пока вы это окно не закроете. Стандартный пример модального окна - окно "О программе", которое присутствует почти в любом приложении. Как правило, такое окно находится в меню "Справка". Пока вы не нажмете "ОК", закрыв это окно, вы не сможете работать с основной программой.

Немодальные окна позволяют переключаться между ними, и программой, и работать одновременно и там, и там. Типичный пример - окна Visual Studio - вы можете переключаться между окнами Редактор кода, Обозреватель решения, Свойства, и другими окнами - они не мешают друг другу, так как все они немодальные.

### Модальные окна на базе класса *MessageBox*

Простейшее модальное диалоговое окно можно создать на базе класса *MessageBox*, входящего в библиотеку *Microsoft .NET Framework*. У данного класса есть метод *Show()*, который создает простое модальное окно с сообщением. Это метод имеет 20 перегрузок, которые позволяют различных типов и спецификации. Рассмотрим наиболее типичные перегрузки метода *Show()*.

Вывод простого сообщения в модальном окне с кнопкой **ОК**

Прототип метода	Пример
<code>MessageBoxResult Show (string messageBoxText)</code>	<code>MessageBox.Show("Сообщение");</code>

Вывод простого сообщения с заголовком и кнопкой **ОК**

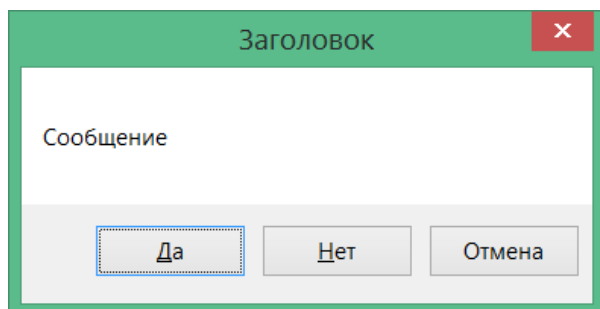
Прототип метода	Пример
<code>MessageBoxResult Show (string messageBoxText, string caption)</code>	<code>MessageBox.Show("Сообщение", "Заголовок");</code>

Вывод сообщения с заголовком и кнопками

Прототип метода	Пример
<code>MessageBoxResult Show (string messageBoxText, string caption, MessageBoxButton button)</code>	<code>MessageBoxResult result; result = MessageBox.Show("Сообщение", "Заголовок", MessageBoxButton.YesNoCancel);</code>



	<pre> if (result == DialogResult.Yes) /*Что-то делаем*/;  //При начале завершения работы с формой private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e) {     DialogResult result;     result = MessageBox.Show("Вы желаете завершить работу с программой", "Выход из программы", MessageBoxButtons.YesNo);     if (result == DialogResult.No)         e.Cancel = true; } </pre>
--	--



MessageBoxButton – перечисление

Имя	Описание
OK	Окно сообщения содержит кнопку ОК.
OKCancel	Окно сообщения содержит кнопки "ок" и "отмена".
YesNo	Окно сообщения содержит кнопки да и нет.
YesNoCancel	Окно сообщения содержит кнопки "да, нет и отмена.





MessageBoxResult - перечисление

Имя	Описание
Cancel	Возвращаемое значение диалогового окна Cancel
No	Возвращаемое значение диалогового окна No
OK	Окно сообщения содержит кнопку ОК.
Yes	Возвращаемое значение диалогового окна Yes
None	Окно сообщения не возвращает никаких результатов

Вывод сообщения с заголовком, кнопками и значком

Прототип метода	Пример
<pre> MessageBoxResult Show (string messageBoxText, string caption, MessageBoxButton button, MessageBoxImage icon) </pre>	<pre> MessageBoxResult result; result = MessageBox.Show("Не курите?", "Вопрос", MessageBoxButtons.YesNo, MessageBoxImage.Question); if (result == DialogResult.Yes)     /*Что-то делаем*/; </pre>

## MessageBoxImage – перечисление

Имя	Значок
Asterisk, Information	
Error, Stop	
Exclamation, Warning	
Question	
None	

## Вывод сообщения с заголовком, кнопками, значком и кнопкой по умолчанию

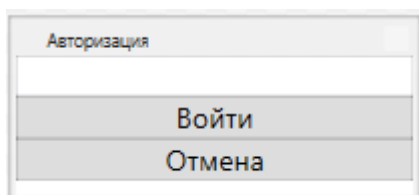
Прототип метода	Пример
<pre> MessageBoxResult Show (string messageBoxText, string caption, MessageBoxButton button, MessageBoxImage icon, MessageBoxResult defaultResult) </pre>	<pre> MessageBoxResult result; result = MessageBox.Show("Не курите?", "Вопрос", MessageBoxButton.YesNo, MessageBoxImage.None, MessageBoxResult.Yes); if (result == MessageBoxResult.Yes) /*Что-то делаем*/; </pre>

**Произвольное окно Window**

В Visual Studio можно создать произвольное окно для размещения необходимой информации или решения необходимых задач.

Добавить окно можно следующим способом. Выбрать в главном меню команду **Проект - Добавить окно WPF**.

Для примера создадим окно с паролем.



```

Title="Авторизация" WindowStartupLocation="CenterOwner" ResizeMode="NoResize"
FontSize="20" Height="126.96" Width="286.52" Activated="Window_Activated">
<StackPanel>
    <PasswordBox x:Name="txtPas" PasswordChar="*"></PasswordBox>
    <Button Content="Войти" IsDefault="True" Click="btnEnter_Click"></Button>
    <Button Content="Отмена" IsCancel="True" Click="btnEsc_Click"/>
</StackPanel>

```

Настроим окно для соответствия поведения окна условиям задачи.

Имя	Описание
<b>Элемент Window</b>	
WindowStartupLocation	CenterOwner – вывод по центру программы
ResizeMode	NoResize – нельзя менять размер окна
<b>Кнопки</b>	
IsDefault	При значении true, нажатие на Enter эквивалентно нажатию этой кнопки
CancelButton	При значении true, нажатие на Esc эквивалентно нажатию этой

	КНОПКИ
<b>Элемент PasswordBox</b>	
PasswordChar	Заменяет вводимые символы на указанный символ

Теперь следуя правилам нужно создать окно и вызвать ее в необходимом месте программы. Наиболее логично вызвать окно с паролем при загрузке главного окна или при инициализации окна, чтобы пользователь авторизовался до начала использования программы.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    Password pas = new Password();
    pas.Owner = this; //Получение ссылки на родителя
    pas.ShowDialog();
}
```

Обратите внимание, если окно открыто в модальном режиме, то выполнение кода в точке открытия окна приостанавливается до закрытия этой окна. При открытии окна в обычном режиме методом *Show*, работа программы не приостанавливается.

Описываем поведение окна. Пользователь в поле ввода должен вести пароль и нажать кнопку «Вход». Если введенный пароль правильный просто закрываем окно с паролем, иначе просим пользователя повторить ввод. При нажатии кнопки «Отмена» завершаем работу программы.

```
private void Window_Activated(object sender, EventArgs e)
{
    txtPas.Focus();
}
ссылка: 1 | 0 изменений | 0 авторов, 0 изменений
private void btnEnter_Click(object sender, RoutedEventArgs e)
{
    if (txtPas.Password == "123") Close();
    else
    {
        MessageBox.Show("Пароль неверен. Повторите ввод");
        txtPas.Focus();
    }
}
ссылка: 1 | 0 изменений | 0 авторов, 0 изменений
private void btnEsc_Click(object sender, RoutedEventArgs e)
{
    //Закрываем главную форму
    this.Owner.Close();
}
```

### **Передача параметров между окнами**

Язык C# является объектно-ориентированным, соответственно создание любого окна это по сути описание структуры класса данной окна. По умолчанию структура класса окна описывается таким образом, что передавать информацию между окнами нельзя. Соответственно для организации такой передачи используются определенные методы.

### **Используем отдельный класс**

Создаем отдельный класс, лучше статический, или как отдельный файл или в основном *namespace*.

Например, **MainWindow** – главное окно и на ней таблица, options – второе окно и на ней задаем размер таблицы и диапазон.

Описываем класс с данными *Data1*.

```
public static class data1 {  
    public static int ColumnCount;  
    public static int Diapazon;  
}
```

На первой форме вызываем вторую и после закрытия используем полученные данные

```
private void Настройки_Click(object sender, RoutedEventArgs e)  
{  
    //Заносим текущие значения в класс для отображения в окне настройки  
    data1.ColumnCount = Convert.ToInt32(columnCount.Text);  
    data1.Diapazon = Convert.ToInt32(diapazon.Text);  
    //Открываем окно настройки  
    Options opt = new Options();  
    opt.ShowDialog();  
    //Берем полученные значения и заносим их в соотв. элементы  
    columnCount.Text = data1.ColumnCount.ToString();  
    diapazon.Text = data1.Diapazon.ToString();  
}
```

На второй форме вводим нужные данные, в данном случае размер таблицы и жмем кнопку Задать, где информацию записываем в класс Data.

```
private void btnSet_Click(object sender, RoutedEventArgs e)  
{  
    int value;  
    if (Int32.TryParse(txtColumn.Text, out value))  
        data1.ColumnCount = value;  
    else  
    {  
        MessageBox.Show("Ошибка в номере столбца");  
        txtColumn.Focus();  
        return;  
    }  
    if (Int32.TryParse(txtDiapazon.Text, out value))  
        data1.Diapazon = value;  
    else  
    {  
        MessageBox.Show("Ошибка в диапазоне");  
        txtDiapazon.Focus();  
        return;  
    }  
    Close();  
}
```

При активации формы информацию с главной формы переносим на форму настройки.

```
private void Window_Activated(object sender, EventArgs e)
{
    txtColumn.Focus();
    txtColumn.Text = data1.ColumnCount.ToString();
    txtDiapazon.Text = data1.Diapazon.ToString();
}
```

## Концепция ресурсов в WPF

Для совместного использования одних и тех же компонентов различными элементами **WPF** применяет концепцию ресурсов. В данном случае под ресурсами понимаются не вспомогательные файлы - изображений и т.д., которые используются в приложении, а логические ресурсы, которые определяются в коде **C#** или **XAML**.

В качестве ресурса можно определить любой объект. Все ресурсы помещаются в объект **ResourceDictionary**. У каждого визуального объекта, например, **Windows** или **Button**, имеется свойство **Resources**, которое как раз хранит объект **ResourceDictionary**. Например, определим несколько ресурсов:

```
<Window.Resources>
    <SolidColorBrush x:Key="textColor">■#004D40</SolidColorBrush>
    <SolidColorBrush x:Key="backColor">■#80CBC4</SolidColorBrush>
    <sys:Double x:Key="fontSize">24</sys:Double>
</Window.Resources>
<Grid>
    <Menu VerticalAlignment="Top">
        <MenuItem Header="Файл">
            <MenuItem Header="Выход" Click="Выход_Click"
                Foreground="■{StaticResource textColor}"
                Background="■{StaticResource ResourceKey=backColor}"
                FontSize="{StaticResource ResourceKey=fontSize}"
            </MenuItem>
        </MenuItem>
    </Menu>
</Grid>
```

Каждый ресурс должен иметь ключ, задаваемый с помощью атрибута **x:Key**. Это своего рода уникальный идентификатор ресурса.

Чтобы обратиться к этому ресурсу в коде, надо использовать расширение **StaticResource**. Свойство **ResourceKey** через ключ ресурса будет ссылаться на данный ресурс. Обратите внимание свойство **ResourceKey** можно опустить.

## Уровни ресурсов

Ресурсы могут определяться на трех уровнях:

- На уровне отдельного элемента управления. Такие ресурсы могут применяться ко всем вложенным элементам, которые определены внутри этого элемента
- На уровне окна. Такие ресурсы могут применяться ко всем элементам в данном окне
- На уровне всего приложения. Эти ресурсы доступны из любого места и из любого окна приложения.

Когда парсер *XAML* встречает в разметке расширения *StaticResource* или *DynamicResource*, он сначала ищет соответствующий ресурс по ключу в словаре ресурсов текущего элемента. Если в ресурсах элемента отсутствует ресурс с указанным ключом, то парсер поднимается выше по дереву элементов и ищет ресурс в ресурсах элементов-контейнеров, затем в ресурсах текущего окна и в конце в ресурсах приложения. Поиск прекращается, когда найден первый попавшийся ресурс, который соответствует ключу. Если парсер нигде не смог найти ресурс, то он генерирует исключение *XamlParseException*.

Выше в примере ресурсы определялись на уровне текущего окна.

### Ресурсы приложения

Для определения общих для всего приложения ресурсов в проекте присутствует файл App.xaml. По умолчанию файл App.xaml уже содержит заготовку для размещения ресурсов:

```
<Application x:Class="WpfApp1.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WpfApp1"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

### Статические и динамические ресурсы

Ресурсы могут быть статическими и динамическими. Статические ресурсы устанавливаются только один раз и значения статического ресурса изменить нельзя. А динамические ресурсы могут меняться в течение работы программы. Для этого вместо *StaticResource* используется расширение *DynamicResource*.

Рассмотрим пример использования статических и динамических ресурсов. Определим ресурсы:

```
<Window.Resources>
    <SolidColorBrush x:Key="textColor" Color="#004D40"/>
    <SolidColorBrush x:Key="backColor" Color="#80CBC4"/>
    <sys:Double x:Key="fontSize">24</sys:Double>
</Window.Resources>
```

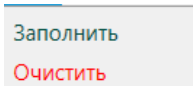
Один и тот же ресурс можно использовать как статический, так и динамический:

```
<MenuItem Header="Заполнить" Foreground="{StaticResource textColor}">
<MenuItem Header="Очистить" Foreground="{DynamicResource textColor}">
```

Изменяем ресурс в программе:

```
this.Resources["textColor"] = new SolidColorBrush(Colors.Red);
```

Свойства заданные статическим ресурсом не меняются. Свойства заданные динамическим ресурсом меняются.



## Стили

Стили позволяют определить набор некоторых свойств и их значений, которые потом могут применяться к элементам. Основная их задача - создать стилевое единообразие для элементов интерфейса. Стили хранятся в ресурсах и отделяют значения свойств элементов от пользовательского интерфейса.

Чтобы понять, как стили упрощают нам работу, рассмотрим простой пример:

```
<Application.Resources>
  <Style x:Key="buttonStyle" TargetType="Button">
    <Setter Property="Foreground" Value="#004D40" />
    <Setter Property="Background" Value="#80CBC4" />
    <Setter Property="FontSize" Value="20" />
  </Style>
</Application.Resources>
```

Стиль создается как ресурс с помощью объекта *Style* и, как любой другой ресурс, он обязательно должен иметь ключ. Атрибут *TargetType* указывает, к какому типу относится стиль. В данном случае это тип *Button*.

С помощью коллекции *Setters* определяется группа свойств, входящих в стиль. В нее входят объекты *Setter*, которые имеют следующие свойства:

- **Property:** указывает на свойство, к которому будет применяться данный сеттер. При этом свойство должно представлять тип *BindableProperty*
- **Value:** собственно значение свойства

Поскольку стиль определяется как ресурс, то для его установки используются расширения *StaticResource* или *DynamicResource*:

```
<Button Content="Выход" Style="{StaticResource buttonStyle}" />
```

## TargetType

Если нам надо создать общий стиль для элементов определенного типа, то можно не задавать ключ ресурса, а достаточно установить у стиля атрибут *TargetType*, в который передается тип элементов:

```
<Application.Resources>
  <Style TargetType="Button">
    <Setter Property="Foreground" Value="#004D40" />
    <Setter Property="Background" Value="#80CBC4" />
    <Setter Property="FontSize" Value="20" />
  </Style>
</Application.Resources>
```

Теперь у кнопок не надо будет указывать ресурс стиля, так как стиль будет автоматически применяться ко всем объектам типа, который указан в атрибуте *TargetType*.

## Переопределение стилей

Стиль позволяет задать некоторые начальные значения. Однако элемент может переопределить отдельные значения из стиля:

```
<StackLayout>
  <Button Text="Кнопка" Style="{StaticResource Key=buttonStyle}"
    TextColor="Red" />
</StackLayout>
```



В данном случае кнопка получает все значения из стиля **buttonStyle**, однако переопределяет цвет текста, так как прямое использование атрибутов элемента имеет приоритет над применяемым стилем.

### Триггеры

Триггеры в **WPF** позволяют декларативно задать некоторые действия, которые выполняются при изменении свойств визуального объекта, изменении данных или генерации событий. Триггеры определяются с помощью коллекции **Triggers** на уровне отдельного элемента, либо в виде ресурса на уровне страницы или приложения.

### Триггеры свойств

Простые триггеры свойств определяются как элементы стиля с помощью объекта **Trigger**. Они следят за значением свойств и в случае их изменения с помощью объекта **Setter** устанавливают значение других свойств.

Например, пусть по переходу к текстовому полю **TextBox** текст в нем приобретает красный цвет:

```
<Style x:Key="textBoxStyle" TargetType="TextBox">
  <Setter Property="Foreground" Value="Black" />
  <Style.Triggers>
    <Trigger Property="IsFocused" Value="True">
      <Setter Property="Foreground" Value="Red" />
    </Trigger>
  </Style.Triggers>
</Style>
<TextBox x:Name="columnCount" Style="{StaticResource textBoxStyle}"
```

Стиль может содержать несколько триггеров, и все они определяются в элементе **Style.Triggers**.

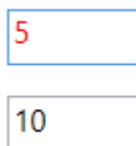
У каждого триггера устанавливаются два свойства:

- **Property:** свойство, на изменение которого должен отслеживать триггер
- **Value:** значение свойства, при котором должен срабатывать триггер

То есть триггер в данном случае будет срабатывать, когда свойство **IsFocused** элемента **TextBox** приобретёт значение **true**.

Работа триггера будет заключаться в установке ряда свойств элемента **TextBox**. Здесь у **TextBox** устанавливается красный текст. Таким образом, при получении фокуса сработает триггер, который окрасит текст в красный цвет.

Зато, когда мы установим фокус в другом месте приложения на какой-то другой элемент, то триггер уже действовать не будет, а текст в **TextBox** приобретет свой стандартный цвет:





## Пример применения стилей

```
<Application.Resources>
  <SolidColorBrush x:Key="MainColor" Color="#3E60C1" />
  <SolidColorBrush x:Key="BackColor" Color="#ddFFFFFF" />
  <Style TargetType="Button">
    <Setter Property="Background" Value="{DynamicResource MainColor}" />
    <Setter Property="BorderBrush" Value="{DynamicResource MainColor}" />
    <Setter Property="Foreground" Value="White" />
    <Setter Property="BorderThickness" Value="2" />
    <Setter Property="FontSize" Value="16" />
    <Setter Property="Margin" Value="0,6" />
    <Setter Property="FontWeight" Value="SemiBold" />
  </Style>
  <Style TargetType="TextBox">
    <Setter Property="Background" Value="#bbffffff" />
    <Setter Property="BorderBrush" Value="{StaticResource MainColor}" />
    <Setter Property="HorizontalAlignment" Value="Left" />
    <Setter Property="Margin" Value="0,5" />
    <Setter Property="MinWidth" Value="300" />
    <Setter Property="BorderThickness" Value="2" />
    <Setter Property="VerticalContentAlignment" Value="Center" />
    <Setter Property="FontSize" Value="16" />
    <Setter Property="MinHeight" Value="37" />
  </Style>
```

## **Практическая работа №14**

### **Многооконные приложения.**

Самостоятельная работа в домашних условиях

1. Изучить формулировку задания.
2. Изучить методы организации многооконного интерфейса и способы передачи информации между окнами программы.

Практическое занятие в учебном классе

1. Создать копию программы из задания №13
2. Добавить окно «Пароль» и организовать авторизацию в программе. Для упрощения проверки пароль задать «123».
3. Создать окно «Настройка». Назначение окна – изменение размера таблицы. При установке размера таблицы сохранять настройки в файл конфигурации «config.ini».
4. При запуске программы задавать размер таблицы согласно файлу конфигурации.
5. Использовать исключения. Например: при чтении файла конфигурации если он отсутствует размер таблицы не задавать.
6. При закрытии программы вывести диалоговое окно с подтверждением завершения работы.
7. Применить стили для оформления программы.
8. Оформить программу комментариями.

## Контроль ввода данных. Исключения

### Введение

В реальных приложениях обеспечение корректности вводимых данных является обязательным условием. Особой тщательности требует организация ввода больших объёмов данных. Для обеспечения контроля корректности ввода или обработки данных используются различные подходы.

### Ограничение ввода

В любом случае желательно использовать такие компоненты, которые уменьшают количество ошибок. Если есть возможность организовать выбор значений из списка, то предпочтение следует отдать компонентам *ListBox*, *ComboBox* и их разновидностям.

К сожалению, полностью отказаться от набора данных на клавиатуре, как правило, не удаётся. Если известны ограничения на вводимые значения. Например, для возраста и стажа работника, скорости движения поезда, грузоподъёмности пассажирского лифта и др. можно указать минимальное и максимальное значения, для шифра – количество символов. Часто в формулировки задачи допустимые значения указываются явно и, используя свойство *MaxLength*, можно задать ограничения на значения.

Можно контролировать ввод символов в поле ввода. При вводе символа проверяем его на соответствие заданному критерию и, если он не соответствует этому критерию, заменяем этот символ на пустой, либо прерываем ввод символа.

### Пример 1. Ограничение ввода символов в поле TextBox.

Используем событие *KeyDown*, которое возникает при нажатии клавиши в элементе *TextBox*. У этого события есть параметр *KeyEventArgs e*, который имеет свойство *Key*, возвращающее нажатую клавишу.

Используя свойство *Key* можно проверить нажатую клавишу и, если она удовлетворяет некоторым критериям продолжить ее получение, или в противном случае отменить ее получение.

```
// Ввод только цифровых данных
ссылка: 1
private void textBox1_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Key >= Key.D0 && e.Key <= Key.D9 || e.Key == Key.Back);
    else
    {
        MessageBox.Show("Введите цифровые данные");
        e.Handled = true;
    }
}
```

**ЗАМЕЧАНИЕ:** использовать такой подход нужно крайне осторожно, т.к. пользователь с низкой квалификацией может не понимать, что и как нужно вводить в такое поле, а невозможность ввести данные может поставить его в «ступор». Для большего понимания такой ситуации можно при вводе ошибочного символа выводить сообщение.

Используем событие *PreviewTextInput*, которое возникает при вводе текста в элементе *TextBox*. У этого события есть параметр *TextCompositionEventArgs e*, который имеет свойство *Text*, возвращающее введенный текст.

Используя свойство *Text*, можно проверить введенный текст и если он удовлетворяет некоторым критериям, продолжить его получение, или в противном случае отменить его получение.

```
private void textBox1_PreviewTextInput(object sender, TextCompositionEventArgs e)
{
    Char val = Convert.ToChar(e.Text);
    if (Char.IsDigit(val));
    else
    {
        MessageBox.Show("Введите цифровые данные");
        e.Handled = true; // отклоняем ввод
    }
}
```

### Контроль введенных параметров

Пользователю можно разрешить свободный ввод информации в поле ввода, тем самым не затрудняя его работу с программой. Правильность ввода данных можно контролировать либо при использовании этих данных, либо завершении ввода данных в поле ввода.

### Пример 2. Проверка правильности числовых данных в поле TextBox, при использовании этих данных.

Для проверки правильности ввода числовых данных в текстовое поле можно использовать функцию *Int32.TryParse*. Если исходные данные корректны, то производим вычисления, в противном случае сообщает об ошибке.

```
public static bool TryParse(string s, out int result)
```

#### Параметры

S - Тип: [System.String](#)

Строка, содержащая преобразуемое число.

Result - Тип: [System.Int32](#)

При возвращении этим методом содержит 32-разрядное целочисленное значение со знаком, эквивалентное числу, содержащемуся в параметре s, если преобразование выполнено успешно, или нуль, если оно завершилось сбоем.

#### Возвращаемое значение

Тип: [System.Boolean](#)

Значение true, если параметр s успешно преобразован; в противном случае — значение false.

```

//Контроль корректности исходных данных
ссылка 1
private void button1_Click(object sender, EventArgs e)
{
    int x1; bool f1;
    //Берем целое значение из TextBox и конвертируем число x1
    //При успешной конвертации f1=true иначе f1=false
    f1 = Int32.TryParse(textBox3.Text, out x1);
    if (f1 == true) x1 = x1 * 2; //Расчет
    else
    {
        MessageBox.Show("Исходные данные неверны");
        textBox3.Focus(); //Фокус в поле ввода
    }
}

```

**Пример 3. Контроль значений в поле TextBox, при передаче фокуса из этого поля. Если значение в поле после изменения не является корректным, то возвращаем прежнее значение.**

Используем событие, *GotFocus* - которое возникает при получении фокуса элементом. При этом событии в глобальной переменной запоминаем исходное значение в элементе *TextBox*.

Используем событие, *LostFocus* - которое возникает при потере фокуса элементом. Проверяем корректность значения в элементе *TextBox*. Если значение в поле не является корректным, то из глобальной переменной восстанавливаем корректное значения в поле *TextBox*.

```

string value;
Ссылка: 0
private void textBox2_GotFocus(object sender, RoutedEventArgs e)
{
    value = textBox2.Text; //Запоминаем исходное значение
}
ссылка: 1
private void textBox2_LostFocus(object sender, RoutedEventArgs e)
{
    int x1; bool f1;
    //Проверяем значение
    f1 = Int32.TryParse(textBox2.Text, out x1);
    //Если значение неверное возвращаем старое значение
    if (f1 == false)
    {
        MessageBox.Show("Исходные данные неверны");
        textBox2.Text = value;
    }
}

```

Если таких текстовых полей много, чтобы каждому из них не делать одинаковые обработчики событий, можно сделать один универсальный обработчик для всех элементов.

У событий есть параметр **object sender**, который является ссылкой на объект, который вызвал это событие. По ссылке **object sender** можно получить доступ к элементу **TextBox**, который вызвал это событие и провести для него проверку.

```
private void textBox2_GotFocus(object sender, RoutedEventArgs e)
{
    value = ((TextBox)(sender)).Text; //Запоминаем исходное значение
}
Ссылка: 2
private void textBox2_LostFocus(object sender, RoutedEventArgs e)
{
    int x1; bool f1;
    //Проверяем значение
    f1 = Int32.TryParse(((TextBox)(sender)).Text, out x1);
    //Если значение неверное возвращаем старое значение
    if (f1 == false)
    {
        MessageBox.Show("Исходные данные неверны");
        ((TextBox)(sender)).Text = value;
    }
}
```

#### Пример 4. Контроль значений в ячейках таблицы **DataGridView**.

Используем событие, **BeginningEdit** - которое возникает, перед началом редактирования ячейки. При этом событии в глобальной переменной запоминаем исходное значение ячейки таблицы.

Используем и событие **CellEditEnding**, которое возникает, после завершения редактирования ячейки таблицы. В этом событии проверяем корректность значения в ячейке таблицы. Если значение не является корректным, то из глобальной переменной восстанавливаем корректное значения в ячейку таблицы или подсвечиваем ее.

```
string cell;
Ссылка: 1
private void dataGrid_BeginningEdit(object sender, DataGridBeginningEditEventArgs e)
{
    //Определяем номер столбца
    int columnIndex = e.Column.DisplayIndex;
    //Определяем номер строки
    int indexRow = e.Row.GetIndex();
    // Получаем текущую строку
    DataRowView row = (DataRowView)dataGrid.Items[indexRow];
    // Запоминаем полученное значение
    cell = row[columnIndex].ToString();
}
```

```
private void dataGrid_CellEditEnding(object sender, DataGridCellEditEndingEventArgs e)
{
    //Определяем номер столбца
    int indexColumn = e.Column.DisplayIndex;
    //Определяем номер строки
    int indexRow = e.Row.GetIndex();
    int value;
    // Заносим полученное значение в массив
    if (Int32.TryParse(((TextBox)e.EditingElement).Text, out value)) mas[indexColumn] = value;
    else
    {
        MessageBox.Show("Ошибка");
        ((TextBox)e.EditingElement).Text = cell;
    }
    /* //Определяем номер столбца
    int indexColumn = dataGrid.CurrentCell.Column.DisplayIndex;
    //Определяем номер строки
    int indexRow = dataGrid.SelectedIndex;
    // Получаем текущую строку
    DataRowView row = (DataRowView)dataGrid.Items[indexRow];
    DataRowView row = (DataRowView)dataGrid.SelectedItems[0];
    DataRowView row = (DataRowView)dataGrid.CurrentCell.Item;
    // Заносим полученное значение в массив
    mas[indexColumn] = Convert.ToInt32(row[indexColumn].ToString());
    mas[indexColumn] = Convert.ToInt32(row["col" + (indexColumn + 1)].ToString());*/
}
```

## **Исключения**

### ***Как возникают исключительные ситуации?***

При работе программы могут возникать различного рода ошибки, которые являются следствием неправильной работы инструкций, процедур, функций или методов программы, а также операционной системы. Например: переполнение, деление на ноль, попытка открыть несуществующий файл, ошибка конвертации данных и т.д.

Программист должен предвидеть возможность возникновения ошибок и предусматривать их обработку. Для обработки таких ошибок введено понятие исключения, которое представляет собой нарушение условий выполнения программы, вызывающее прерывание или полное прекращение ее работы. Обработка исключения состоит в нейтрализации вызвавшей его динамической ошибки.

Исключение – это объект специального вида, характеризующий возникшую в программе исключительную ситуацию. Он может также содержать в виде параметров некоторую уточняющую информацию.

Если исключение не перехвачено нигде в программе, то оно обрабатывается стандартным методом, который обеспечивает стандартную реакцию программы на большинство исключений – выдачу пользователю краткой информации в окне сообщений.

Например, введено неправильное целое значение в поле ввода и при преобразовании *Convert.ToInt32* выдается ошибка

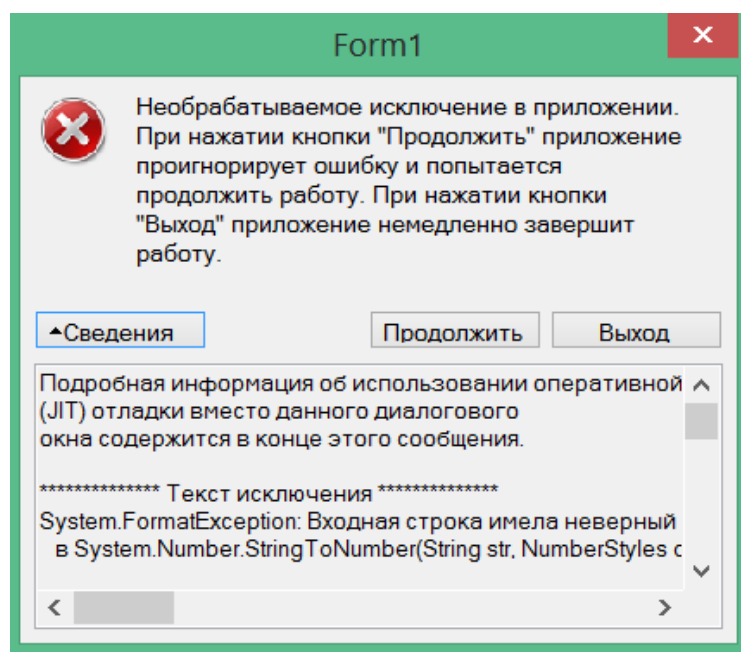


рис. 4. Стандартные сообщения об ошибках: неправильное целое значение

Если вы работаете в среде разработки Visual Studio, то исключения перехватываются отладчиком и сообщений выводятся в среде отладки

```
private void button2_Click(object sender, EventArgs e)
{
    int x1;
    //Берем целое значение из TextBox и конвертируем число x1
    x1=Convert.ToInt32(textBox1.Text);
    x1 = x1 * 2; //Расчет
```

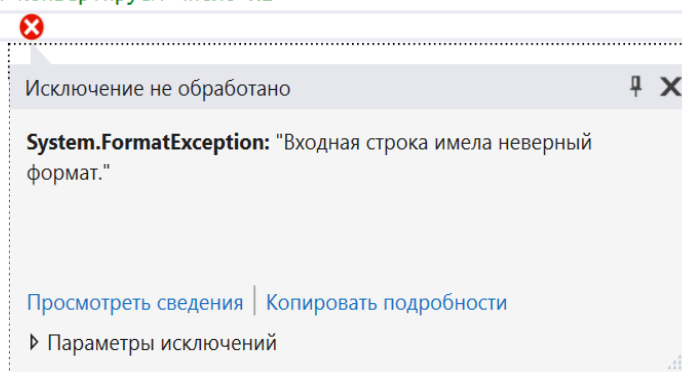


рис. 5. Ошибка конвертирования в целое значение

### Обработка исключений

Для обработки исключений в приложении есть один глобальный обработчик и несколько специализированных процедур-обработчиков, реагирующих на соответствующие исключения. Каждое исключение обрабатывается своим локальным обработчиком. Исключение, не имеющее своего локального обработчика, обрабатывается глобальным обработчиком приложения.

Один из способов борьбы с исключениями – обработка их с помощью блоков *Try ... Catch*. Синтаксис этих блоков следующий:



```

Try
{
    Исполняемый код программы, в котором могут возникнуть ошибки выполнения
}
Catch
{
    Код выполняемый только в случае генерации исключения
}

```

В блоке **Try** располагают операторы, при выполнении которых могут возникнуть исключения. В блоке **Catch** располагают операторы, которые выполняются при возникновении исключения. Если же исключение не возникло, то инструкции блока **Catch** не выполняются. При появлении исключения инструкции блока **Catch** могут ликвидировать исключительную ситуацию и восстановить работоспособность программы. Для исключений, обрабатываемых в конструкции **try .. Catch** глобальный обработчик не вызывается, а обработку ошибок должен обеспечить программист.

Рассмотрим пример контроля значений при получении иннокции из текстового поля. В данном случае мы используем конструкцию **try .. Catch**.

```

private void button2_Click(object sender, EventArgs e)
{
    int x1;
    //Берем целое значение из TextBox и конвертируем число x1
    //x1=Convert.ToInt32(textBox1.Text);
    //x1 = x1 * 2;//Расчет
    //Исключение
    try{
        x1=Convert.ToInt32(textBox1.Text);
        x1 = x1 * 2;//Расчет
    }
    catch
    {
        MessageBox.Show("Исходные данные неверны");
        textBox1.Focus();
    }
}

```

Наиболее ценным является то, что в блоке **Catch** можно определить тип сгенерированного исключения и избирательно реагировать на различные исключительные ситуации.

В одном и том же операторе **try-catch** можно использовать несколько предложений **catch**. В этом случае будет иметь значение порядок следования предложений **catch**, поскольку предложения **catch** будут проверяться именно в этом порядке. Более общие исключения следует перехватывать после более частных.

### Примеры стандартных классов исключений

Класс	Обрабатываемое исключение
<b>ArgumentException</b>	Это исключение выбрасывается, если один из передаваемых методу аргументов является недопустимым.
<b>ArgumentNullException</b>	Исключение, которое выбрасывается, при передаче указателя NULL (Nothing в Visual Basic) методу, который

	не принимает его как допустимый аргумент.
<b>ArgumentOutOfRangeException</b>	Исключение, которое выбрасывается, когда значение аргумента находится вне допустимого диапазона значений, как определено вызываемым методом.
<b>ArithmeticException</b>	Исключение, которое выбрасывается для ошибок арифметических действий, а также операций приведения к типу и преобразования.
<b>DivideByZeroException</b>	Исключение, выбрасываемое при попытке деления целого или дробного числа на нуль.
<b>NotFiniteNumberException</b>	Исключение, которое выбрасывается, когда значение с плавающей запятой является плюс бесконечностью, минус бесконечностью или не является числовым (NaN).
<b>OverflowException</b>	Исключение, которое выбрасывается, когда при выполнении арифметических операций, операций приведения типов и преобразования происходит переполнение.
<b>IndexOutOfRangeException</b>	Исключение, которое выбрасывается при попытке обращения к элементу массива с индексом, который находится вне границ массива. Этот класс не наследуется.
<b>ArrayTypeMismatchException</b>	Исключение, которое выбрасывается при попытке сохранить в массиве элемент неправильного типа.
<b>FormatException</b>	Исключение, выбрасываемое, если аргумент не соответствует спецификациям параметра вызываемого метода.
<b>OutOfMemoryException</b>	Исключение, которое выбрасывается при недостаточном объеме памяти для выполнения программы.

Рассмотрим пример контроля значений при получении информации из текстового поля и контроль ошибки деления на нуль

```
private void button3_Click(object sender, EventArgs e)
{
    int x1,x2;
    try
    {
        x1 = Convert.ToInt32(textBox4.Text);
        x2 = Convert.ToInt32(textBox5.Text);
        x1 = x1 / x2; //Расчет
    }
    catch (FormatException ex) //Ошибка преобразования
    {
        MessageBox.Show("Исходные данные неверны");
        textBox4.Focus();
    }
    catch (DivideByZeroException ex) //Ошибка деления на нуль
    {
        MessageBox.Show("Ошибка деления на Нуль");
        textBox5.Focus();
    }
}
```

## ***Практическая работа №15***

### **Контроль правильности исходных данных.**

Самостоятельная работа в домашних условиях

1. Изучить формулировку задания.
2. Изучить методы контроля вводимых данных.

Практическое занятие в учебном классе

1. Изучить программы, созданные в заданиях 1-3.
2. Обеспечить в заданиях 1-3 контроль корректности ввода исходных данных.