

При обычной привязке данные от источника отображаются без каких-либо изменений. Однако, иногда требуется преобразовать отображаемые данные. Это позволяют делать следующие средства, доступные в WPF:

- *Форматирование строк.* Позволяет преобразовать текстовые данные;
- *Конвертеры значений.* Позволяют преобразовывать любой тип исходных данных в любой тип представления объекта.

Возьмём пример из прошлой лабораторной работы, и зададим формат вывода значения цены так, чтобы всегда отображалось 2 знака после точки (сотые доли)

```
<TextBox Text="{Binding Path=Price.Value, UpdateSourceTrigger=PropertyChanged, StringFormat={}{0:F2}}"/>
```

Выражение для форматирования строится аналогично методу String.Format, используемому для форматирования строк в коде C# (с одной поправкой: если выражение начинается с фигурной скобки, нужно в начале поставить ещё одну пару фигурных скобок {}, как в данном примере). Документация по методу String.Format:

[https://msdn.microsoft.com/ru-ru/library/system.string.format\(v=vs.110\).aspx](https://msdn.microsoft.com/ru-ru/library/system.string.format(v=vs.110).aspx)

Конвертеры

Для более сложных преобразований используются конвертеры значений.

Модель данных из прошлой работы содержала булевское свойство Availability. Изменим отображение этого свойства так, чтобы выводился текст «Есть в наличии» или «Нет в наличии». Для этого потребуется написать конвертер, преобразующий булевское значение в строку «есть» или «нет». Свойство Availability будет доступно только для чтения.

```
using System;
using System.Globalization;
using System.Windows.Data;

namespace Tpu.Aics.Converters
{
    [ValueConversion(typeof(bool), typeof(string))]
    public class SplitStringConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter,
            CultureInfo culture)
        {
            bool booleanValue = (bool)value;

            return booleanValue ? "Есть" : "Нет";
        }

        public object ConvertBack(object value, Type targetType, object parameter,
            CultureInfo culture)
        {
            return null;
        }
    }
}
```

Перед описанием класса конвертера указывается атрибут ValueConversion с двумя параметрами: исходный тип данных и тип для преобразования. Само преобразование выполняется в функции Convert, аргумент value – преобразуемое значение.

Конвертеры могут работать в обе стороны, но в данном примере это не требуется, поэтому функция обратного преобразования просто возвращает null.

Чтобы использовать конвертер в XAML-коде, нужно добавить в атрибуты первого элемента (в примере - элемент Window) пространство имён (namespace) конвертера:

```
xmlns:conv="clr-namespace:Tpu.Aics.Converters"
```

и добавить конвертер в ресурсы:

```
<Window.Resources>
    <conv:ThereIsConverter x:Key="ThereIsConverter"/>
</Window.Resources>
```

Теперь, конвертер можно использовать:

```
<TextBlock Text="{Binding Path=Availability, Converter={StaticResource
ThereIsConverter}}, StringFormat={}{{0}} в наличии"/>
```

MultiBinding

При форматировании строк можно помещать в одну строку сразу несколько значений свойств связанного объекта. Такой механизм называется MultiBinding.

Выведем стоимость и валюту в одной строке.

```
<TextBlock Style="{StaticResource ValueStyle}">
    <TextBlock.Text>
        <MultiBinding StringFormat="{}{{0:F2}} {{1}}">
            <Binding Path="Price.Value"/>
            <Binding Path="Price.Currency"/>
        </MultiBinding>
    </TextBlock.Text>
</TextBlock>
```

Стоимость: 13990.00 RUR

Задание 1

Модифицировать программу из прошлой работы, добавить для некоторых значений форматирование строк, использовать конвертеры и MultiBinding.

Списочные элементы

В прошлой работе рассматривалась привязка данных к индивидуальным значениям. Для привязки к спискам или массивам данных используют списочные элементы управления.

Расширим JSON-файл с данными, добавив список дополнительных характеристик товара:

```
{
  "Id": 73766,
  "Name": "Asus ZenPad 8 Z380KNL",
  "Description": "Внешний вид Asus ZenPad 8 Z380KNL - яркий, стильный, современный - не оставит равнодушным ни одного пользователя!",
  "Price": {
    "Value": 13990,
    "Currency": "RUR"
  },
}
```

```

    "Availability": true,
    "Properties": [
        { "PropertyName": "Цвет", "Value": "Чёрный" },
        { "PropertyName": "Диагональ", "Value": "11" }
    ]
}

```

Для представления характеристики товара потребуется класс:

```

public class ItemProperty
{
    public string PropertyName { get; set; }
    public string Value { get; set; }
}

```

Теперь добавим список характеристик в класс-модель:

```

public class MarketItem
{
    public int Id { get; set; }

    public string Name { get; set; }

    public string Description { get; set; }

    public Price Price { get; set; }

    public bool Availability { get; set; }

    public List<ItemProperty> Properties { get; set; }
}

```

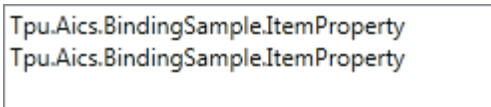
В XAML-код добавим списочный элемент (к списочным относятся ListBox, ListView, ComboBox и др.). Для связывания со свойством-списком указывается атрибут ItemsSource:

```

<ListBox ItemsSource="{Binding Properties}"/>

```

Запустив приложение, видим следующее:



Вместо значений характеристик выводится название класса. Это происходит потому, что WPF не знает, как отображать наш класс ItemProperty. Возможно следующее решение – переопределить в нашем классе метод ToString:

```

public override string ToString()
{
    return string.Format("{0}: {1}", PropertyName, Value);
}

```

Теперь в списке будут выводиться нужная нам информация. Однако, этот метод имеет недостаток: нарушается принцип независимости модели данных от способа её отображения.

Однако есть другой способ: WPF позволяет задать представление элементов списка в XAML.

Если требуется отобразить одно свойство элемента, можно указать его с атрибутом DisplayMemberPath

```

<ListBox ItemsSource="{Binding Properties}" DisplayMemberPath="Value"/>

```

Чёрный
11

Можно также задать форматирование строки через атрибут `ItemStringFormat` (аналогично `StringFormat` для индивидуальных элементов).

```
<ListBox ItemsSource="{Binding Properties}" DisplayMemberPath="Value"
        ItemStringFormat="Значение: {0}"/>
```

Значение: Чёрный
Значение: 11

Если же требуется отобразить несколько свойств (как в нашем случае - `PropertyName`, `Value`), потребуется задать шаблон для элементов списка `ItemTemplate`

```
<ListBox ItemsSource="{Binding Properties}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <TextBlock>
        <TextBlock.Text>
          <MultiBinding StringFormat="{0}: {1}">
            <Binding Path="PropertyName"/>
            <Binding Path="Value"/>
          </MultiBinding>
        </TextBlock.Text>
      </TextBlock>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Цвет: Чёрный
Диагональ: 11

Как и в других случаях, шаблоны позволяют больше, чем просто отобразить строку с несколькими свойствами.

```
<ListBox ItemsSource="{Binding Properties}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <TextBlock FontStyle="Italic" Text="{Binding Path=PropertyName,
          StringFormat={0}: }"/>
        <TextBlock FontWeight="Bold" Margin="4,0" Text="{Binding Value}"/>
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Цвет: Чёрный
Диагональ: 11

Наблюдаемые коллекции

Допустим, в наш список характеристик требуется добавить элемент. Добавим в обработчик какого-нибудь события (например, нажатия кнопки) следующий код:

```
var newProp = new ItemProperty()
```

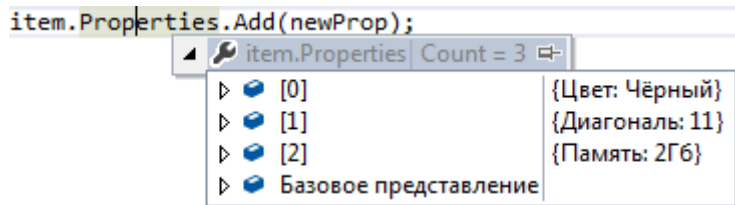
```

{
    PropertyName = "Память",
    Value = "2Г6"
};

var item = itemGrid.DataContext as MarketItem;
item.Properties.Add(newProp);

```

Однако запустив программу, мы увидим, что новый элемент не добавляется в список. При этом отладчик покажет, что значение переменной Properties содержит нужный элемент



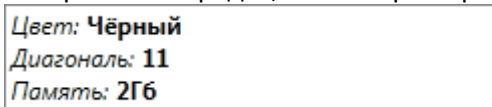
Так происходит, потому что переменная Properties имеет тип List (простой список), который не имеет функции автоматического обновления связанного списочного элемента. Зато этой функцией обладают наблюдаемые коллекции (ObservableCollection). Изменим тип свойства Properties класса-модели:

```

using System.Collections.ObjectModel;
...
public ObservableCollection<ItemProperty> Properties { get; set; }

```

Теперь всё в порядке, новая характеристика отображается в списочном элементе.



Таким образом, при работе со статическими списками можно использовать простые коллекции (списки, массивы и т.д.). Если же требуется добавлять/удалять элементы списка, следует использовать ObservableCollection.

Так же обратим внимание, что изменение типа List<ItemProperty> на ObservableCollection<ItemProperty> в классе MarketItem не повлияло на загрузку данных из JSON.

Задание 2

Модифицировать JSON, заменив одиночное значение на массив (согласно строке для своего варианта из таблицы ниже).

customer.json	<pre> "PhoneNumber": [{ "Type": "home", "Number": "212 555-1234" }, { "Type": "work", "Number": "727-35-73" }] </pre>
food.json	<pre> "Param": [{ "Name": "Дополнительно", "Text": "Kimbo Gold Arabica: содержание арабики 100%" }, { </pre>

	<pre> "Name": "Степень обжарки", "Text": "средняя" }] </pre>
star-fit.json	<pre> "Param": [{ "Name": "Материал", "Text": "Пластик, Металл" }, { "Name": "Вес, кг", "Text": "1.38" }] </pre>
tonic.json	<pre> "Param": [{ "Name": "Тип кожи", "Text": "Для всех типов кожи" }, { "Name": "Производитель", "Text": "США" }] </pre>
weigher.json	<pre> "Param": [{ "Name": "Тип", "Text": "Напольные весы" }, { "Name": "Максимальный вес, кг", "Text": "180" }] </pre>
widget.json	<pre> "Image": [{ "Src": "Images/Sun.png", "Name": "sun1", "HOffset": 250, "VOffset": 250, "Alignment": "center" }, { "Src": "Images/Forest.png", "Name": "forest", "HOffset": 100, "VOffset": 250, "Alignment": "left" }] </pre>
email.json	<pre> "Copy": ["rosalee.fleming@enron.com", "steven.kean@enron.com"] </pre>
facebook.json	<pre> "Action": [{ "Name": "Like", "Link": "http://www.facebook.com/X998/posts/Y998" }, { "Name": "Comment", "Link": "http://www.facebook.com/X998/posts/Y998" }] </pre>
twitter.json	<pre> "ToUser": [{ "Id": 396524, </pre>

	<pre> "Name": "TwitterAPI" }, { "Id": 893574, "Name": "Ivanov89" }] </pre>
youtube.json	Добавить поле: <pre> "Content": ["rtsp://v5.cache3.c.youtube.com/CiILENy.../0/0/0/video.3gp", "http://www.youtube.com/v/hYB0mn5zh2c?f..."] </pre>
bank.json	<pre> "ProjectDocs": [{ "DocTypeDesc": "Procurement Plan (PROP), Vol.1 of 1", "DocType": "PROP", "EntityID": "000456286_20130926172640", "DocURL": "http://www-wds.worldbank.org/servlet/WDSServlet?pcont=details&eid=000456286_20130926172640", "DocDate": "2013-09-24", "Active": true }, { "DocTypeDesc": "Project Appraisal Document (PAD), Vol.1 of 1", "DocType": "PAD", "EntityID": "000356161_20131011122810", "DocURL": "http://www-wds.worldbank.org/servlet/WDSServlet?pcont=details&eid=000356161_20131011122810", "DocDate": "2013-10-02", "Active": false }] </pre>

Отобразить значения списка в списочном элементе (на выбор: ComboBox либо TextBox). Сделать интерфейс для добавления и удаления элементов списка.

Представление списка в виде таблицы

Приложение из примера отображало данные для одного товара. Изменим его для работы со списком товаров. Расширим JSON-файл, заменив одиночный объект на массив объектов (и добавим в этот массив ещё один объект)

```

[
  {
    "Id": 73766,
    "Name": "Asus ZenPad 8 Z380KNL",
    "Description": "Внешний вид Asus ZenPad 8 Z380KNL - яркий, стильный, современный - не оставит равнодушным ни одного пользователя!",
    "Price": {
      "Value": 13990,
      "Currency": "RUR"
    },
    "Availability": true,
    "Properties": [
      { "PropertyName": "Цвет", "Value": "Чёрный" },
      { "PropertyName": "Диагональ", "Value": "11" }
    ]
  },
  {
    "Id": 31616,
    "Name": "Samsung Galaxy S6",

```

```

    "Description": "Первый и единственный в мире смартфон с изогнутым с обеих сторон экраном.",
    "Price": {
        "Value": 36488.40,
        "Currency": "RUR"
    },
    "Availability": false,
    "Properties": [
        {"PropertyName": "Тип", "Value": "Смартфон"},
        {"PropertyName": "Год выпуска", "Value": "2015"},
        {"PropertyName": "Диагональ", "Value": "5.1"}
    ]
}
]

```

Добавим в проект новое окно GridWindow.xaml, и сделаем его главным окном приложения. Для этого нужно изменить значение параметра StartupUri в файле App.xaml

```
StartupUri="GridWindow.xaml"
```

В окно GridWindow.xaml добавим элемент ListView с типом отображения GridView

```

<ListView x:Name="grid">
    <ListView.View>
        <GridView>
            <GridView.Columns>

                <GridViewColumn Header="Артикул" DisplayMemberBinding="{Binding Path=Id}"/>
                <GridViewColumn Header="Название" DisplayMemberBinding="{Binding Path=Name}"/>
                <GridViewColumn Header="Описание" DisplayMemberBinding="{Binding Path=Description}" Width="200"/>

            </GridView.Columns>
        </GridView>
    </ListView.View>
</ListView>

```

К элементу ListView будет привязан наш список товаров. Для каждой колонки задаётся отображаемое свойство через параметр DisplayMemberBinding. Для DisplayMemberBinding, так же как и для других случаев привязки, можно использовать строки форматирования, конвертеры и Multibinding. Добавим ещё две колонки для цены и наличия товара с изменённым отображением:

```

<GridViewColumn Header="Наличие" DisplayMemberBinding="{Binding Path=Availability, Converter={StaticResource ThereIsConverter}, StringFormat="{0} в наличии}"/>

<GridViewColumn Header="Цена">
    <GridViewColumn.DisplayMemberBinding>
        <MultiBinding StringFormat="{0:F2} {1}">
            <Binding Path="Price.Value"/>
            <Binding Path="Price.Currency"/>
        </MultiBinding>
    </GridViewColumn.DisplayMemberBinding>
</GridViewColumn>

```

Для более сложного отображения можно воспользоваться шаблоном. В этом случае, колонке задаётся параметр CellTemplate (вместо DisplayMemberBinding). Добавим колонку с отображением первого элемента из Properties

```

<GridViewColumn Header="Дополнительно">
    <GridViewColumn.CellTemplate>
        <DataTemplate>

```



```

        <StackPanel Orientation="Horizontal">
            <TextBlock FontStyle="Italic" Text="{Binding
                Path=Properties[0].PropertyName, StringFormat={}{}{0}:}" />
            <TextBlock FontWeight="Bold" Margin="4,0" Text="{Binding
                Path=Properties[0].Value}" />
        </StackPanel>
    </DataTemplate>
</GridViewColumn.CellTemplate>
</GridViewColumn>

```

Привязку данных сделаем в коде, аналогично прошлому примеру. С одним отличием: теперь JSON представляет не одиночный объект MarketItem, а их список/массив/коллекцию. Поэтому при десериализации JSON укажем тип `ObservableCollection<MarketItem>`

```

// Загрузка содержимого файла в строку
string json = File.ReadAllText(@"..\..\items.json");

// Десериализация JSON-данных в объект MarketItem
var items = JsonConvert.DeserializeObject<ObservableCollection<MarketItem>>(json);

// Привязка данных к графическому интерфейсу
grid.ItemsSource = items;

```

Артикул	Название	Описание	Наличие	Цена	Дополнительно
73766	Asus ZenPad 8 Z380KNL	Внешний вид Asus ZenPad 8 Z380K	Есть в наличии	13990.00 RUR	Цвет: Чёрный
31616	Samsung Galaxy S6	Первый и единственный в мире с	Нет в наличии	36488.40 RUR	Тип: Смартфон

Задание 3

Изменить JSON-документ, вместо одиночного объекта сделать массив таких объектов. Добавить в массив 1-2 объекта с такой же структурой, но другими значениями.

Добавить в проект новое окно, сделать его основным. В окно добавить элемент ListView с типом отображения GridView. Добавить колонки для свойств объектов с использованием конвертеров и форматов строк (если в JSON-файле Вашего варианта более десяти свойств, не обязательно описывать колонки для всех свойств, можно выбрать 10 из них на своё усмотрение). Привязать данные из JSON для отображения в таблице.