MVVM - Databinding

Enterprise & Mobile .Net

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

MVVM

- Model – View – ViewModel pattern
- Used for XAML based applications
- Makes applications more testable and easier to maintain

What will we be building

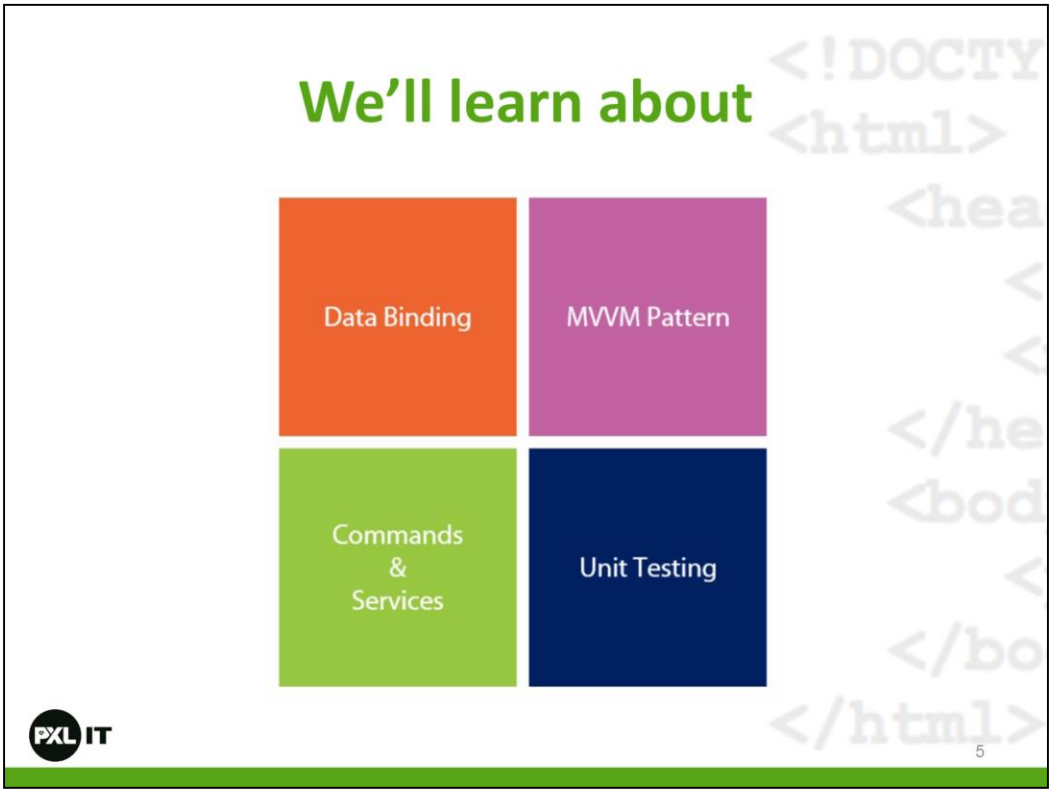
- **Joe's Coffee Store Administration**
  - WPF application
  - Stock management application
  - View with list of coffees
  - View with details of coffee

PXL IT

3

Demo: Taking a look at the ompleted application

PXL IT


We'll learn about
Data Binding
MVVM Pattern
Commands & Services
Unit Testing
PXL IT
5

Before you can learn about MVVM

You have knowledge of **XAML**

You are familiar with **WPF** on a basic level

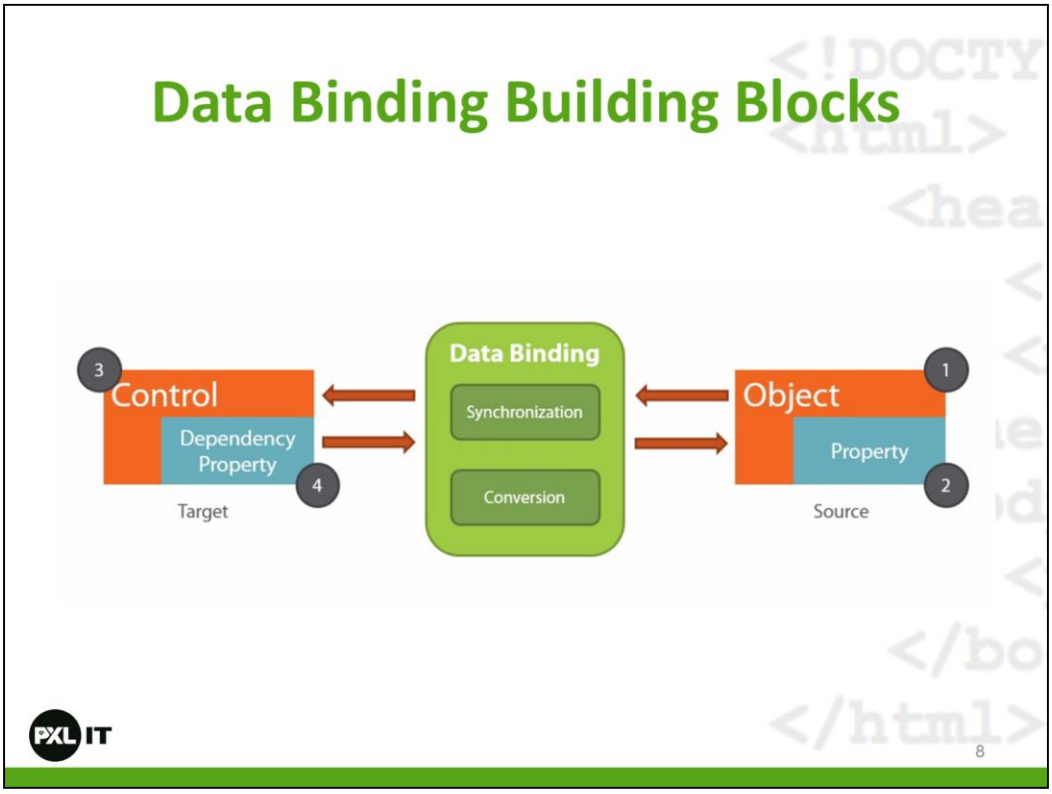
You want to learn how to create **maintainable** and **testable** apps

PXL IT

Data Binding

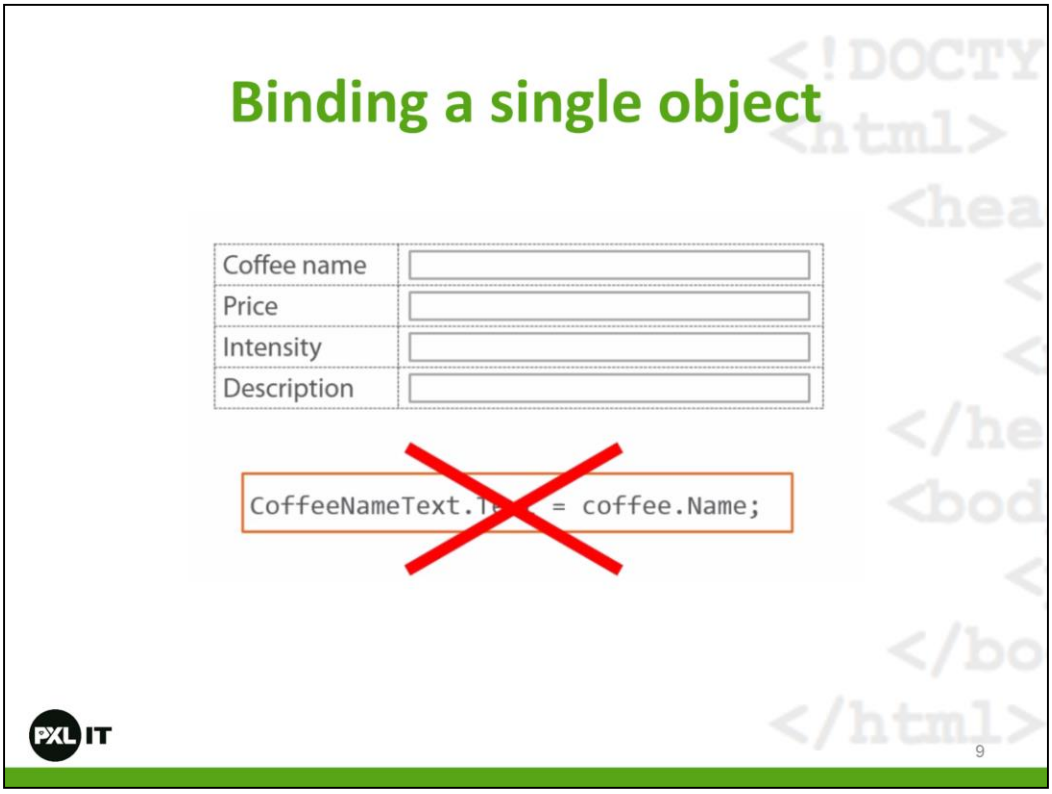
- Infrastructure that enables linking / binding the data of objects to properties of controls in our UI
- XAML feature

PXL IT

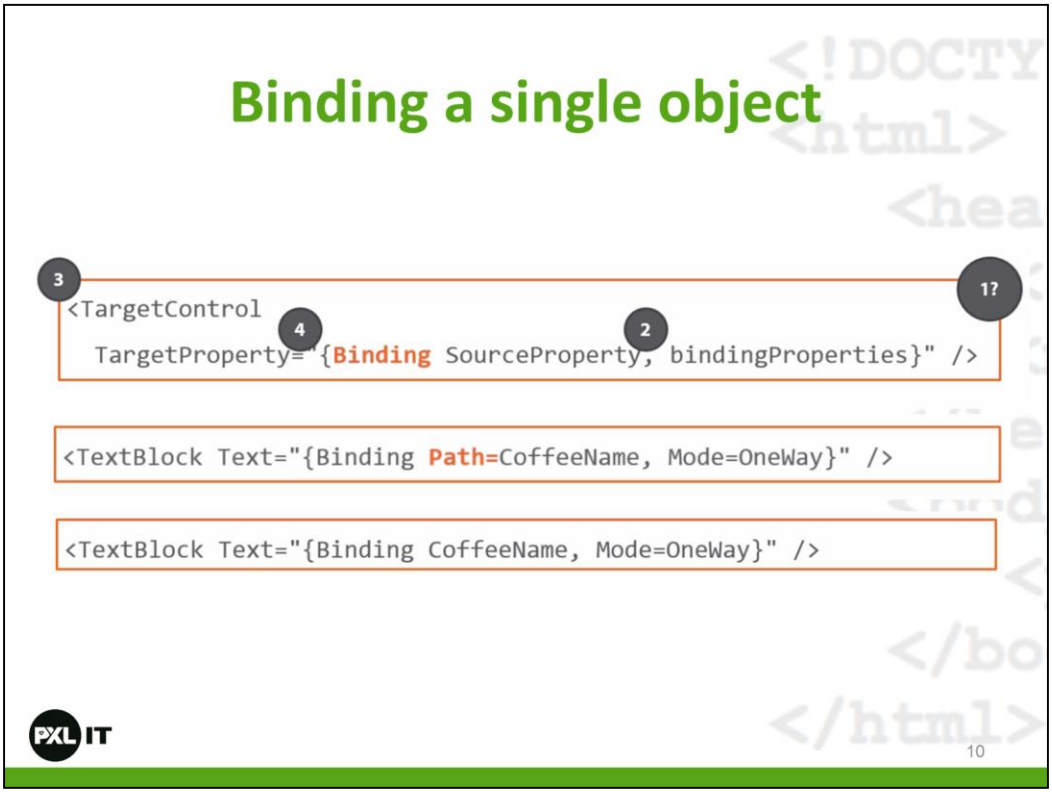


Any databinding is always composed out of 4 important building blocks. On one hand we have the source of the binding. This will be an object. And in this object, data is available in properties. So the source will number one and the source property will be number 2. On the other hand we have a UI control to which we're binding our data. This control is the target of the binding. On these controls, we have a specific type of properties named dependency properties. Most properties on UI controls are in fact dependency properties so will be able to use most of them in data binding statements. The UI control, so the target control is number 3, and its dependency property is the target property and that will be number 4. The most important thing to see here, is that, in order to function, the data binding needs to have these 4 building blocks assigned. When 1 is missing, the databinding will not work.

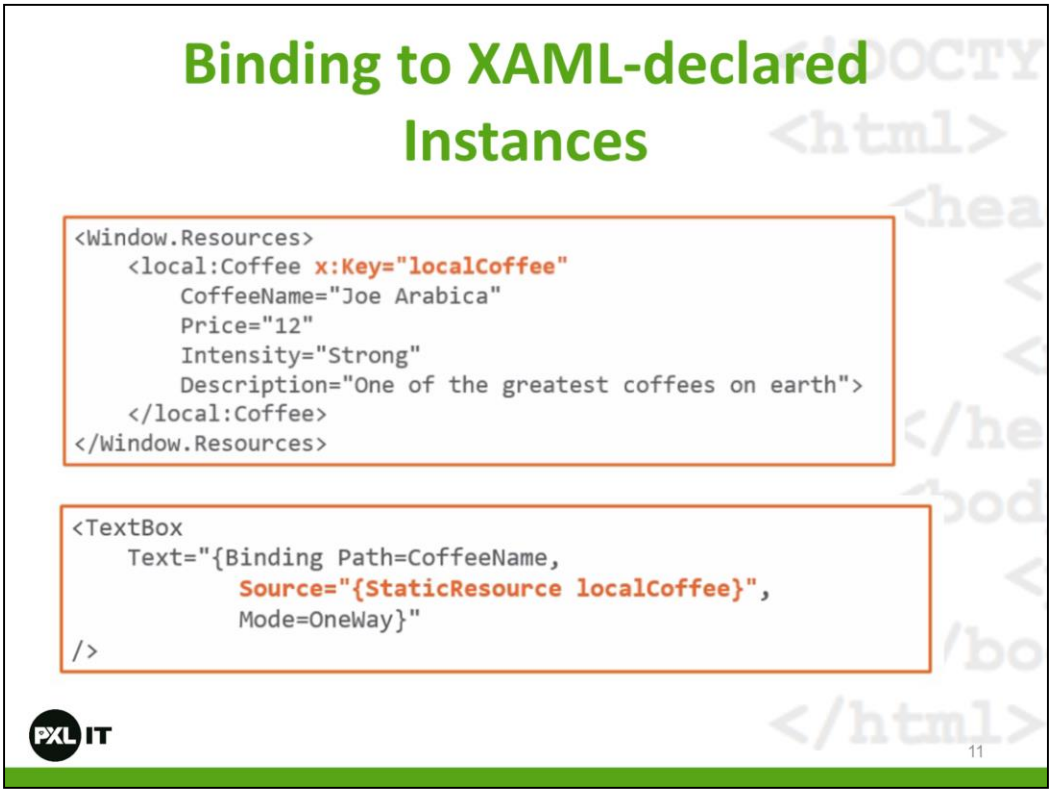
Our data can flue from the source to the target. Visa versa, data can also flue from the target to the source. This could be for example when the user has entered a new value in a text box in the UI. If this source object implements the correct interface, then the databinding engine will make sure that the source and the target stay in sync. Also, the databinding engine gives us a hook in the data binding process through conversions. Which can happen on our data if needed. This could be for example a formatting happening on a string we need to display in the UI.



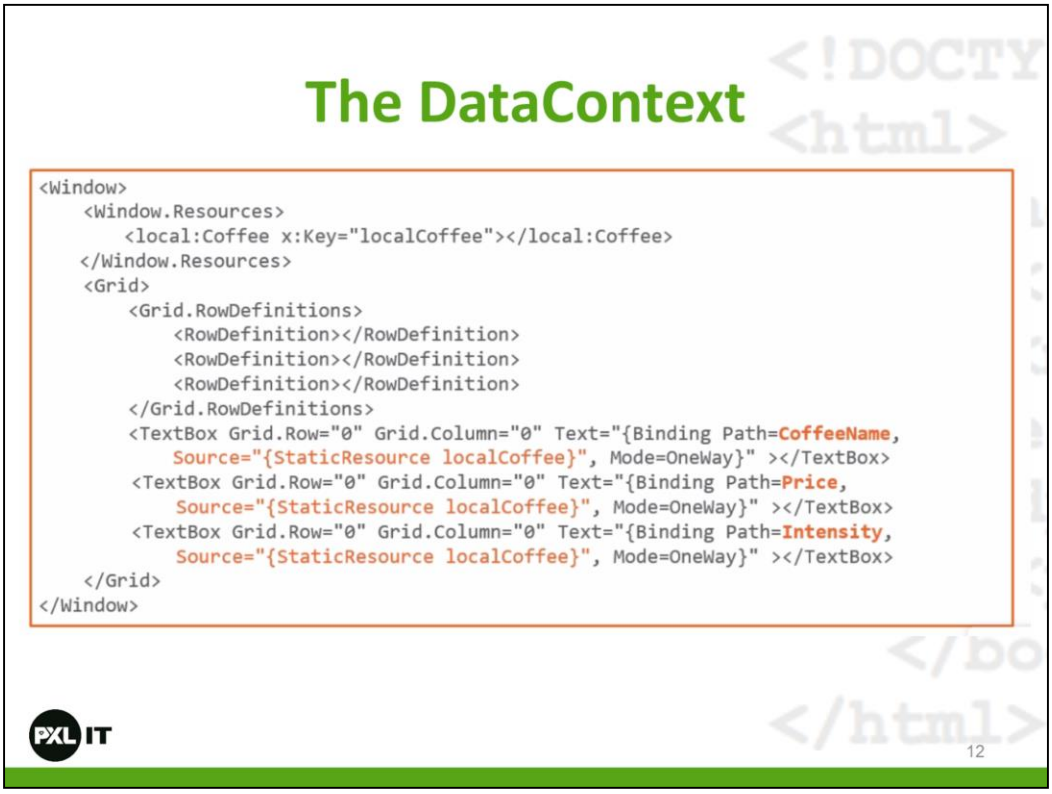
Showing details of a particular coffee



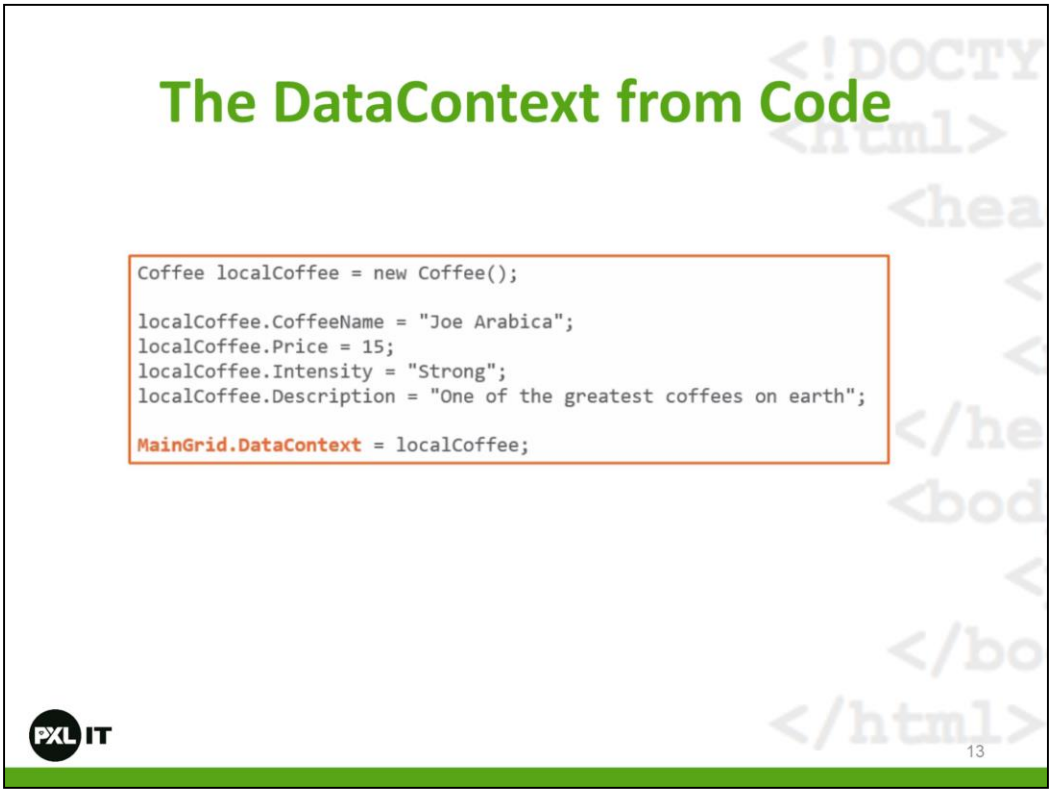
We need to create a data binding declaration. Most databindings are created declaratively in XAML, and typically we have the format you see here on the screen. You have the target control, that is number 3. We need to specify which property we are databinding, so that is number 4. Next we are creating the databinding itself and we can do so by creating a binding object. And specify the name of the property we are bind it to. And that will be number 2. Also we can pass optionally some extra binding properties. The syntax with the curly braces you see here, is called in itself a markup extension. What this means is that behind the scenes a binding object is being instantiated and it is attached an object in the xaml tree but in itself it is not part of the xaml tree. So we have here number 2, number 3, number 4 but we are missing the number one, the source of the binding. We'll come to this is just a second. First let see some applied samples of data binding statements. In this line of code here, we have a textblock and its text property is being databound. We are binding it to the property CoffeeName and we add mode=OneWay to the properties of the binding. Notice here that I'm using Path=CoffeeName. This is the default way of writing this statement. However, if the source property is the first thing that you write in your binding statement, you can omit this. As you can see here in the last line of code.



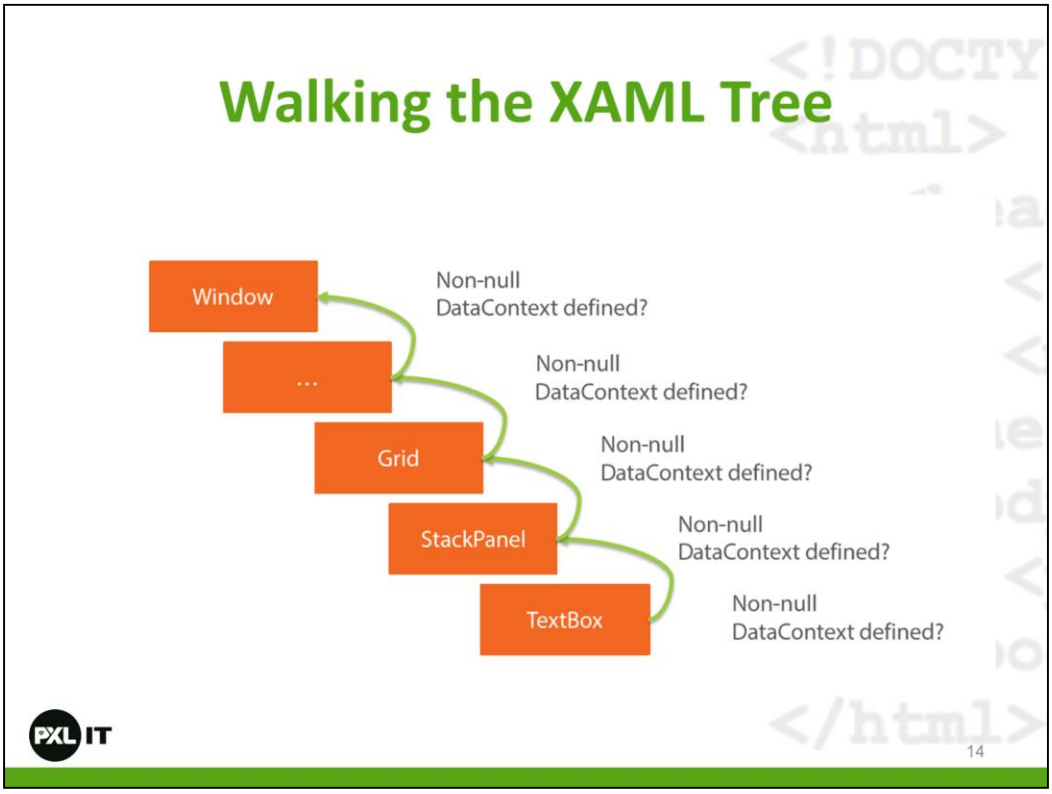
So remember that we were missing number 1, we need a source for the binding. There are several ways to add a source for the binding. Lets first see how we can bind to objects declared in XAML itself. It is possible to instantiate just any object entirely from XAML. As you can see here in the sample code. This object has being placed in the resources of the container of the UI. E.g. the window. Everything we place in the resources block, typically needs to get a key, since it's internally a dictionary. With this object created, you can now complete our databinding statement. In the binding we can simple use the source property and we refer to this object in XAML, using the staticresource markup extension. And we need to pass also the key of the objects. And while this works and will sometimes be used, it's not a very commonly used approach. Since most of the time objects won't be created from XAML, but instead, will be getting objects e.g. from a service.



Another problem we have can be seen on this slide. For each of the controls in the grid, so the 3 text boxes, we need to specify the source. So we are basically writing the same code 3 times. Of course dupplicating code is never a good practice, we all know that. However the 3 text boxes, they live in the same parent. They're all children of that surrounding gridview. Now what we can do in this case is declaring or attaching the source of the data to the surrounding parent, so this means that we don't have the source declarations on the individual controls anymore, but instead we are moving the source declaration to the common parent. And in this case we don't use the source property anymore, instead we are using the datacontext property. This property is very commonly used, will also come accross quite a few times when using MVVM. One more thing, note that is is very well possible that we declare the datacontext on the direct parent, but you can also declare it higher in the XAML tree.



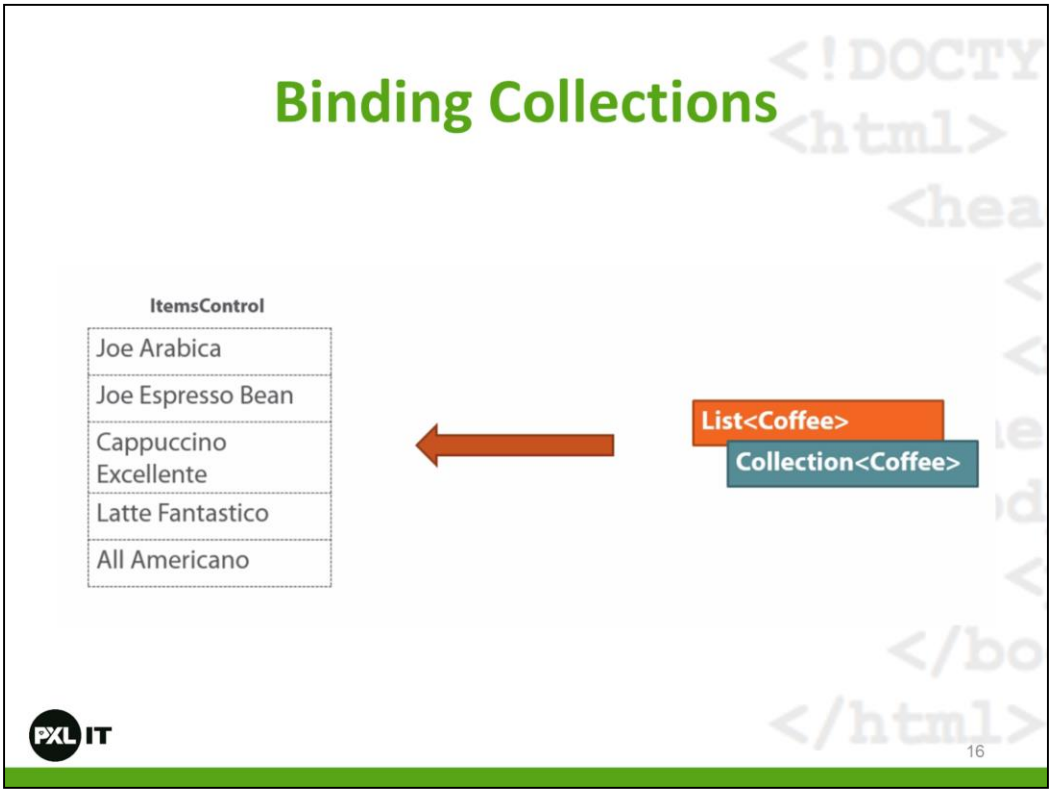
So like I said it is not very common that we declare an object in XAML and then use it as the datacontext. Very often the object being the source for the binding will be coming from a service layer and perhaps in term got the object from a repository which downloaded the data from a webservice or retrieved it from a database.Where exactly the data is coming from isn't really relevant, but it is common that we will be binding the datacontext from code. In the sample you see here on the slide we are instantiating an object in code. Like I just mentioned, this object could in fact be coming from a webservice, remember that. But anyway, take a look at the last line of the code here. We are setting the DataContext for the MainGrid, which is the grid containing the textboxex. This is how we can link an object to UI control and set it as its dataContext.



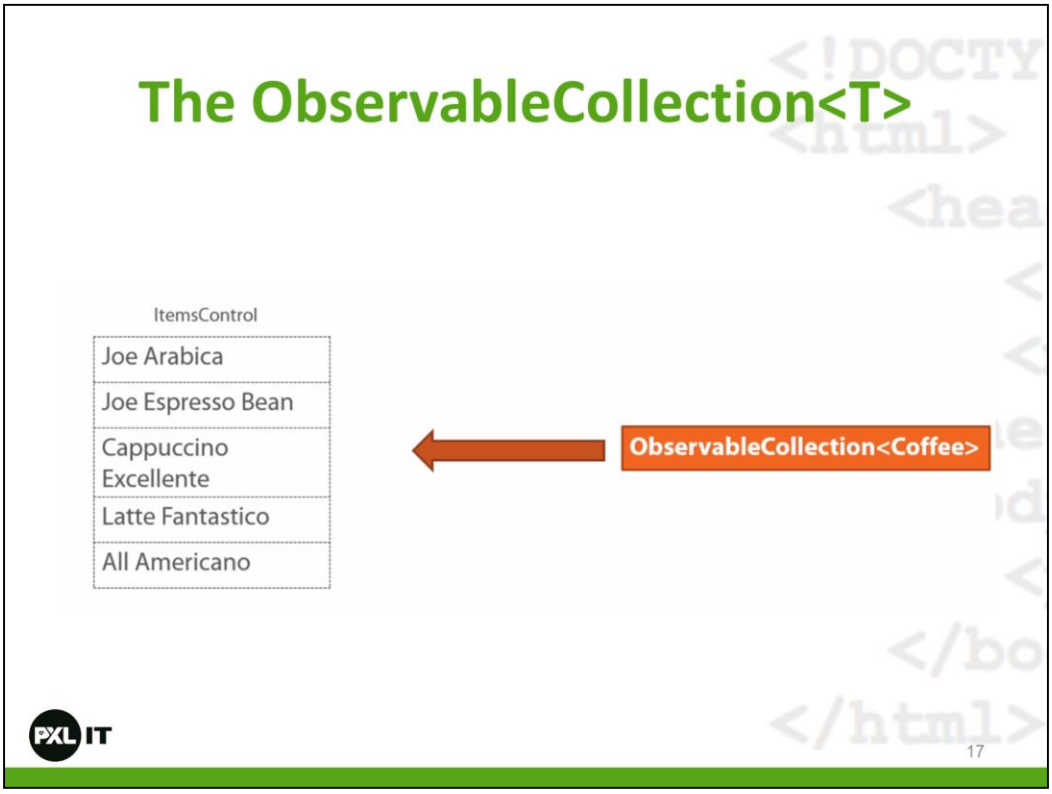
You probably know that XAML is XML and very often objects are nested. It is very common to have a XAML tree as you can see here on the slide. Take a look at the textbox at the bottom on the screen. It is a direct child of the stackpanel. This StackPanel in term is a child of the grid and this grid can also still be nested. And at the very top of the hierarchy, we have a window. In other XAML technologies this could be e.g. a page. If we have a databinding statement set on the TextBox, where no Non-Null datacontext has been specified, we'll see something happening called XAML tree walk. WPF will look in the parent of the control if a non-null datacontext has been specified on the parent. If so, this one will be used as the source of the binding. If not, it will look a level higher. All the way up untill it reaches the top level control. When it finds a non-null datacontext, it will stop going higher up in the tree. Know that, if the required properties are not found on the datacontext, it will not search further. It stops with the very first non-null instance of the datacontext.

Demo: Single Object Data Binding

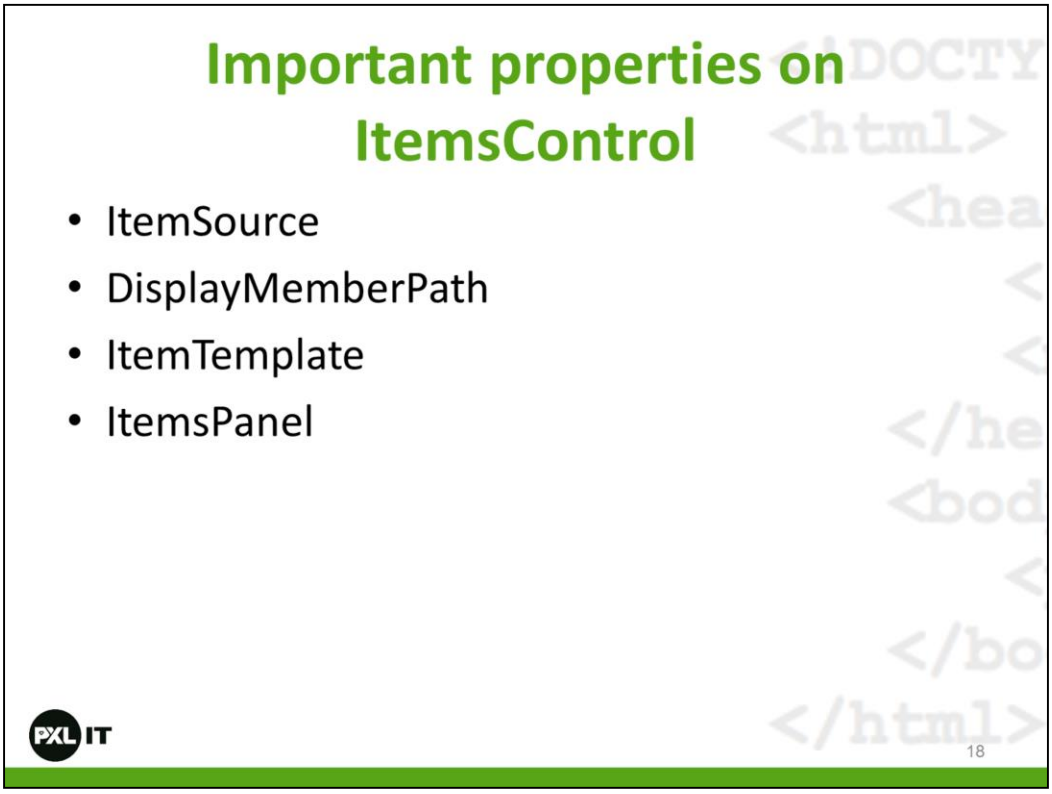
PXL IT



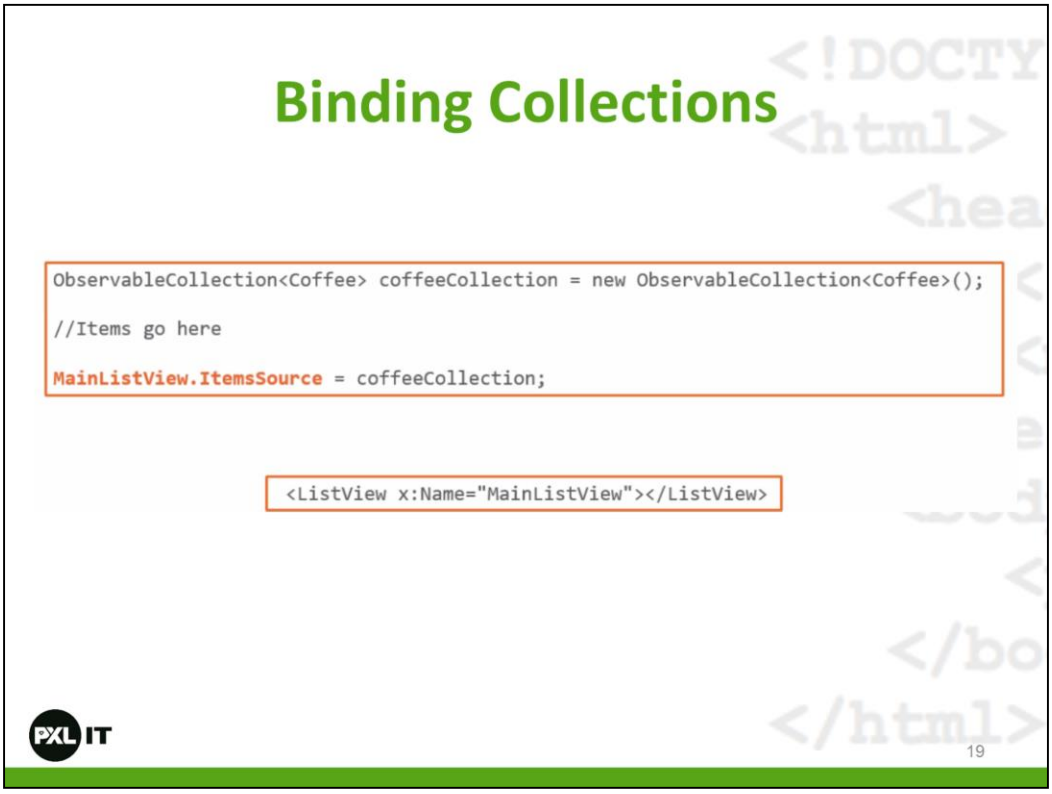
We've now seen how we can bind a single object, let shift our intention to binding to collections. Many applications have the need to display lists of data. Songs, images, context are typical examples of data which need to be displayed in lists. In our sample application, we have a list of coffees to be displayed. In code, this list of data is typically stored in a generic list of Coffees or a generic collection of coffees. Using databinding, we can directly bind these lists to an item control instance. An item control is the base class of all kind of list controls that we can use in our view. Typically, binding a list of a collection is one-way traffic. In other words, if the list is static, and you are not interessed in changes of this list reflected in the UI, than using the generic list or collection will work fine.



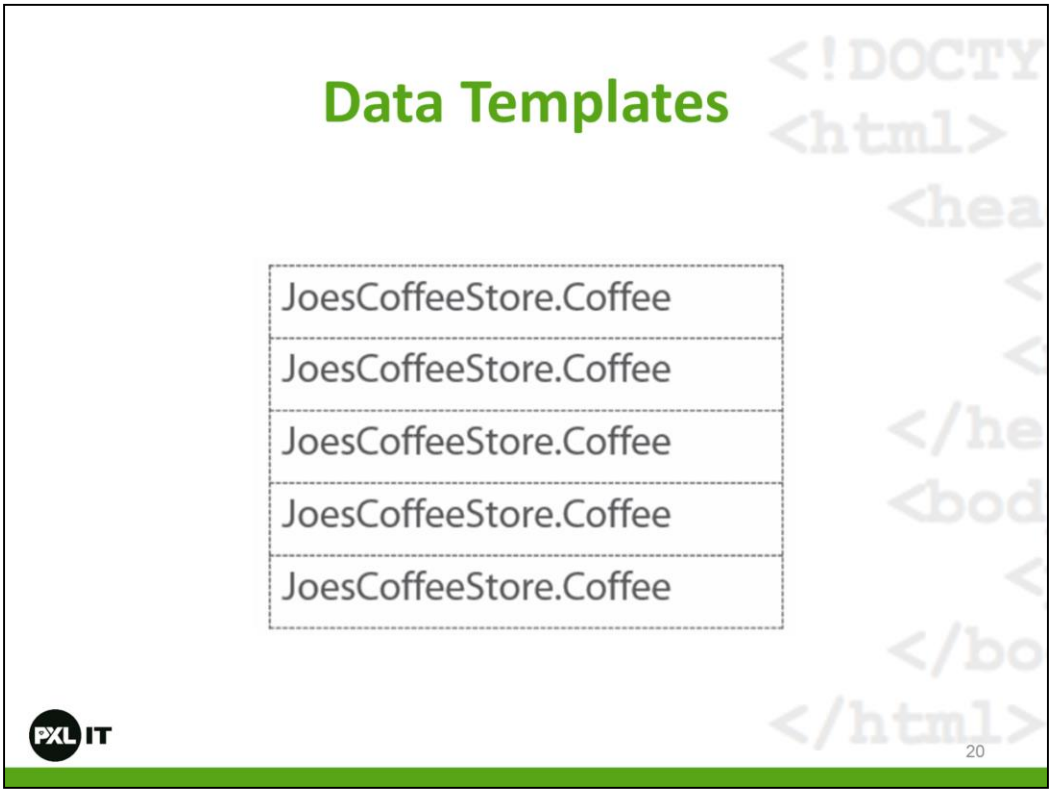
Now, if on the other hand, you would like to see updates to the collection, resulting in updates in the items control, you can use a different class, named ObservableCollection of T. We will soon learn more of this class, but for now, remember that if your databinding an observable collection to your items control, that changes to the collection, so adding or removing items, will result in the list being updated in the UI as well.



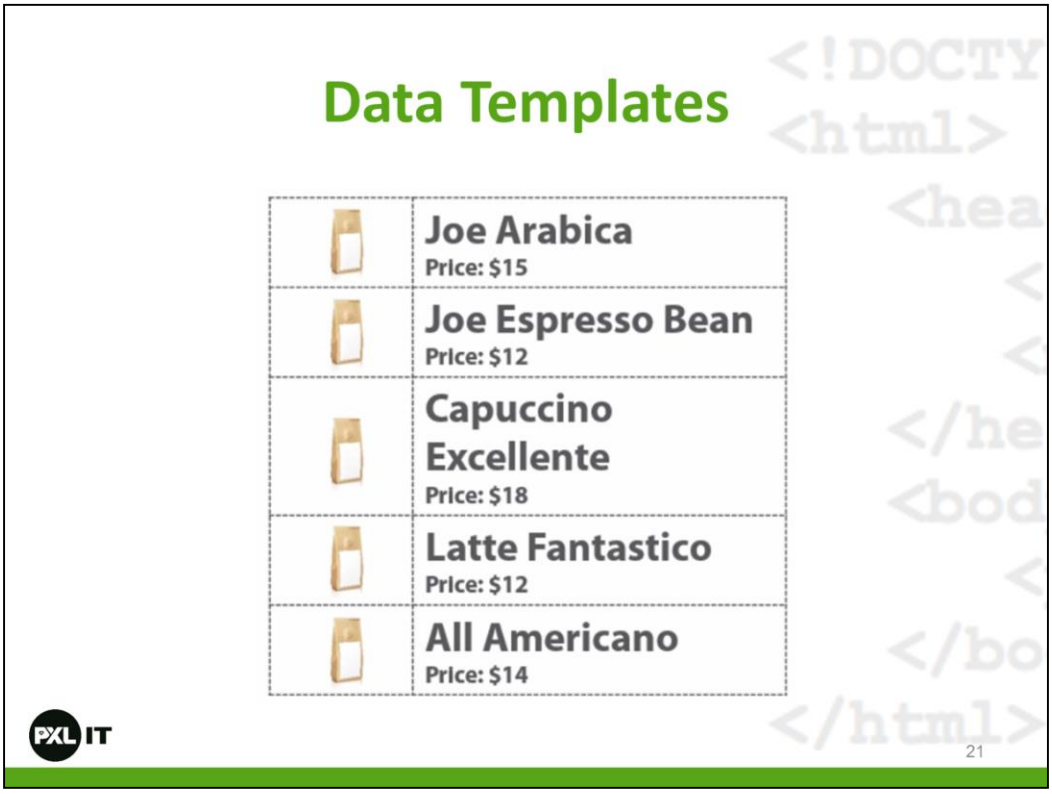
When binding a collection, there are number of important properties which take part in the data binding proces. The first one is the ItemsSource property. The itemsSource property is defined on the items control, and is therefor available on all UI list controls. Using this property, you can specify which items we want to display in our list. If we simple bind the data to our control using the itemsSource property, we'll only see the toString implementation of each object being used for the visualisation. If you want to specify which property needs to be displayed we can use the DisplayMemberPath property. Set the value to a property of the object that you are binding to, so for example point it to the name property of a Coffee instance. We'll also be discussing the next one in a couple of slides, that is the ItemTemplate. Basically, the ItemTemplate allows us to define a block of XAML, to visualise an item in the list. The itemTemplate allows us to define a block of XAML to visualise an item in a list. So we are overriding the default, which is just using a textblock. The ItemsPanel is the container for the items in the itemsControl



Lets take a look to the code for binding an observable collection of coffees to a ListView in our UI. Our ListView has been named MainListView and to bind it to our data, we are using the ItemsSource property and we set it to the observable collection, named coffeeCollection. The listview is simply declared in XAML and the name is set to MainListView as you can see on the slide.



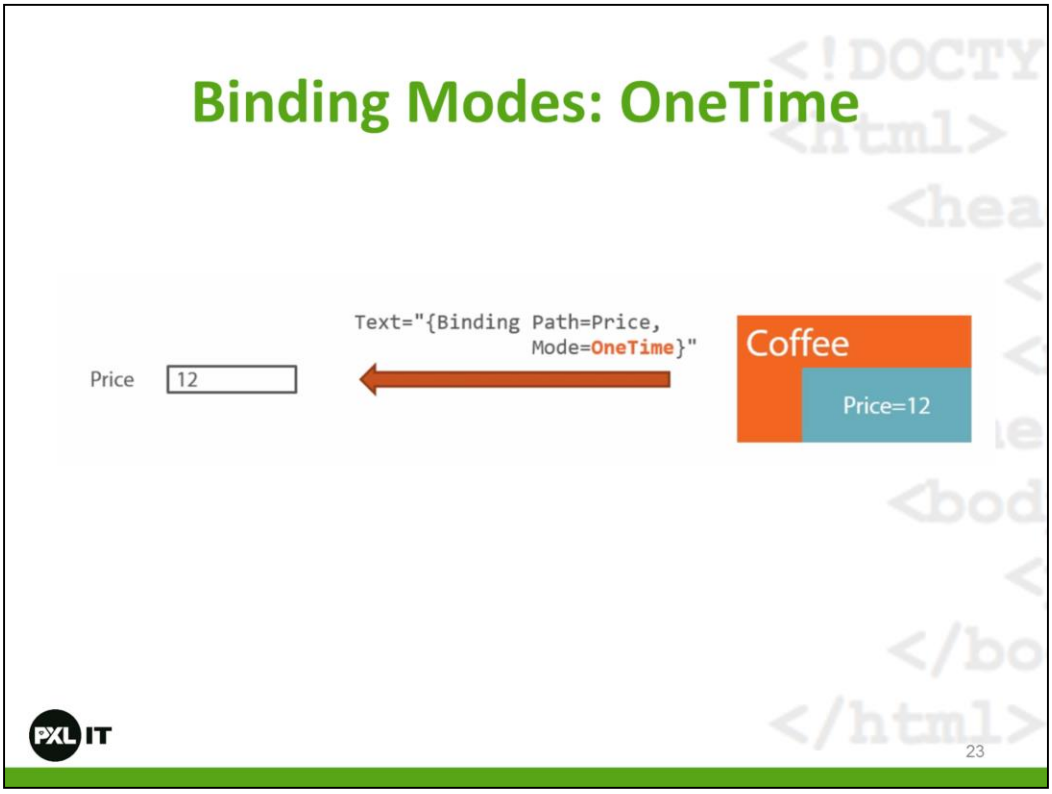
Before we look at the next demo, as promised, a word on Data Templates. When you would run the code from the previous slide, so without any specification what soever on which data the listview should be displaying, WPF would simple invoke the ToString Method on each instance. If this is not overridden in the Coffee class, you would simple see the class name being displayed. If you use the DisplayMemberPath property and we point it to a property on our object, we can visualise basically one property.



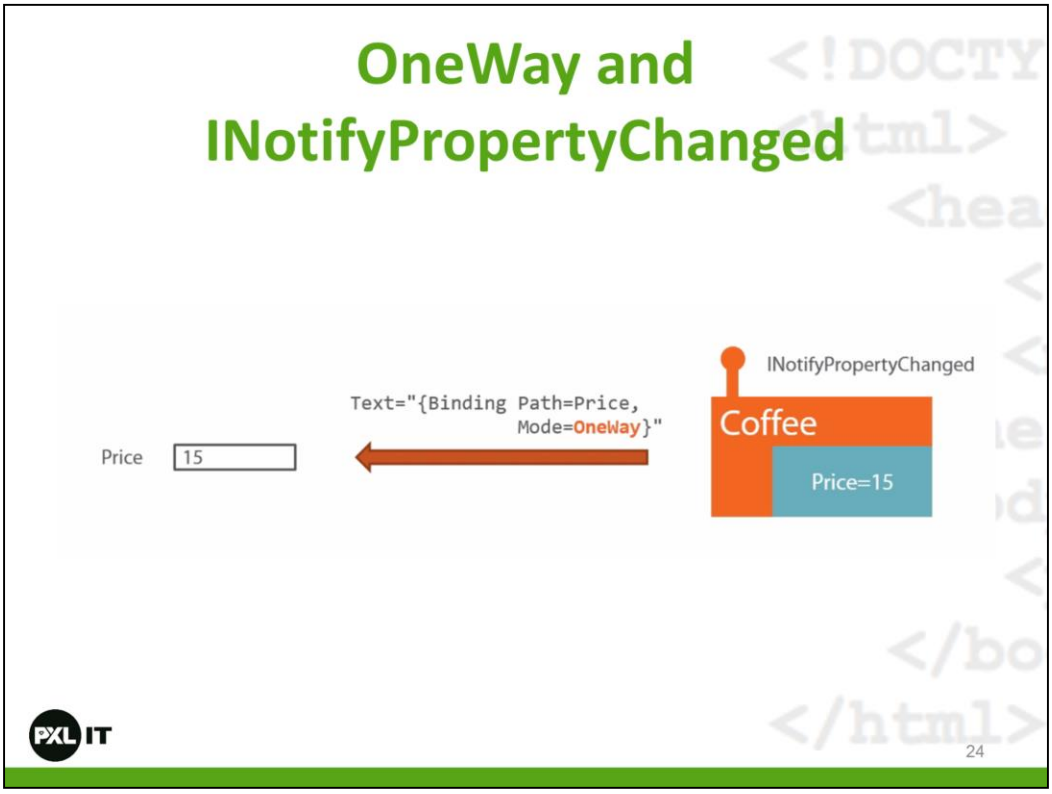
Data Templates, as mentioned, give us the abillity to completely override how an item should be displayed in the Ui. The template iteself, is the piece of XAML which is repeated for every instance in the collection. We can put whatever XAML we want there, so e.g. we can create a grid with an image and 2 textblocks.

Demo: Binding collections

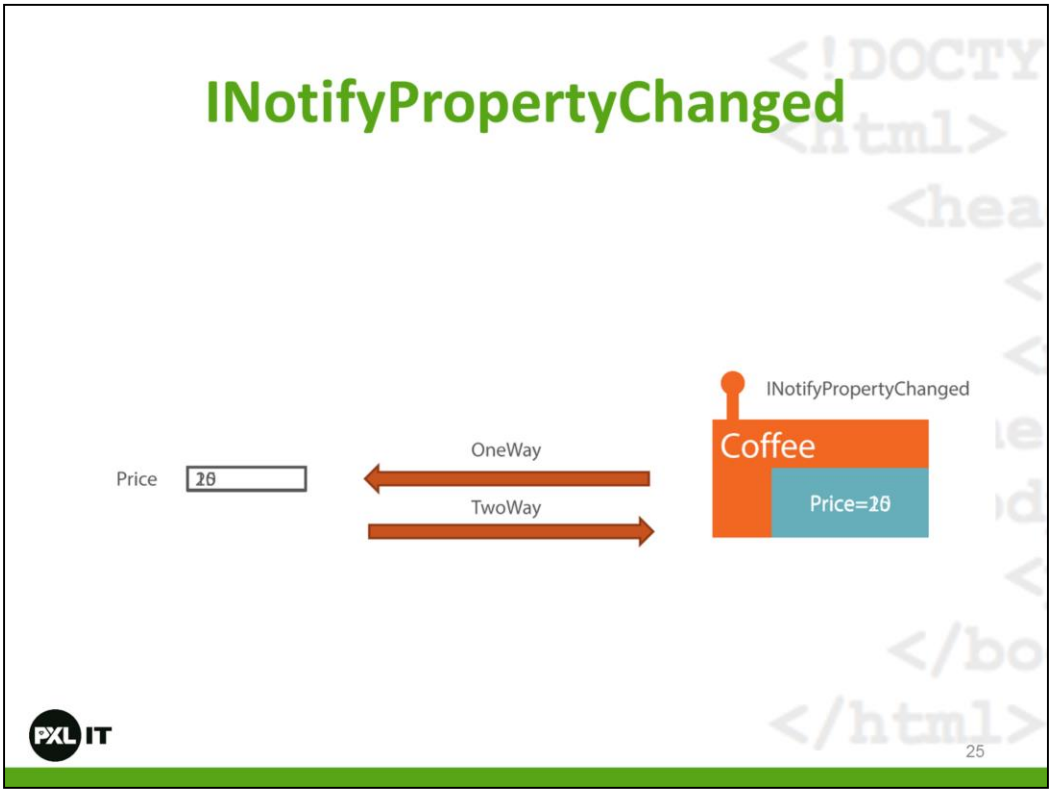
PXL IT



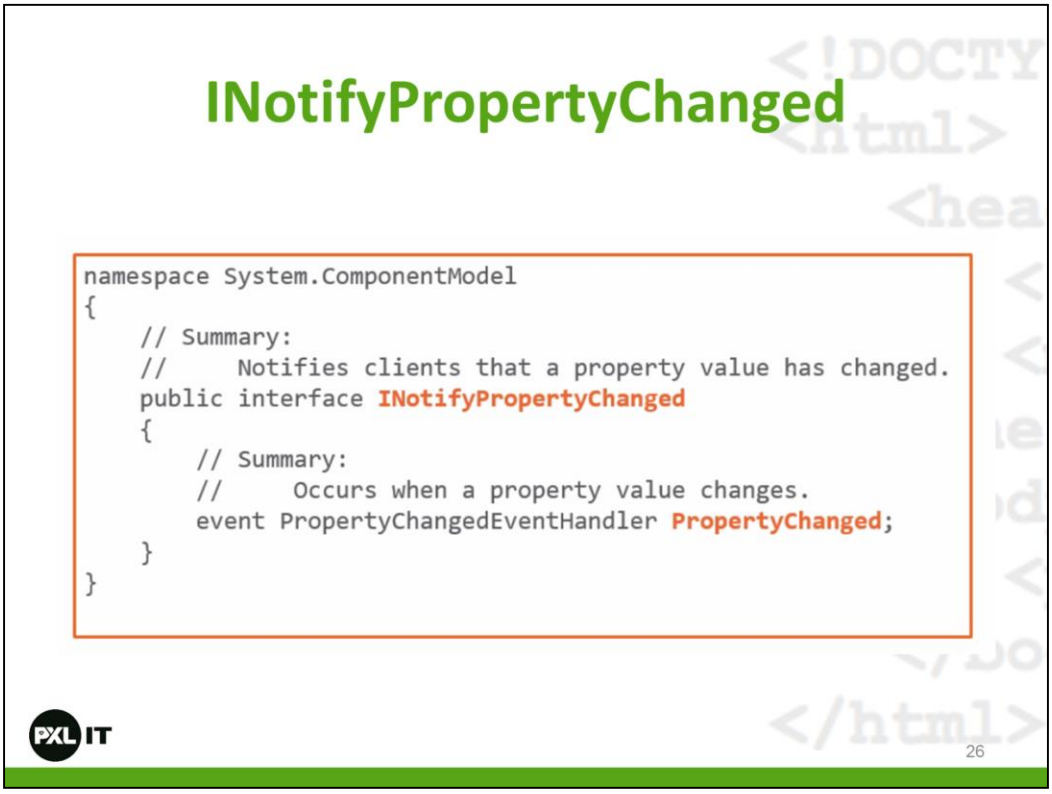
Assume that we have again an instance of our beloved coffee class, and the instance initially has set the price property to 12. Using a data binding we can get this value to display in the UI. On the binding, we can set the binding mode to One Time. A OneTime binding it is about the most basic type of binding you can have. Using this mode, the value will go once, from the source to the target. A change of the value in the source, will have no effect soever on the value being displayed in the UI.



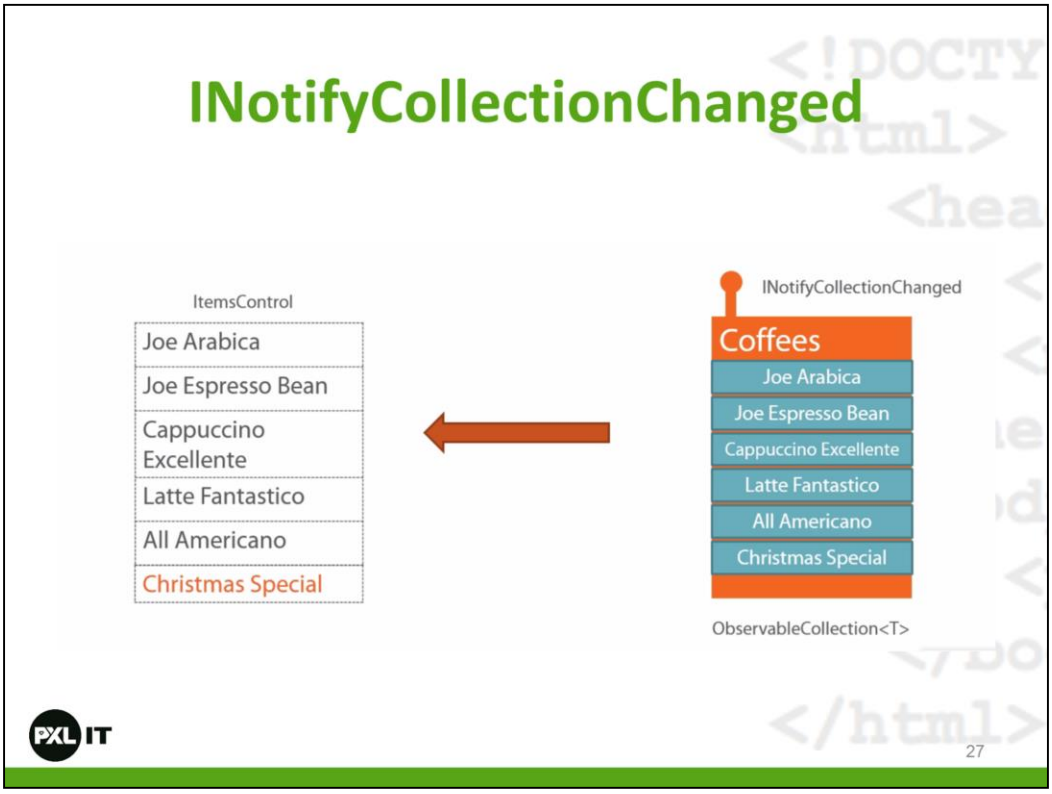
It is however possible that databinding keeps things in sync. The precondition is that our object implements a specific interface, namely the INotifyPropertyChanged interface. When the value in the source is now changed, the UI will be updated as well. We do need to mark the binding now with mode set to OneWay. Which indicates that all updates from the source should be pushed to the UI. And behind the scenes, WPF is listening for events being raised from the source object, and when this event is being raised, it will update the UI for us.



If we allow the target to update the value so e.g. the user enters a new value in the textbox than you probably want to register this value in the source as well. OneWay probably won't help us with that, considering its name. If you want to capture these values in the source object, we need to set the binding mode to TwoWay. Again for this to work, we need to implement the INotifyPropertyChanged interface on the source object.



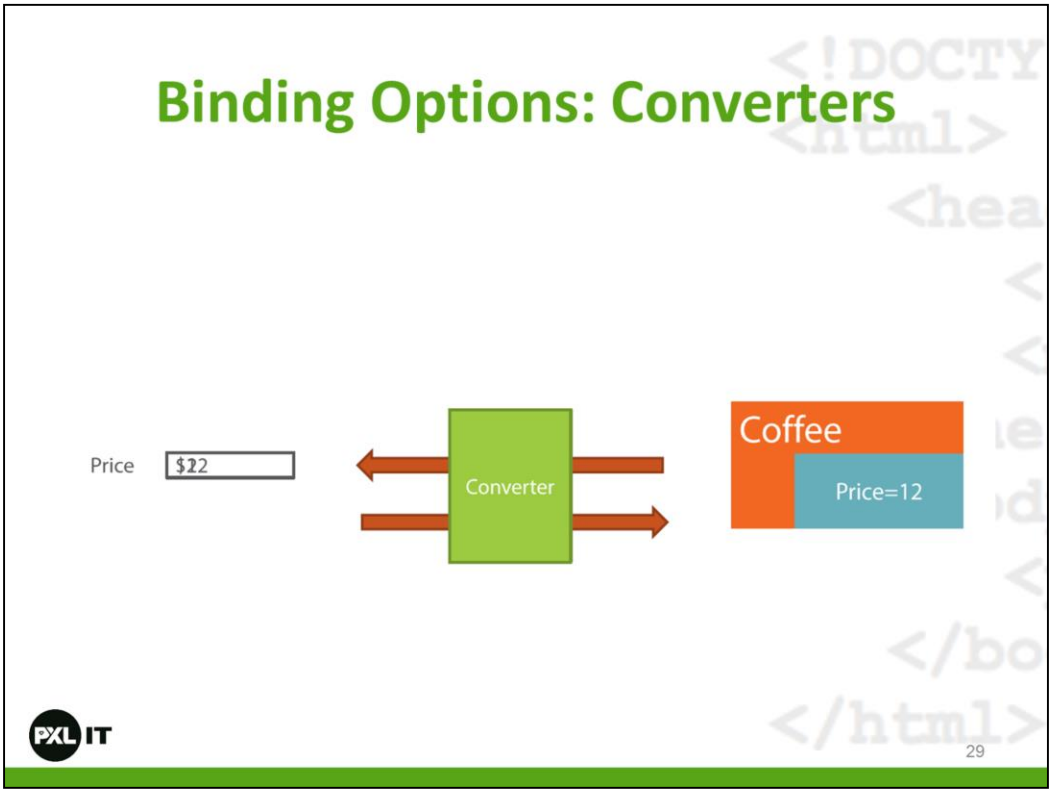
The interface that brings us al this magic and is therefor releaving us from a lot of manual coding work is actually very simple. The INotifyPropertyChanged interface lives in the System.ComponentModel namespace and only declares one event named PropertyChanged. On our classes that implement this interface we need to raise this event when the value of our property has changed. This is also the event that WPF is listening for to update the UI.



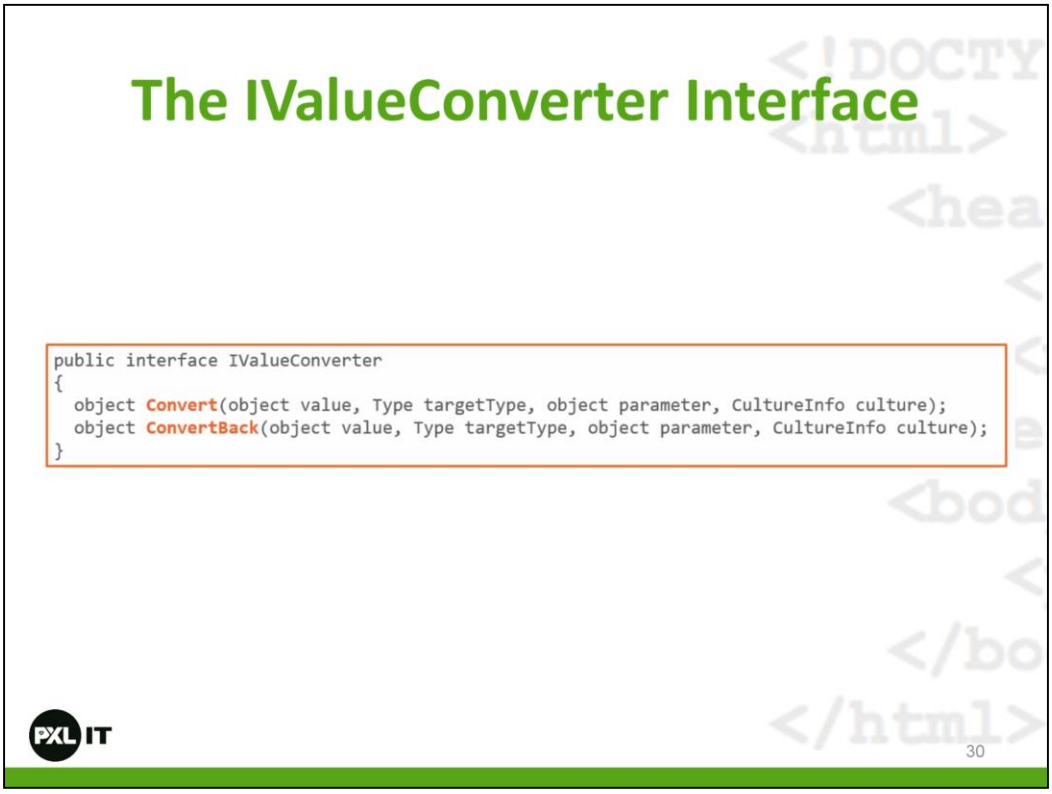
Now what if we are working with collections and we also want this automatic updating of the UI to work. A simular interface exists for working with collections, namely the INotifyCollectionChanged interface. Initially, we can make sure that the data will be displayed in the UI. We already covered that a couple of slides ago. If an item is not being added, to the collection, than the items control in the UI will be updating automatically and will be showing this newly added or removed item. Notice that changes to individual items won't be visible, because of this. For this to work, the individual items still need to implement themselves the INotifyPropertyChanged interface. In many cases, you won't be creating a collection yourself which implements the INotifyCollectionChanged interface. Implementing this interface requires quite a lot of work. Instead you'll mostly use the generic ObservableCollection which comes with .NET.

Demo: Change Management

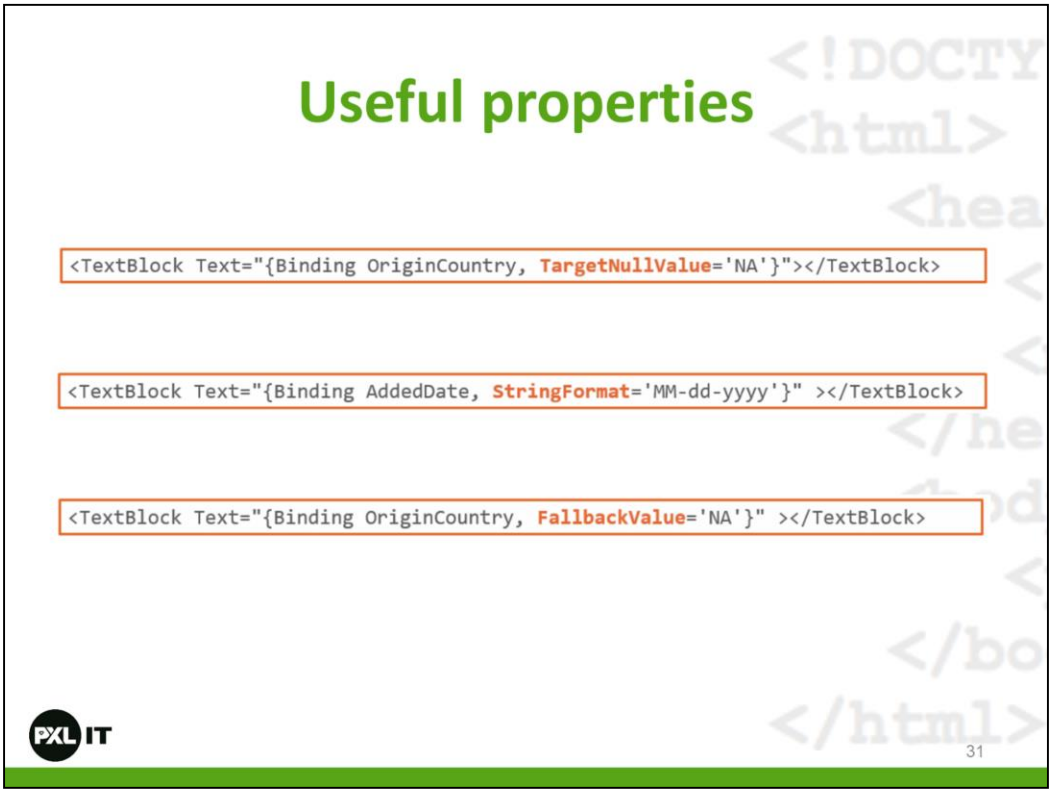
PXL IT



In the final part of this module, we'll take a look at some other options we have at our disposal to influence how the binding works. A very common used feature is a Converter. As the name implies, it will be converting something from one value to another and that's exactly what it will be doing in the case of databinding. Assume again our coffee instance with a value of the price property set to 12. Now, what exactly is that 12. We can't simple say to the user: the price is 12. Would it be €12, $12, … By default we can't specify this. Our converter can come in handy in this type of scenario, It gives us a hook in the databinding proces. You can capture the value going from the source before it reaches the target and do something with it. We can for example add a currency symbol and send the combined value to the UI. We can do many more codings like formatting dates or even convert a string into a color. The sky is the limit and we will see a couple of this in the next demo. Note that converters also work in 2 ways, just like the binding modes. Imagine that in a two way binding mode, the value entered by the user contains a currency symbol and the property on or class is of type integer. We therefore need to remove the currencty symbol using that same converter.



Technically a converter is a class which implements the IValueConverter Interface. This interface is pretty simple, it contains just 2 methods: Convert and ConvertBack. The Convert method will be called to transform or convert the data when going from the source to the target. So in the Convert method, you would typically add the currency sign or perform formatting for date values. The ConvertBack method is only used when we use a two-way binding. So to convert the data going from the target to the source. In this method you would then write code where we remove the currency sign again, so that the value then again can be stored in an integer property



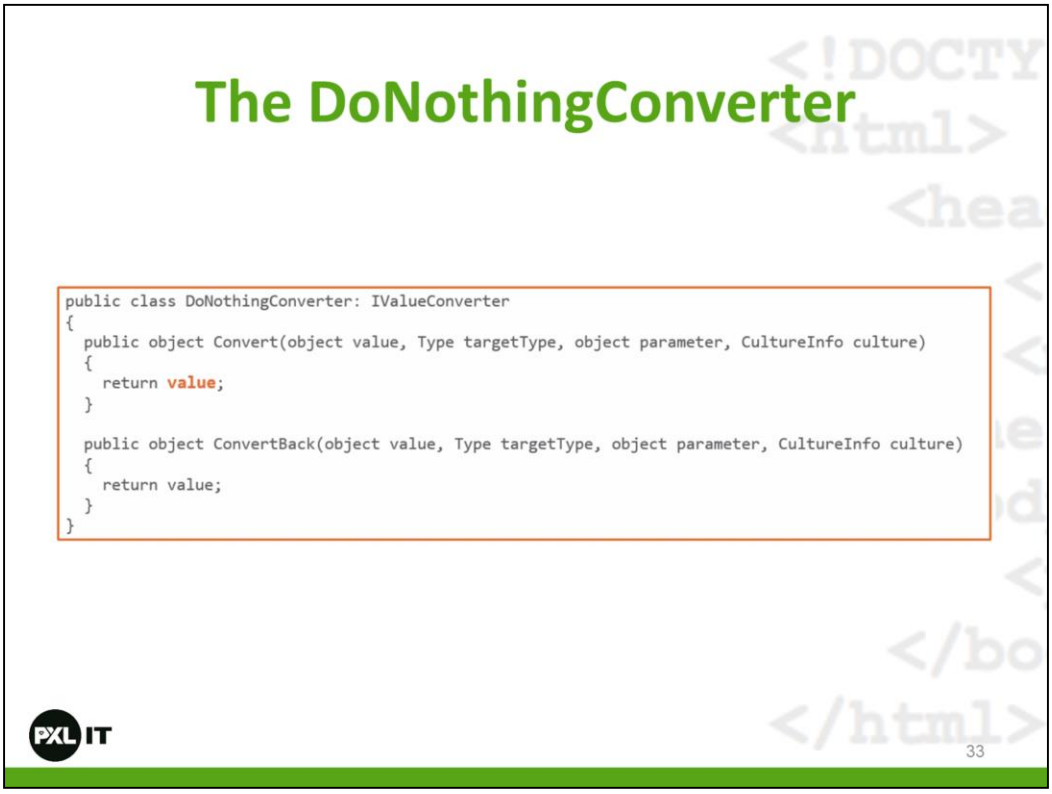
The binding class inherits from BindingBase. A class which contains som more interesting properties we can use on our bindings. The TargetNullValue property allows us to specify a value which will be displayed when the value we are binding to evaluates to null. So in essence, the target of the binding would be null. In this case here, if OriginCountry would be null on the object we're working with, we would see 'NA' being displayed in the Ui. Note that we could easily write a converter which does exactly the same thing.

Another interesting property on the BindingBase is the StringFormat property. Stringformatting is also something that we would typically do, using converters. However, the StringFormat property allows us to specify a formatting for a value such as a date, directly in the databinding statement.

Finally, the FallbackValue property allows us to specify a value which will be displayed if the binding for some reason goes wrong. Eg. Imagine that we have a class which inherits from a base class. And assume that the inheriting class adds some properties. In a databinding scenario, it can happen that we receive eighter an instance of the base class, or of the child class. In the case that we need to bind to a property defined only on the child class, and that we have an object of the base class, the binding would typically fail, and not display anything. The FallbackValue property can really be usefull here and allows us to specify the value to be displayed in this particular case.

Demo: Binding options

PXL IT

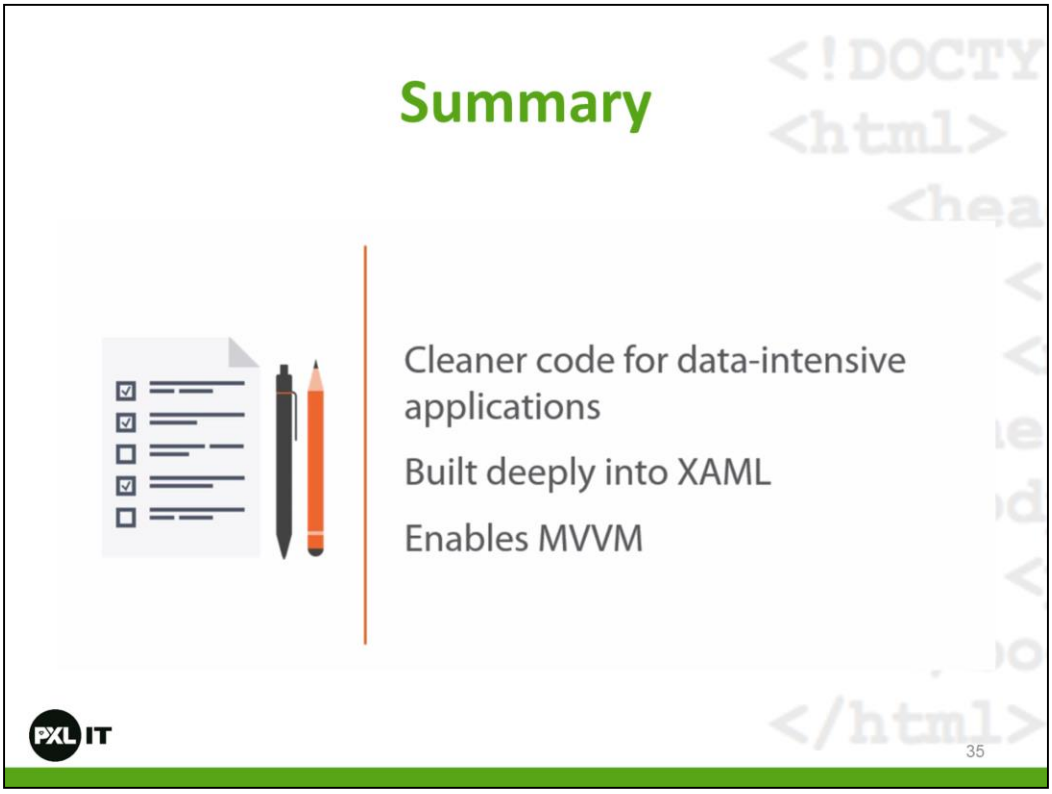


What if you want to debug your databinding statements. Some XAML technologies such as Silverlight, allow us to do exactly that by putting a breakpoint directly in the XAML code. However, others, such as WPF, do not. An often used trick is to create a converter which simple returns the received value. I call it the DoNothingConverter. If it doesn't do anything, where do we use it for. Well the converter is itself just C# code and typically in C# code, we can put a breakpoint. Using this technique, we can take a look at what data the databinding statement is getting in.

Demo: Debugging Data Binding Statements

PXL IT

34



We have reached the end of the module. In this module, we have given you an overview of the options we have with databinding in WPF, or even better, in most XAML technologies. Using databinding, we can use data-driven applications with more ease. No longer we do need to write prawn code to display and update values in the UI. The resulting code is much cleaner.

Databinding as mentioned is a XAML thing. While there are subtle differences between the different XAML technologies, it works very simularly. And we see that databinding is the enabler for MVVM, the pattern we will look at in the next module.