Data Binding: Collections

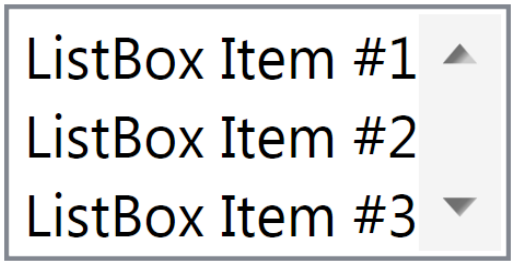
DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

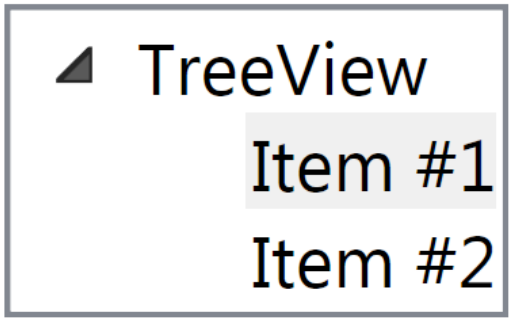
Reminder: several types of controls

- Several predefined ways available to present data in WPF

Single object – displays single piece of content
typically derived from `ContentControl`



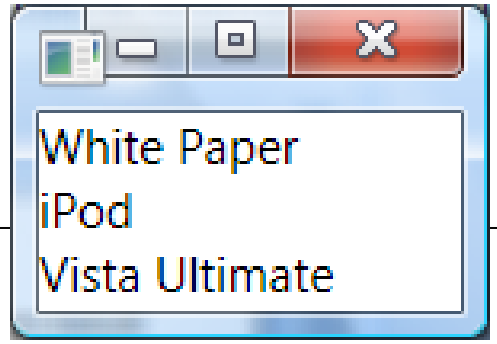
Collection – displays a list of items
typically derives from `ItemsControl`



Hierarchy – displays a related group of items
typically derives from `HeaderedItemsControl`

Displaying collections of objects

- Collection oriented controls derive from `ItemsControl`
  - **`ItemsSource`** property assigns data source
- Data source can come from a variety of places
  - **`IList`** (collections, arrays, generic collections)
  - **`IBindingList[View]`** (**`BindingList<T>`**, **`DataTable`**)
  - **`IEnumerable`**[1](LINQ, iterator methods)

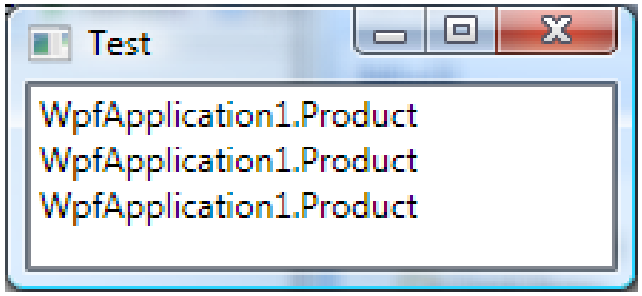



```
void InitializeComponent()
{
   string[] productList = GetProductNamesFromDatabase();
   itemList.ItemsSource = productList;
}
```

Default binding behavior

- Data binding sources are typically non-visual elements
  - works fine in the singular form, but not so well with collections

```
void InitializeComponent()
{
   Products[] productList = GetProductListFromDatabase();
   itemList.ItemsSource= productList;
}
```



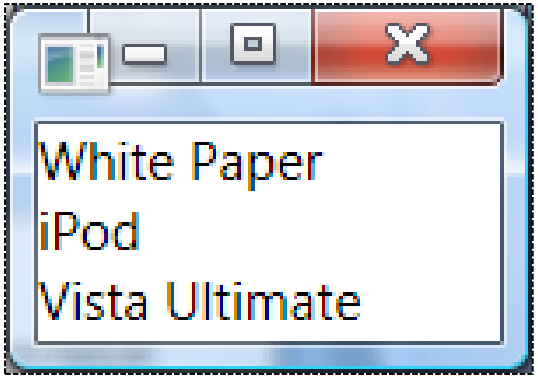
… it works but WPF does not know how to display the `Product` object so it converts it to a string and uses a single `TextBlock` as the visual tree

Restricting data view to a single property

- Can tell WPF to use a specific property of the object
  - **`DisplayMemberPath`** identifies one public property to display
  - uses **`TextBlock`** to display property value

```
<ListBox DisplayMemberPath="Name"
         ItemsSource="{Binding}" />
```

```
public class Product
{
   public string Name { get; set; }
   public double Price { get; set; }
      ...
}
```

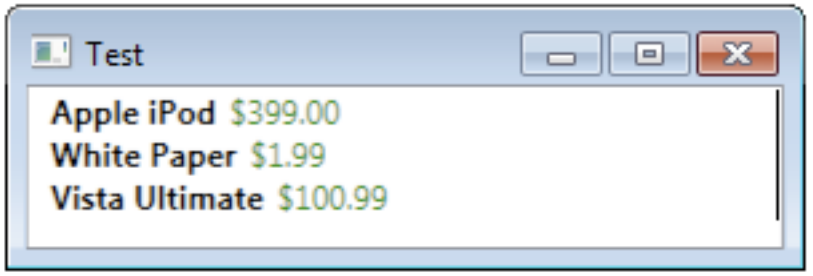


Going beyond strings using DataTemplates

- Designers can specify the visual tree through a `DataTemplate`
  - directs WPF to create a set of visual objects for a non-visual
- Assigned to appropriate template property of control
  - **`ContentControl.ContentTemplate`**
  - **`ItemsControl.ItemTemplate`**

```
<ListBox ItemTemplate="{StaticResource productVisual}"
         ItemsSource="{Binding}" />
```

note how multiple text elements are generated for each Product



Creating a Data Template

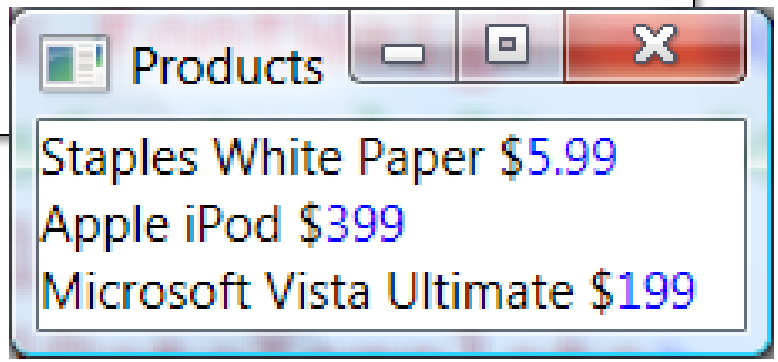
1. Declare a `DataTemplate` object
2. Set the `DataType` property to the `System.Type` to represent
3. Add the visual representation as the content
4. Bind the properties of the visuals to the underlying type
   - **`DataContext`** automatically set to data object being templated
5. Assign the data template to a control

```
<DataTemplate DataType="{x:Type local:Product}">
   <StackPanel Orientation="Horizontal">
      <TextBlock FontWeight="Bold" Text="{Binding Name}" />
      <TextBlock Foreground="Green"
                 Text="{Binding Price, StringFormat=C}" />
   </StackPanel>
</DataTemplate>
```

Assigning the DataTemplate

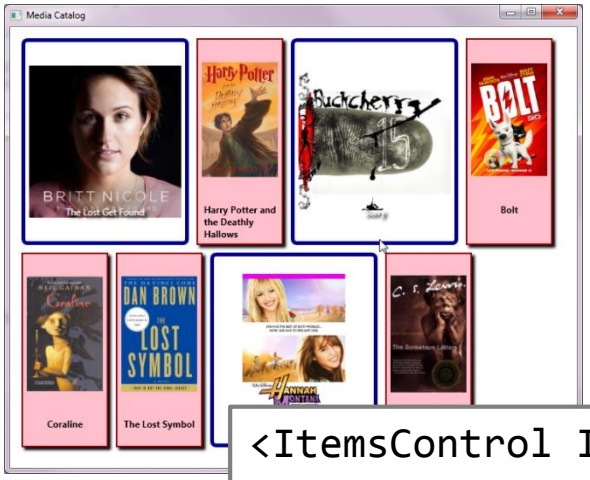
- `DataTemplate` must be assigned to `ItemTemplate`
  - can be defined inline or in resources (preferred)
  - if supplied, cannot use **`DisplayMemberPath`**

```
<ListBox ItemsSource="{Binding}"
   <ListBox.ItemTemplate>
      <DataTemplate DataType="{x:Type Product}">
         <StackPanel>...</StackPanel>
      </DataTemplate>
   </ListBox.ItemTemplate>
</ListBox>
```



Dynamically selecting a template

- **`DataTemplateSelector`** dynamically selects template[1]
  - assigned to **`ContentTemplateSelector`** or **`ItemTemplateSelector`** property



different media types presented together – videos, books, movies, etc.

`DataTemplateSelector` allows runtime selection of template based on code decisions

```
<ItemsControl ItemsSource="{Binding MediaItems}" ...>
   <ItemsControl.ItemTemplateSelector>
      <local:MediaTemplateSelector
         BestSeller="{StaticResource bsTemplate}"
         Normal="{StaticResource normalTemplate}" />
   </ItemsControl.ItemTemplateSelector>
</ItemsControl>
```

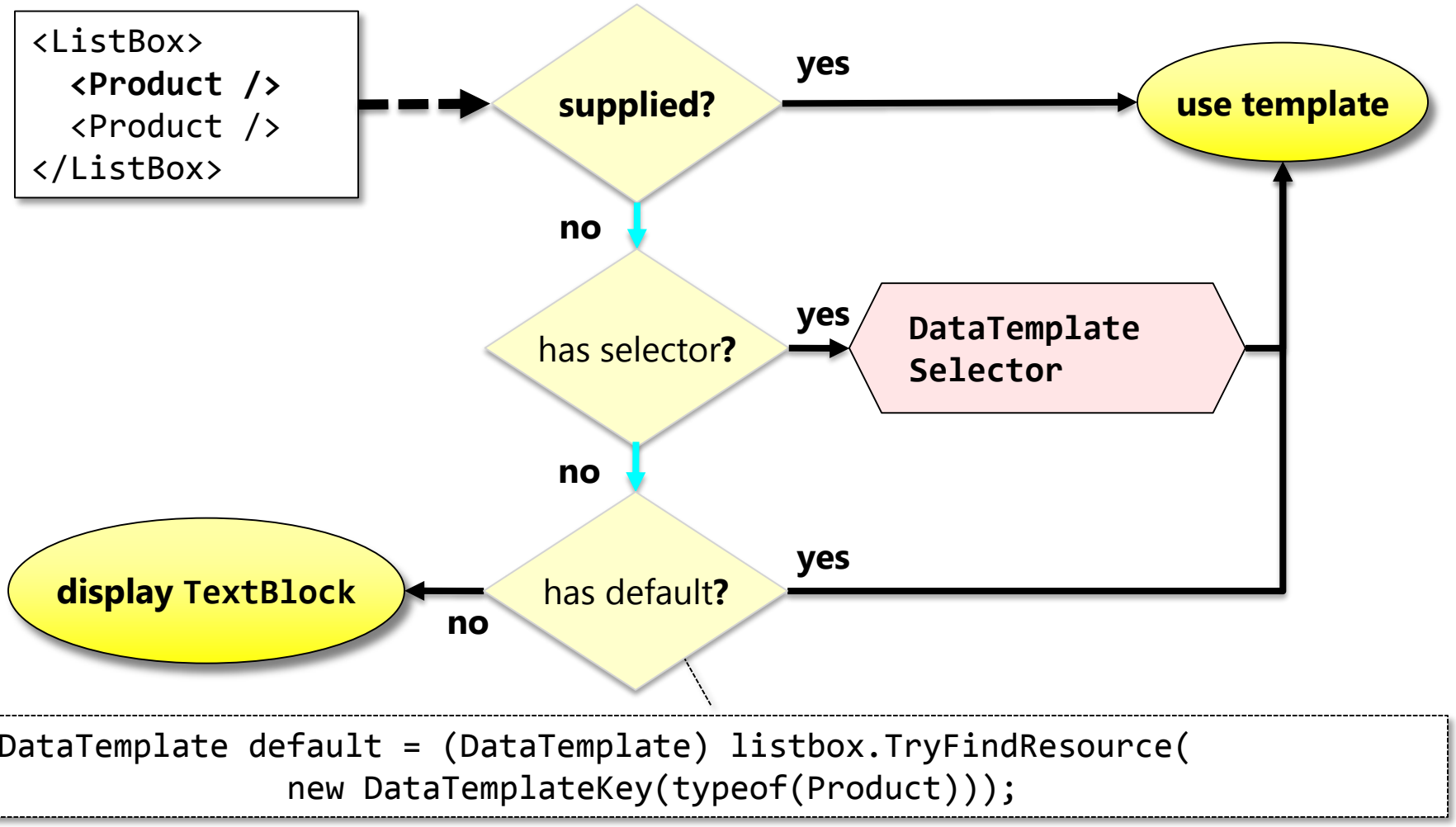
Creating a DataTemplateSelector

- `SelectTemplate` is called for each item to resolve template
  - method can look at item and determine appropriate template

```
public class MediaTemplateSelector : DataTemplateSelector
{
   public DataTemplate BestSeller { get; set; }
   public DataTemplate Normal { get; set; }

   public override DataTemplate SelectTemplate(
                  object item, DependencyObject container)
   {
      if (item is MusicAlbum && ((MusicAlbum)item).BestSeller)
         return BestSeller;
      if (item is Video && ((Video)item).Popularity > 90)
         return BestSeller;
      ...
      return Normal;
   }
}
```

How does WPF find the template?



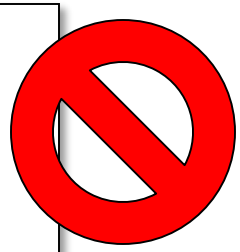

```
DataTemplate default = (DataTemplate) listbox.TryFindResource(
                new DataTemplateKey(typeof(Product)));
```

Adding and removing items

- Cannot modify `Items` collection of bound `ItemsControls`
  - **`Items`** collection is marked read-only
  - you must change the data *in the underlying collection*

```
<ListBox x:Name="productListBox" ItemsSource="{Binding}" />
```

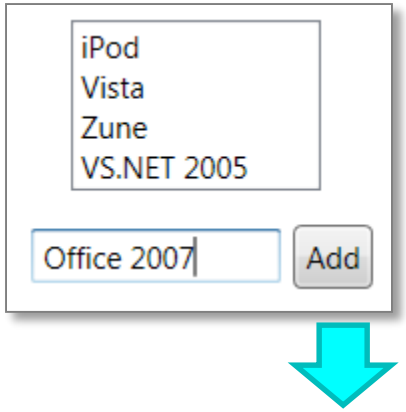
```
void OnAdd(Product newProduct)
{
   productListBox.Items.Add(newProduct);
}
```

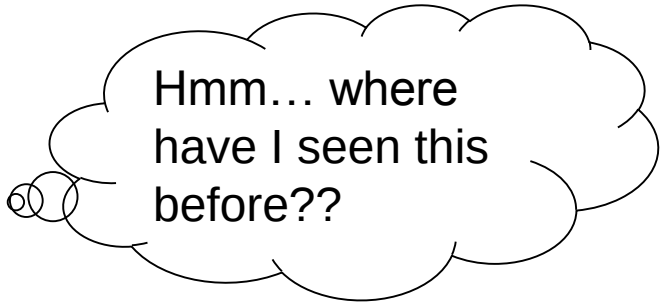


```
List<Product> productList;
void OnAdd(Product newProduct)
{
     productList.Add(newProduct);
}
```

How does WPF know the collection changed?

- Collections in .NET have no built-in change notification
  - WPF cannot "see" changes made to underlying collection





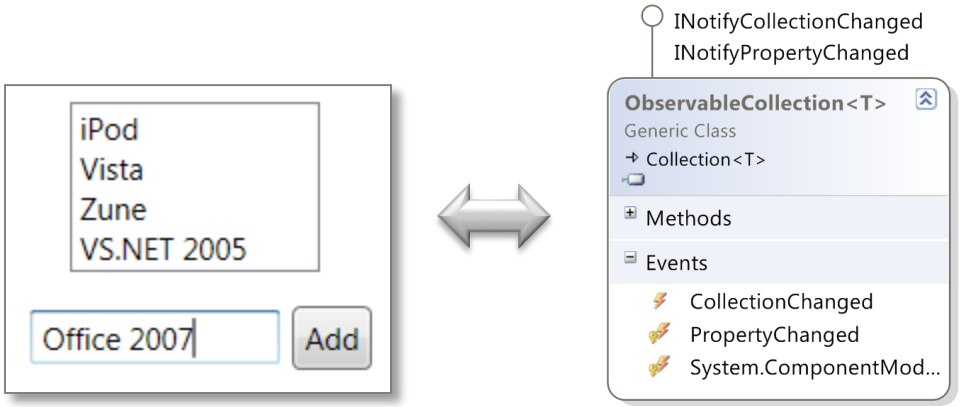

```
List<Product> productList;
void OnAdd(object sender, RoutedEventArgs e)
{
   productList.Add(new Product(textBoxProduct.Text));
}
```

item is added to collection, but not `ListBox`

Two-way binding to collections

- `ObservableCollection<T>` provides notification support
  - implements **`INotifyCollectionChanged`**
  - keeps collection and bound target synchronized

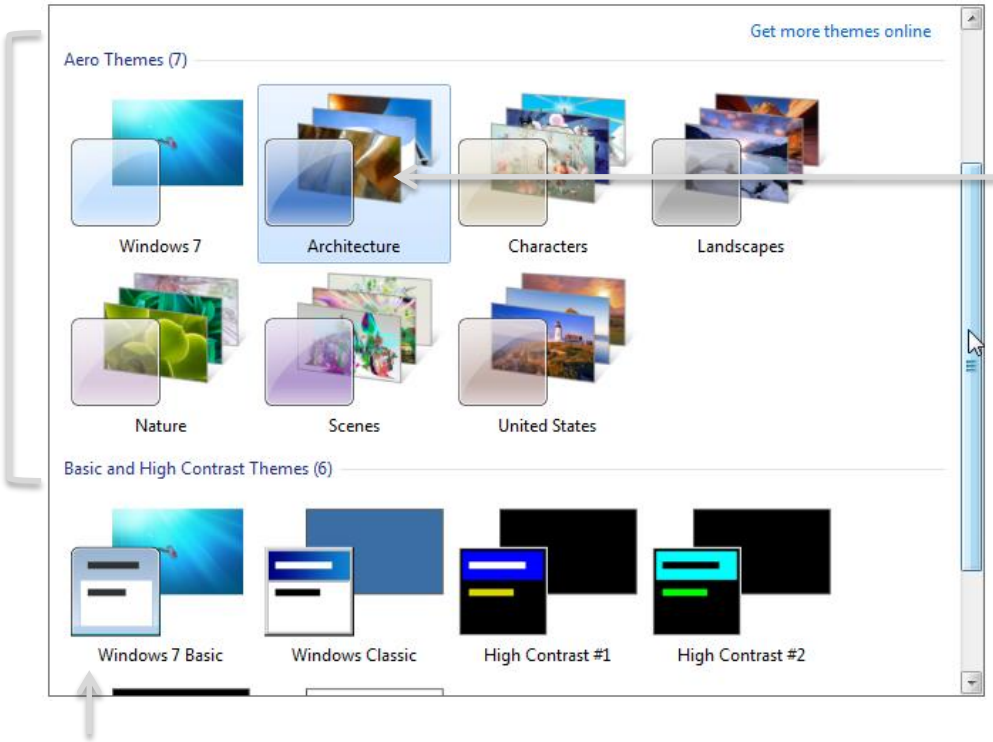
```
ObservableCollection<Product> productList;
void OnAdd(object sender, RoutedEventArgs e)
{
    productList.Add(new Product(textBoxProduct.Text));
}
```



Mapping collections to UI elements

- UI often has capabilities not expressed by collections

items are grouped by category



one or more items are selected

items are sorted in groups

CollectionView

- ItemsControls use ICollectionView to manage binding
  - created automatically for any data-bound collection
  - accessible by **CollectionViewSource.GetDefaultView**
  - can maintain current position in collection (currency)

```
ObservableCollection<Product> productList;
DataContext = productList;
...
ICollectionView cv = CollectionViewSource.GetDefaultView(
                                productList);
Product selectedProduct = cv.SelectedItem as Product;
```

```
<ListBox x:Name="productListBox"
    IsSynchronizedWithCurrentItem="true"
    ItemsSource="{Binding}" />
```

Creation of the ICollectionView class

- WPF wraps all bound collections in an `ICollectionView`
  - **`IList` → `ListCollectionView`**
  - **`IBindingList` → `BindingListCollectionView`**
  - **`IEnumerable` → `EnumerableCollectionView`**
- Can also create custom collection views
  - for specialized logic or to optimize sorting/filtering/grouping

Here we will define a "paging" collection view to avoid a large scrolling area

```
class PagedCollectionView : ICollectionView
{
   ...
}
```

And then assign it as the data source

```
productListBox.ItemsSource = new CustomListView(productList);
```

Monitoring selection change with CollectionView

- `ICollectionView` supports selection change notification
  - **`CurrentChanging`** raised before selection is changed[1]
  - **`CurrentChanged`** raised after selection is changed

```
public class MainWindow : Window
{
   private Window_Loaded(object sender, RoutedEventArgs e)
   {
      ICollectionView cv =
        CollectionViewSource.GetDefaultView(productList);
      cv.CurrentChanged += this.OnSelectionChanged;
   }

   void OnSelectionChanged(object sender, EventArgs e)
   {
     ...
   }
}
```

Navigating CollectionViews

- `ICollectionView` has several properties to manage position
  - **`CurrentItem`** returns current item (null if none)
  - **`CurrentPosition`** exposes current item index (-1 if none)
  - **`MoveXXX`** methods change position
  - **`IsXXX`** test current position for boundaries

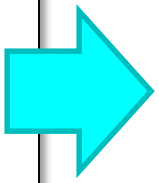
```
void OnClickNext(object sender, RoutedEventArgs e)
{
   ICollectionView cv =
       CollectionViewSource.GetDefaultView(productList);
   cv.MoveCurrentToNext();
   if (cv.IsCurrentAfterLast)
      cv.MoveCurrentToFirst();
}
```

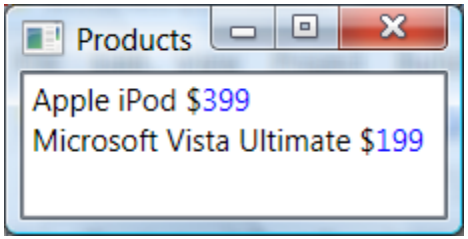
* requires `IsSynchronizedWithCurrentItem = "true"`

Filtering Items

- Items in the view may be filtered through the `Filter` property
  - items passed through **`Predicate<object>`** to perform filtering[1]
  - not supported by all implementations of **`ICollectionView`**

```
private double _maxPrice = 500;
private bool PriceLessMax(object o)
{
   Product prod = (Product) o;
   return prod.Price < _maxPrice;
}
```



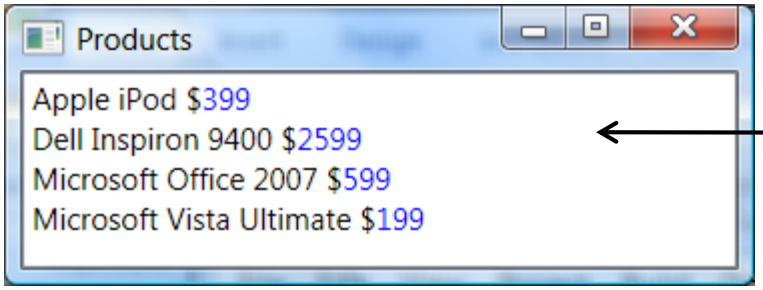



```
ICollectionView cv =
         CollectionViewSource.GetDefaultView(productList);
if (cv.CanFilter)
   cv.Filter = new Predicate<object>(PriceLessMax);
```

the underlying collection is not changed – only the view

Sorting items

- Adding `SortDescriptions` to the view sorts items[1]
  - can add more than one to provide sub-sort criteria



sorted by Manufacturer
and then price

```
ICollectionView cv =
      CollectionViewSource.GetDefaultView(productList);
using (cv.DeferRefresh()) {
   cv.SortDescriptions.Clear();
   cv.SortDescriptions.Add(new SortDescription("Manufacturer",
                                ListSortDirection.Ascending));
   cv.SortDescriptions.Add(new SortDescription("Price",
                                ListSortDirection.Descending));
}
```

Custom Sorting

- `ListCollectionView` supports `IComparer` based sorting
  - much faster than using **`SortDescription`**
  - requires that underlying collection is **`IList`**

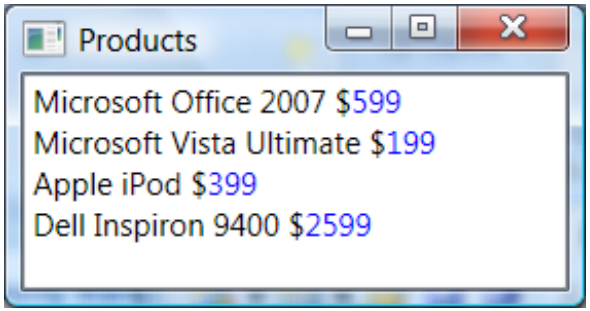
custom sorting logic called for each item in collection

```
class ProductSorter : IComparer
{
   public int Compare(object x, object y)
   {
      ...
   }
}
```

```
ListCollectionView cv =
       CollectionViewSource.GetDefaultView(productList)
           as ListCollectionView;
if (cv != null)
   cv.CustomSort = new ProductSorter();
```

Grouping Items

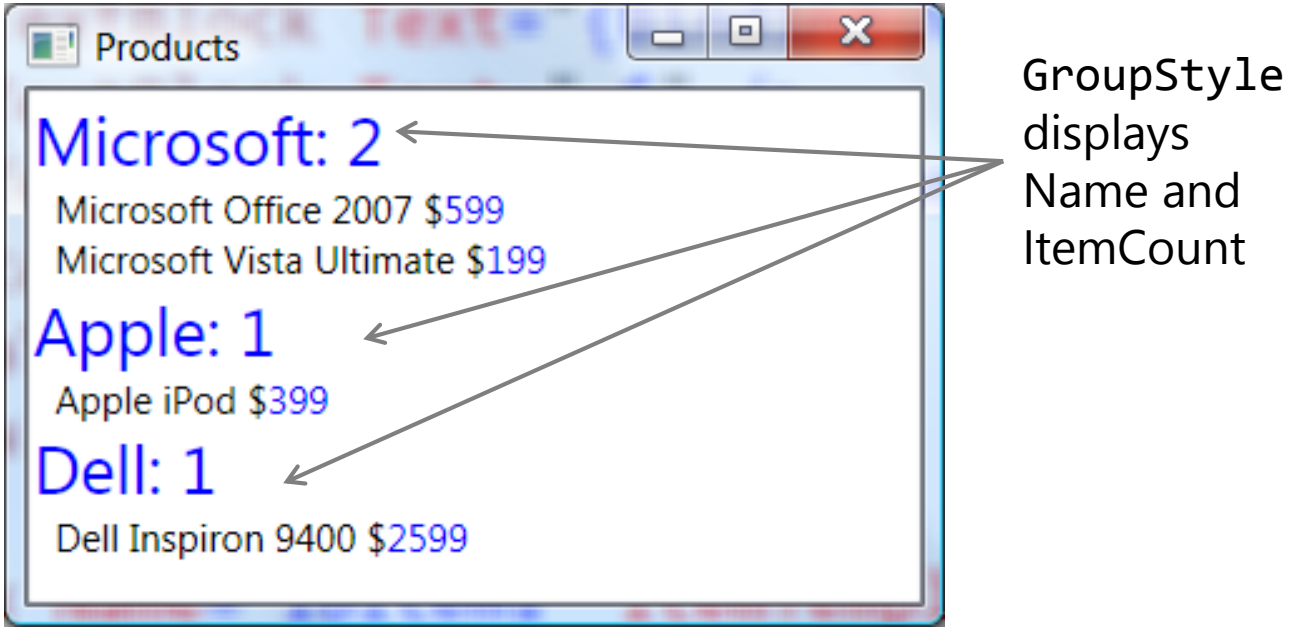
- Collections provide grouping through `GroupDescriptions`
  - can group by property (**`PropertyGroupDescription`**)
  - can provide custom **`GroupDescription`**-derived class[1]
  - disables **`ItemsControl`** virtualization[2]




```
ICollectionView cv =
     CollectionViewSource.GetDefaultView(productList);
if (cv.SupportsGrouping)
{
   cv.GroupDescriptions.Clear();
   cv.GroupDescriptions.Add(
                new PropertyGroupDescription("Manufacturer"));
}
```

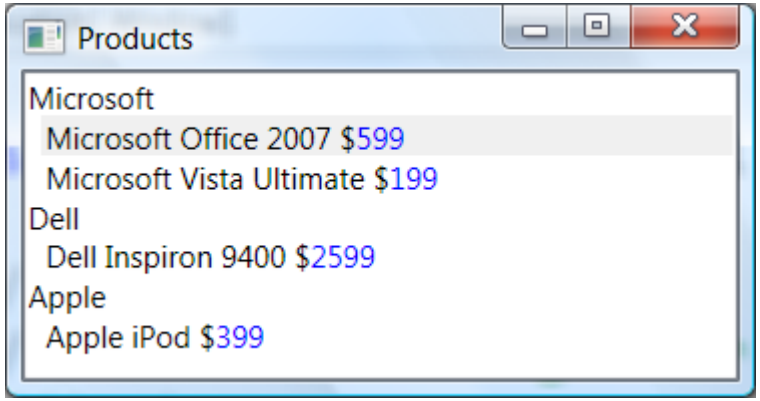
Defining the group header

- Headers can be defined to separate groups by `GroupStyle`
  - WPF wraps each group in a **`CollectionViewGroup`** - anything from this object may be used in header



Using the default group header

- WPF provides a default group header which may be used
  - name of the group is used as the header




```
<ListBox ItemTemplate="{StaticResource prodTempl}">
   <ListBox.GroupStyle>
      <x:Static Member="GroupStyle.Default" />
   </ListBox.GroupStyle>
</ListBox>
```

Defining a custom group header

- Headers defined as a `DataTemplate`
  - underlying **`DataType`** is a **`CollectionViewGroup`**

```
<ListBox Name="lbItems" ItemTemplate="{StaticResource prodTempl}">
   <ListBox.GroupStyle>
      <GroupStyle>
         <GroupStyle.HeaderTemplate>
            <DataTemplate>
               <StackPanel Orientation="Horizontal">
                  <TextBlock Text="{Binding Name}" />
                  <TextBlock Text=": "/>
                  <TextBlock Text="{Binding ItemCount}" />
               </StackPanel>
            </DataTemplate>
         </GroupStyle.HeaderTemplate>
      </GroupStyle>
   </ListBox.GroupStyle>
</ListBox>
```

Dynamic changes to objects

- Sorting / Filtering / Grouping is a useful feature
  - but is only applied once at assignment or as items are added
  - changes to object properties do not impact collection view
- Code can use the `Refresh` method to force recalculation
  - necessary if properties could change over time

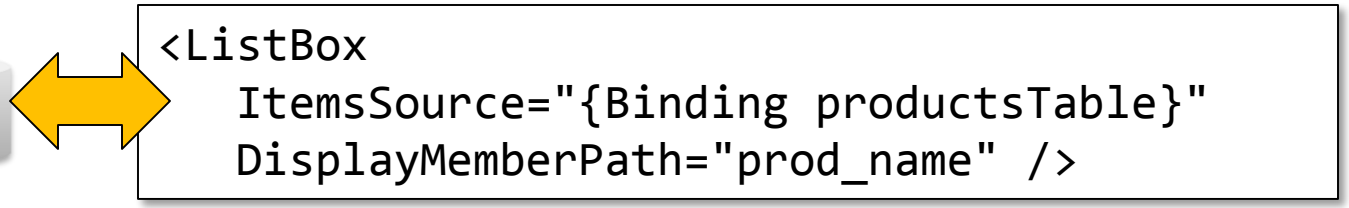
```
<TextBox SourceUpdated="OnPriceChanged"
     Text="{Binding Price, NotifyOnSourceUpdated=true}" />
```

anytime the Price of any product is changed we need to refresh the collection view contents to re-sort, group and filter based on the new value

```
void OnPriceChanged(object sender, DataTransferEventArgs e)
{
   ICollectionView cv =
      CollectionViewSource.GetDefaultView(productlist);
   cv.Refresh();
}
```

Binding to ADO.NET objects

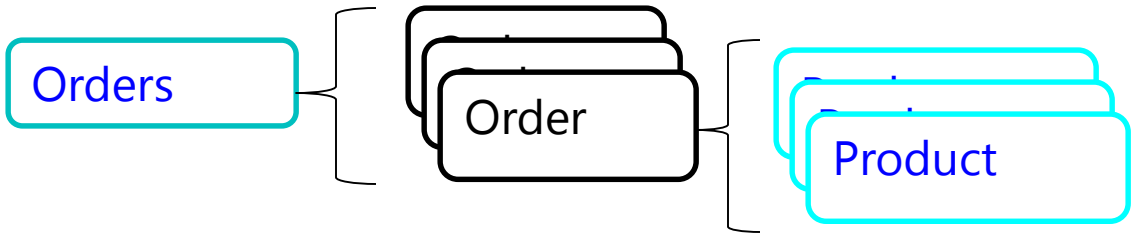
- WPF has explicit support for `DataTable` and `DataView`
  - exposes schema as bindable column "properties"
  - change notifications supported through **`IBindingList`**
- Performance was dramatically improved for WPF 3.5
  - better support for **`DBNull`** and identity mapping provided
  - reduced refreshes for changing data
- Should prefer to filter and sort using `DataView`



```
<ListBox
   ItemsSource="{Binding productsTable}"
   DisplayMemberPath="prod_name" />
```

Displaying child relationships

- Some data structures have parent/child relationships
  - common with relational data
  - useful to data bind these structures to lists or trees

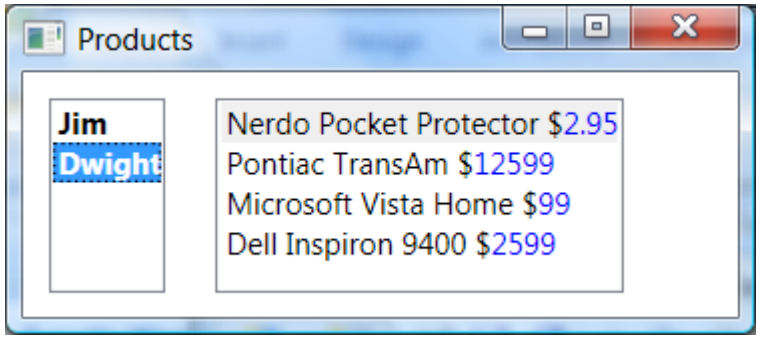



```
ObservableCollection<Order> Orders;

public class Order
{
    public string Customer {...}
    public ObservableCollection<Product> Products {...}
}
```

Binding master/detail items to lists [example 1]

- `ItemsControls` can be synchronized to display child collections
  - "child" control uses "/" in binding **`Path`** to indicate current item



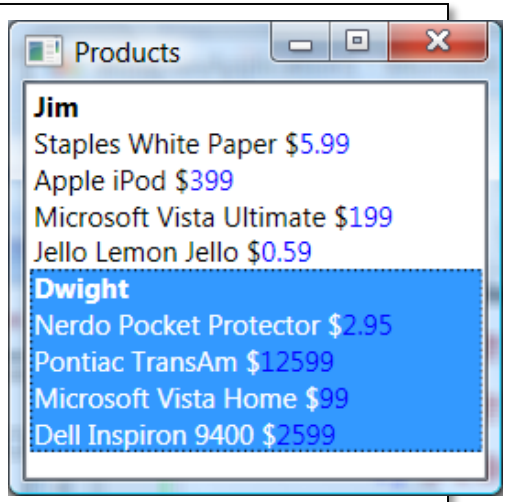

```
<StackPanel Orientation="Horizontal">
   <ListBox IsSynchronizedWithCurrentItem="True"
            ItemsSource="{Binding Orders}" />
   <ListBox IsSynchronizedWithCurrentItem="True"
            ItemsSource="{Binding Orders/Products}" />
</StackPanel>
```

Binding parent/child items to lists [example 2]

- `DataTemplates` can contain child collections
  - WPF groups templates together into one selectable item

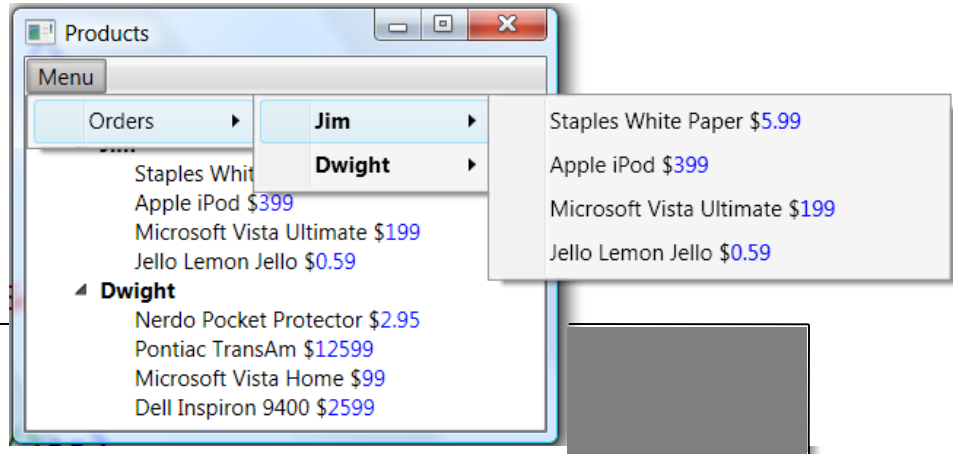
```
<DataTemplate
      DataType="{x:Type local:Product}">
   ...
</DataTemplate>

<DataTemplate
      DataType="{x:Type local:Order}">
    <StackPanel>
       <TextBlock FontWeight="Bold"
                  Text="{Binding Customer}" />
       <ItemsControl
                  ItemsSource="{Binding Products}" />
    </StackPanel>
</DataTemplate>
```



Working with hierarchies

- Hierarchical controls can be data bound using `ItemsSource`
  - typically used to represent parent-child details
  - each level defined by separate data template




```
<Menu DockPanel.Dock="Top">
   <MenuItem Header="Menu">
      <MenuItem Header="Orders" ItemsSource="{Binding}" />
   </MenuItem>
</Menu>
<TreeView>
   <TreeViewItem Header="Orders" ItemsSource="{Binding}" />
</TreeView>
```

Defining the shape of the hierarchical data

- Hierarchical data is defined by `HierarchialDataTemplate`
  - identifies additional **`ItemsSource`** for children
  - **`ItemTemplate`** identifies **`DataTemplate`** for children[1]

```
<DataTemplate x:Key="prdTempl">
   <StackPanel> ... </StackPanel>
</DataTemplate>

<HierarchicalDataTemplate x:Key="orderTempl"
        ItemsSource="{Binding Products}"
        ItemTemplate="{StaticResource prdTempl}">
   <TextBlock FontWeight="Bold"
              Text="{Binding Customer}" />
</HierarchicalDataTemplate>
```

could use hierarchical data template here to have multiple levels

property of `Order` which has child collection

Common Collection Bindings Cheat Sheet

Binding Syntax	Description
{Binding}	Bind to the current data context
{Binding /}	Bind to the current item in the data context – this assumes the data context is a collection
{Binding Items/}	Bind to the current item in the "Items" property on the data context – this assumes "Items" is a collection
{Binding Items/Text}	Bind to the "Text" property of the current item in the "Items" property on the data context

remember that to maintain current position, you must set the property `IsSynchronizedWithCurrentItem = "true"`

Summary

- Visual representation of data can be defined in markup
  - can be changed by UI designer without altering code
- `CollectionView` used to wrap bound collection data
  - sorting
  - grouping
  - filtering
- Data Binding supports collections natively
  - **`ObservableCollection<T>`** provides two-way binding
- Hierarchies supported through `HierarchialDataTemplate`
  - can add multiple levels