בס"ד

**מדריך WPF – נושאים מתקדמים**

המאמרים במסמך זה נלקחו מאתר <http://webmaster.org.il/> ונכתבו על ידי: [arikp](http://webmaster.org.il/user/11021), [מכללת סלע](http://www.sela.co.il/)

אריק פוזננסקי הוא יועץ בכיר ומרצה בסלע. הוא השלים שני תארי B.Sc. במתמטיקה ומדעי המחשב בהצטיינות יתרה בטכניון. לאריק ידע נרחב בטכנולוגיות מיקרוסופט, כולל .NET עם C#, WPF, Silverlight, WinForms, Interop, COM/ATL, C++ Win32 ו reverse engineering.

תוכן העניינים

[**שימוש ב :Data Binding מבוא** 3](#_Toc58261589)

[מה זה Data Binding ? 3](#_Toc58261590)

[עקרונות Data Binding 4](#_Toc58261591)

[**שימוש בData Binding : כיוון זרימת המידע** 5](#_Toc58261592)

[**שימוש ב :Data Binding מועד ביצוע הסנכרון** 7](#_Toc58261593)

[**שימוש ב : Data Binding הגדרת מקורות** 10](#_Toc58261594)

[הגדרת אובייקט מקור באמצעות ElementName 10](#_Toc58261595)

[הגדרת אובייקט מקור באמצעות Source 10](#_Toc58261596)

[הגדרת אובייקט מקור באמצעות DataContext 12](#_Toc58261597)

[הגדרת אובייקט מקור באמצעות RelativeSource 13](#_Toc58261598)

[**שימוש ב :Data Binding שימוש ב IValueConverter** 14](#_Toc58261599)

[**שימוש ב:Data Binding שימוש ב INotifyPropertyChange** 16](#_Toc58261600)

[דוגמא שלא עובדת 16](#_Toc58261601)

[דוגמא עובדת באמצעות שימוש ב INotifyPropertyChanged 18](#_Toc58261602)

[דוגמא עובדת באמצעות שימוש ב Dependency Properties 19](#_Toc58261603)

[**שימוש בData Binding: חיבור ל Collection** 20](#_Toc58261604)

[התכונה ItemsSource 20](#_Toc58261605)

[שינוי Collection בצורה דינמית 22](#_Toc58261606)

[**שימוש ב Data Templates** 24](#_Toc58261607)

[מה זה Data Templates? 24](#_Toc58261608)

[כיצד נגדיר Data Template? 26](#_Toc58261609)

[שימוש ב Implicit Data Templates 27](#_Toc58261610)

[**שימוש ב Dependency Properties** 28](#_Toc58261611)

[‏ ‏הקדמה 28](#_Toc58261612)

[תהליך בחירת הערך לDP 29](#_Toc58261613)

[יתרונות של Dependency Properties 30](#_Toc58261614)

[הגדרת Dependency Property במחלקה שלנו 30](#_Toc58261615)

[**WPF Dependency Property vs INotifyPropertyChanged** 31](#_Toc58261616)

[**שימוש בResources** 32](#_Toc58261617)

[כיצד נגדיר Resource? 33](#_Toc58261618)

[כיצד נשתמש ב Resource שהגדרנו? 34](#_Toc58261619)

[כיצד מתבצע תהליך החיפוש? 35](#_Toc58261620)

[**תוספות מאושרי כהן OSF – WPF נושאים מתקדמים** 36](#_Toc58261621)

[**שימוש ב Routed Events** 38](#_Toc58261622)

[הקדמה 38](#_Toc58261623)

[יתרונות Routed Events 38](#_Toc58261624)

[סוגים שונים של Routed Events 40](#_Toc58261625)

[בשביל מה צריך Style? 42](#_Toc58261626)

[הגדרה בסיסית של Style 42](#_Toc58261627)

[הגדרת של Implicit Style 43](#_Toc58261628)

[**Trigger, DataTrigger & EventTrigger** 45](#_Toc58261629)

[Property trigger 45](#_Toc58261630)

[Data triggers 46](#_Toc58261631)

[Event triggers 47](#_Toc58261632)

[Summary 47](#_Toc58261633)

[**WPF MultiTrigger and MultiDataTrigger** 48](#_Toc58261634)

[MultiTrigger 48](#_Toc58261635)

[MultiDataTrigger 48](#_Toc58261636)

[Summary 49](#_Toc58261637)

# [**שימוש ב :Data Binding**](http://webmaster.org.il/articles/wpf-data-binding) **מבוא**

מה זה Data Binding ?

**Data Binding**היא טכניקה שמאפשרת **לסנכרן בין שתי תכונות של אובייקטים שונים.**   
לרוב משתמשים בטכניקה זו בשביל לסנכרן בין אובייקט המכיל מידע (Data) לאובייקט המציג את המידע למסך (UI). לדוגמא התכונה Name של אובייקט מסוג Person יכולה להיות מסונכרנת לתכונה Text של אובייקט TextBox. הסנכרון יכול להיות דו-כיווני או חד-כיווני. לפעמים נעשה גם סנכרון בין שני פקדים כמו בדוגמא הבאה.

בדוגמא הבאה ניתן לראות שימוש פשוט ב**Data Binding** . בדוגמא זו יש לנו פקד CheckBox וכפתור ונרצה שהכפתור יהיה פעיל רק אם הCheckBox מסומן. כלומר נרצה שהתכונה IsChecked של ה CheckBox תהיה מסונכרנת לתכונה IsEnabled של הכפתור. הקטעים החשובים לצורך ה  **Data Binding**מודגשים:

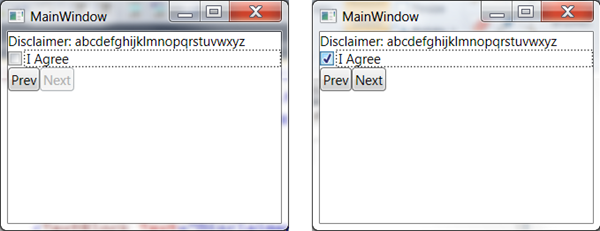
<Window x:Class="SimpleDataBinding.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
    <StackPanel>  
        <TextBlock Text="Disclaimer: abcdefghijklmnopqrstuvwxyz" />  
        **<CheckBox x:Name="checkBoxAgree"  
                 Content="I Agree"  
                 IsChecked="False" />**        <StackPanel Orientation="Horizontal">  
            <Button Content="Prev" />  
            **<Button Content="Next"  
                   IsEnabled="{Binding ElementName=checkBoxAgree, Path=IsChecked}" />**        </StackPanel>  
    </StackPanel>  
</Window>

**אובייקט היעד**

**Dependency Property**

**אובייקט המקור**

ע"י הרצה של תוכנית זו ניתן לראות שסימון ה CheckBox משנה את הכפתור לפעיל ולהפך.



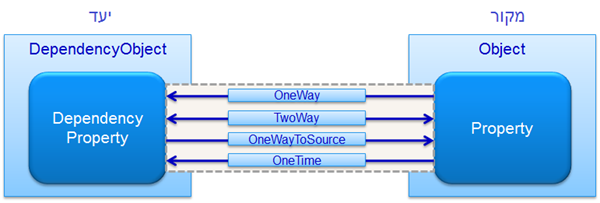
כמובן שיכולנו לעשות את הדוגמא הזאת גם ללא שימוש ב **Data Binding**, אלא ע"י טיפול באירוע **Click** של ה CheckBox וכתיבת קוד ב #C. היתרון ב **Data Binding** הוא שנחסך מאתנו לכתוב הרבה קוד שאחראי על סנכרון שכזה.

## עקרונות Data Binding

נרחיב בהמשך מסמך זה

**Binding** הוא הדבק שמחבר בין אובייקט מקור ואובייקט יעד. ולכן, בתהליך  **Binding** יש ארבעה דברים שחייבים להגדיר:

1. **אובייקט היעד** – זהו האובייקט שנרצה לשנות תכונה שלו. אובייקט זה חייב לרשת ישירות או בעקיפין מ DependencyObject. כל הפקדים בWPF הם כאלו. בדוגמא הקודמת אובייקט היעד היה ה Button.
2. **תכונת היעד** – זוהי התכונה שנרצה לשנות. תכונה זו חייבת להיות Dependency Property . למזלנו, בWPF רוב התכונות הן כאלו, לכן לרוב זוהי אינה מגבלה. בדוגמא הקודמת, תכונת היעד הייתה IsEnabled.
3. **אובייקט המקור** – זהו האובייקט שאליו היעד מתחבר. אין מגבלות על סוג האובייקט. בדוגמא הקודמת האובייקט מקור היה ה CheckBox.
4. **תכונת המקור –** זוהי התכונה שממנה נרצה לקחת את הערך ולהציבו בתכונת היעד. תכונה זו יכולה להיות גם תכונה רגילה, כלומר לא Dependency Property. בדוגמא הקודמת תכונת המקור הייתה IsChecked . נעיר כי ניתן בתכונת המקור לרשום ביטויים מורכבים המכילים גם תתי תכונות (תכונה של תכונה).



בנוסף להגדרות הללו, ניתן להגדיר אפשרויות נוספות שישפיעו על תהליך ה**Binding,** לדוגמאכיוון זרימת המידע.

# [**שימוש בData Binding : כיוון זרימת המידע**](http://webmaster.org.il/articles/wpf-data-binding-mode)

כאמור ניתן לשלוט בכיוון זרימת המידע מאובייקט המקור לאובייקט היעד. כדי לעשות זאת יש לשנות את התכונה  **Mode** של אובייקט ה **Binding** . האפשרויות לתכונה זו הן:

1. **OneTime** – הערך של תכונת המקור יועתק לתכונת היעד פעם אחת בלבד.
2. **OneWay** – הערך של תכונת המקור יועתק לתכונת היעד בכל פעם שתכונת המקור תשתנה.
3. **OneWayToSource** – הערך של תכונת היעד יועתק לתכונת המקור בכל פעם שתכונת היעד תשתנה.
4. **TwoWay** – סנכרון דו-כיווני בין תכונת המקור לתכונת היעד. כלומר, בכל פעם שתכונת המקור תשתנה, תכונת היעד תעודכן. וכן, בכל פעם שתכונת היעד תשתנה, תכונת המקור תעודכן.

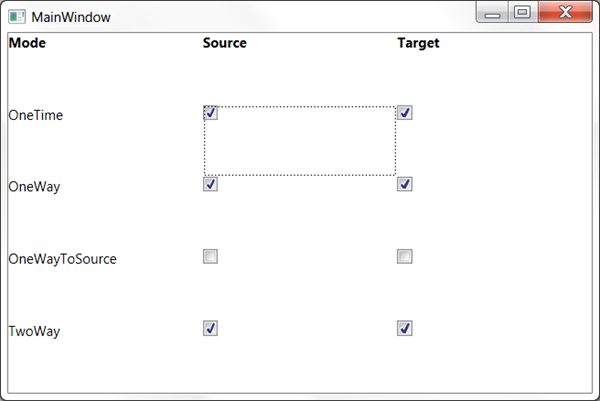
ברירת המחדל של MODE משתנה מסוג תכונה אחת לאחרת.

דוגמת הקוד הבאה מדגימה את ארבעת האפשרויות לתכונה **Mode**:

<Window x:Class="DataFlowDirections.MainWindow"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 Title="MainWindow"  
 Height="350"  
 Width="525">  
 <Grid>  
 <Grid.RowDefinitions>  
 <RowDefinition />  
 <RowDefinition />  
 <RowDefinition />  
 <RowDefinition />  
 <RowDefinition />  
 </Grid.RowDefinitions>  
   
 <Grid.ColumnDefinitions>  
 <ColumnDefinition />  
 <ColumnDefinition />  
 <ColumnDefinition />  
 </Grid.ColumnDefinitions>  
   
 <TextBlock Grid.Row="0"  
 Grid.Column="0"  
 Text="Mode"  
 FontWeight="Bold" />  
 <TextBlock Grid.Row="0"  
 Grid.Column="1"  
 Text="Source"  
 FontWeight="Bold" />  
 <TextBlock Grid.Row="0"  
 Grid.Column="2"  
 Text="Target"  
 FontWeight="Bold" />  
   
 <TextBlock Grid.Row="1"  
 Grid.Column="0"  
 Text="OneTime" />  
 <TextBlock Grid.Row="2"  
 Grid.Column="0"  
 Text="OneWay" />  
 <TextBlock Grid.Row="3"  
 Grid.Column="0"  
 Text="OneWayToSource" />  
 <TextBlock Grid.Row="4"  
 Grid.Column="0"  
 Text="TwoWay" />  
   
 <CheckBox x:Name="source1"  
 Grid.Row="1"  
 Grid.Column="1"  
 IsChecked="True" />  
 <CheckBox x:Name="source2"  
 Grid.Row="2"  
 Grid.Column="1"  
 IsChecked="True" />  
 <CheckBox x:Name="source3"  
 Grid.Row="3"  
 Grid.Column="1"  
 IsChecked="True" />  
 <CheckBox x:Name="source4"  
 Grid.Row="4"  
 Grid.Column="1"  
 IsChecked="True" />  
   
 <CheckBox x:Name="target1"  
 Grid.Row="1"  
 Grid.Column="2"  
 IsChecked="{Binding ElementName=source1, Path=IsChecked, **Mode=OneTime**}" />  
 <CheckBox x:Name="target2"  
 Grid.Row="2"  
 Grid.Column="2"  
 IsChecked="{Binding ElementName=source2, Path=IsChecked, **Mode=OneWay**}" />  
 <CheckBox x:Name="target3"  
 Grid.Row="3"  
 Grid.Column="2"  
 IsChecked="{Binding ElementName=source3, Path=IsChecked, **Mode=OneWayToSource**}" />  
 <CheckBox x:Name="target4"  
 Grid.Row="4"  
 Grid.Column="2"  
 IsChecked="{Binding ElementName=source4, Path=IsChecked, **Mode=TwoWay**}" />  
 </Grid>  
</Window>

מומלץ להעתיק תוכנית זו ולהתנסות בעצמכם איך כל אפשרות **Mode**של **Binding**מתנהגת.

מסך הפתיחה של תוכנית זו נראה כך:

[](http://webmaster.org.il/data/upl/articles2/-WPF--Data-Binding---_8E22/image%5b8%5d_2.png)

# [**שימוש ב :Data Binding מועד ביצוע הסנכרון**](http://webmaster.org.il/articles/wpf-data-binding-update-source-trigger)

אפשרות נוספת שניתנת לשליטה היא המועד שבו מתרחש סנכרון המידע. אפשרות הגיונית אחת היא לבצע סנכרון ברגע שהערך משתנה. אולם יש מקרים שבהם שיטה זו תהיה בזבזנית, חשבו על מצב שבו יש התכונה Text של TextBox מקושרת לתכונה מסוג String באובייקט נתונים, אם נבצע סנכרון בכל פעם שהמשתמש מקיש אות נוספת תתבצענה המון פעולות מיותרות. במקרה זה היה עדיף לעשות את הסנכרון רק בסוף מילוי ה TextBox , כאשר המשתמש עובר לשדה אחר.

השליטה במועד ביצוע הסנכרון נעשית באמצעות התכונה**UpdateSourceTrigger** של אובייקט ה  **Binding.** האפשרויות לתכונה זו הם:

1. **PropertyChanged**– העדכון יתבצע **בכל**שינוי בתכונת המקור.
2. **LostFocus**– העדכון יתבצע רק כאשר הפקד יאבד את הפוקוס. אפשרות זו מצוינת כאשר מבצעים סנכרון בין שדה טקסטואלי לבין TextBox..
3. **Explicit**– סנכרון ידני. העדכון יבוצע אך ורק כאשר נריץ שורת קוד ב#C שתגרום לסנכרון. אפשרות זו טובה כאשר רוצים שליטה מלאה על זמן ביצוע הסנכרון.

ברירת המחדל של **UpdateSourceTrigger** משתנה מסוג תכונה אחת לאחרת.

בדוגמת הקוד הבאה ניתן לראות שימוש לשלושת השיטות:

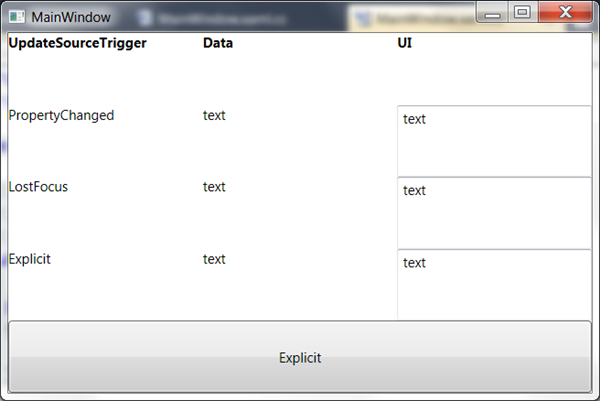
<Window x:Class="WhenToUpdate.MainWindow"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 Title="MainWindow"  
 Height="350"  
 Width="525">  
 <Grid>  
 <Grid.RowDefinitions>  
 <RowDefinition />  
 <RowDefinition />  
 <RowDefinition />  
 <RowDefinition />  
 <RowDefinition />  
 </Grid.RowDefinitions>  
   
 <Grid.ColumnDefinitions>  
 <ColumnDefinition />  
 <ColumnDefinition />  
 <ColumnDefinition />  
 </Grid.ColumnDefinitions>  
   
 <TextBlock Grid.Row="0"  
 Grid.Column="0"  
 Text="UpdateSourceTrigger"  
 FontWeight="Bold" />  
 <TextBlock Grid.Row="0"  
 Grid.Column="1"  
 Text="Data"  
 FontWeight="Bold" />  
 <TextBlock Grid.Row="0"  
 Grid.Column="2"  
 Text="UI"  
 FontWeight="Bold" />  
   
 <TextBlock Grid.Row="1"  
 Grid.Column="0"  
 Text="PropertyChanged" />  
 <TextBlock Grid.Row="2"  
 Grid.Column="0"  
 Text="LostFocus" />  
 <TextBlock Grid.Row="3"  
 Grid.Column="0"  
 Text="Explicit" />  
   
 <TextBlock x:Name="data1"  
 Grid.Row="1"  
 Grid.Column="1"  
 Text="text" />  
 <TextBlock x:Name="data2"  
 Grid.Row="2"  
 Grid.Column="1"  
 Text="text" />  
 <TextBlock x:Name="data3"  
 Grid.Row="3"  
 Grid.Column="1"  
 Text="text" />  
   
 <TextBox x:Name="ui1"  
 Grid.Row="1"  
 Grid.Column="2"  
 Text="{Binding ElementName=data1, Path=Text, UpdateSourceTrigger=PropertyChanged, Mode=TwoWay}" />  
 <TextBox x:Name="ui2"  
 Grid.Row="2"  
 Grid.Column="2"  
 Text="{Binding ElementName=data2, Path=Text, UpdateSourceTrigger=LostFocus, Mode=TwoWay}" />  
 <TextBox x:Name="ui3"  
 Grid.Row="3"  
 Grid.Column="2"  
 Text="{Binding ElementName=data3, Path=Text, UpdateSourceTrigger=Explicit, Mode=TwoWay}" />  
   
 <Button Grid.Row="4"  
 Grid.Column="0"  
 Grid.ColumnSpan="3"  
 Content="Explicit"  
 Click="Button\_Click" />  
 </Grid>  
</Window>

קוד ה#C שמתלווה לתוכנית זו הוא:

using System.Windows;  
using System.Windows.Controls;  
   
namespace WhenToUpdate  
{  
 /// <summary>  
 /// Interaction logic for MainWindow.xaml  
 /// </summary>  
 public partial class MainWindow : Window  
 {  
 public MainWindow()  
 {  
 InitializeComponent();  
 }  
   
 private void Button\_Click(object sender, RoutedEventArgs e)  
 {  
 ui3.GetBindingExpression(TextBox.TextProperty).UpdateSource();  
 }  
 }  
}

שימו לב לשימוש בפקודה **UpdateSource**, זו הפקודה שגורמת לסנכרון ידני.

תוצאת הרצת התוכנית נראית כך:

[](http://webmaster.org.il/data/upl/articles2/-WPF--Data-Binding_8E6B/image%5b11%5d_2.png)

כאשר נשנה את ה TextBox בשורה הראשונה (PropertyChanged) הData ישתנה בכל לחיצה. לעומת זאת, בשורה השנייה (LostFocus), הData ישתנה רק כאשר נעזוב את הTextBox. בשורה השלישית (Explicit) הData לא יתעדכן כל עוד לא לחצנו על הכפתור Explicit .  
מומלץ לכתוב בעצמכם את התוכנית ולהתנסות בהבדלים.

# [**שימוש ב : Data Binding הגדרת מקורות**](http://webmaster.org.il/articles/wpf-data-binding-sources)

כזכור, בעת יצירת  **Binding**יש להגדיר ארבעה חלקים:

1. **אובייקט היעד** – זהו האובייקט שנרצה לשנות תכונה שלו
2. **תכונת היעד** – זוהי התכונה שנרצה לשנות
3. **אובייקט המקור** – זהו האובייקט שאליו היעד מתחבר
4. **תכונת המקור –** זוהי התכונה שממנה נרצה לקחת את הערך ולהציבו בתכונת היעד

את אובייקט המקור ניתן להגדיר בארבע שיטות שונות, כאשר לכל שיטה יש שימושים משלה.

הגדרת אובייקט מקור באמצעות ElementName

השימוש בתכונה **ElementName** הוא הדרך הפשוטה ביותר להגדיר אובייקט מקור, ואכן כל הדוגמאות שראינו עד כה עשו שימוש ב **ElementName** להגדרת מקור.

**ElementName**מאפשר להגדיר פקד UI בתור המקור של פקד היעד, לפיכך השימוש ב **ElementName** מצוין כאשר נרצה לחבר בין פקדים (UI to UI).

כדי להשתמש בשיטה זו יש להוסיף להגדרת ה **Binding** את התכונה  **ElementName** ולאחריה את שם האלמנט שאליו אנו מעוניינים להתחבר. שם האלמנט הוא כזה שהוגדר באמצעות **x:Name** בלבד.

לדוגמא, הקוד הבא גורם לחיבור של התכונה Text של TextBox בשם textColor לתכונה Background של אותו פקד! מה שיקרה הוא שברגע שנכתוב שם של צבע חוקי בתוך הפקד, הרקע שלו ישתנה לצבע שנכתוב.

<TextBlock Text="Using ElementName " />  
<StackPanel Orientation="Horizontal">  
 <TextBlock Text="Color: " />  
 <TextBox x:Name="textColor"  
  **Background= "{Binding ElementName=textColor, Path=Text}"** />  
</StackPanel>

[מדריך WPF – שימוש ב Data Binding: הגדרת מקורות](http://webmaster.org.il/data/upl/articles2/-WPF--Data-Binding--_8CC4/image%5b14%5d_2.png)

Binding באמצעות ElementName מקשר בין שני פקדים גרפים (UI to Ui), ומה אם נרצה לקשר בין פקד גרפי לאובייקט מחלקה שלנו? לצורך כך, נכיר עכשיו את 2 השיטות הבאות:

* הגדרת אובייקט מקור באמצעות Source
* הגדרת אובייקט מקור באמצעות DataContext – שימושי יותר

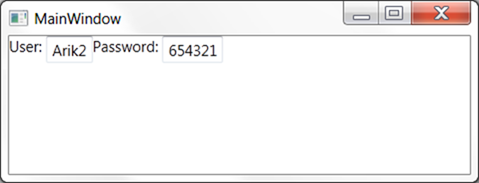
הגדרת אובייקט מקור באמצעות Source

באמצעות התכונה  **Source** של אובייקט  **Binding** ניתן להגדיר באופן מפורש מי האובייקט מקור. בשיטה זו אנו מציבים בתכונה  **Source** ממש את האובייקט עצמו (ולא את שמו כמו בשיטה הקודמת). ניתן לעשות זאת ע"י שימוש בתגית StaticResource כאשר אובייקט המקור מוגדר בתוך Resource, או אפילו בקוד. נראה כיצד לעשות זאת באמצעות StaticResource.

בדוגמא הבאה נרצה לחבר בין חלון שמכיל פקדים עבור שם משתמש וסיסמא לבין אובייקט נתונים שיכיל מידע זה.

Source

לימוד עצמי

[](http://webmaster.org.il/data/upl/articles2/-WPF--Data-Binding--_8CC4/image%5b17%5d_2.png)

לשם כך ראשית נוסיף מחלקה חדשה לפרויקט בשם MyData שתייצג את אובייקט הנתונים שלנו:

namespace SourceDemo  
{  
 public class MyData  
 {  
 public string User { get; set; }  
 public string Password { get; set; }  
 }  
}

כעת נרצה להשתמש במחלקה זו בXAML, לצורך כך נצטרך להוסיף הגדרת xmlns חדשה שממפה בין ה .NET namespace לבין ה XML namespace. עושים זאת ע"י הוספת השורה הבאה לקובץ הXAML:

xmlns:local="clr-namespace:SourceDemo"

כעת נוכל להגדיר אובייקט מטיפוס MyData בתוך קובץ הXAML, באזור ה Resources, ולהשתמש בו כמקור ע"י שימוש בתכונה **Source**  של ה **Binding**:

<Window x:Class="SourceDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       xmlns:local="clr-namespace:SourceDemo"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <StackPanel>  
    <StackPanel Orientation="Horizontal">  
      <StackPanel.Resources>  
        <local:MyData x:Key="mySource"  
                     User="Arik2"  
                     Password="654321" />  
      </StackPanel.Resources>  
      <TextBlock Text="User: " />  
      <TextBox Text="{Binding Source={StaticResource mySource}, Path=User}" />  
      <TextBlock Text="Password: " />  
      <TextBox Text="{Binding Source={StaticResource mySource}, Path=Password}" />  
    </StackPanel>  
  </StackPanel>  
</Window>

שימו לב שאנו רושמים באופן מפורש את אובייקט המקור בכל פעם שאנו מבצעים Binding, בדוגמא שלנו אנו עושים זאת פעמיים, גם עבור הקישור ל User וגם עבור Password.

הגדרת מקור בדרך זו נראית הגיונית, אך תכף נראה דרך הרבה יותר שימושית.

הגדרת אובייקט מקור באמצעות DataContext

השימוש בתכונה **DataContext** מאוד דומה לשימוש בתכונה  **Source**, אך הרבה יותר שימושי. גם בתכונה**DataContext** יש להציב את האובייקט המבוקש בתור מקור. ההבדל המרכזי הוא שהתכונה **DataContext** שייכת לפקד יעד עצמו ובנוסף היא תכונה שעוברת בירושה לפקדים המוכלים. ולכן אם נרצה לחבר מספר פקדים לאותו אובייקט מידע (כמו בדוגמא הקודמת) אזי מספיק שנגדיר את ה **DataContext**במקום מרכזי יותר בעץ הפקדים, לדוגמא בפקד הראשון (לרוב מסוג Window).

בדוגמא הבאה אנו עושים שימוש באותה מחלקה MyData כמו מקודם, אבל הפעם נציב מופע שלה לתכונה DataContext בעזרת קוד. ראשית נראה את קוד הXAML:

<Window x:Class="DataContextDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <StackPanel x:Name="stackPanel"  
             Orientation="Horizontal">  
    <TextBlock Text="User: " />  
    <TextBox Text="**{Binding Path=User}**" />  
    <TextBlock Text="Password: " />  
    <TextBox Text="**{Binding Path=Password}**" />  
  </StackPanel>  
</Window>

נשים לב שלא הגדרנו כלל את המקור בקוד הXAML. קביעת מקור הנתונים נעשית באמצעות שורת הקוד הבאה:

using System.Windows;  
   
namespace DataContextDemo  
{  
  /// <summary>  
  /// Interaction logic for MainWindow.xaml  
  /// </summary>  
  public partial class MainWindow : Window  
  {  
    public MainWindow()  
    {  
      InitializeComponent();  
   
      MyData myData = new MyData()  
      {  
        User = "Arik",  
        Password = "123456"  
      };  
      **stackPanel.DataContext = myData;**

**שינוי דרך ה GUI ישפיע על האובייקט MyData.**

**אך לא להיפך!!**

**ואם נרצה שגם האובייקט יודיע ל GUI?**

**השדות של MyData יצטרכו להיות מוגדרים כ DependencyProperties**

**או שהמחלקה MyData תממש ממשק INotifyPropertyChanged.**

**או שנצטרך שוב לקרוא ל אותן שורות קוד שמשמאל של יצירת אובייקט וקביעת הקשר הנתונים שלו.**

**מיד נרחיב...**

    }  
  }  
}

כדי לקבוע את המקור אנו שמים את האובייקט בתוך התכונה **DataContext** של הפקד StackPanel ( ולא אובייקט Binding כלשהוא), התכונה **DataContext** מועברת בירושה לפקדים שמתחת ל StackPanel וכך מגיעה גם לשני ה TextBox שעושים בה שימוש.

הגדרת אובייקט מקור באמצעות RelativeSource

RelativeSource

לימוד עצמי

השיטה האחרונה לקביעת אובייקט מקור היא אמנם קצת פחות נפוצה אך גם לה יש מקום. קביעת אובייקט מקור הנמצא בעץ הפקדים במיקום יחסי לפקד היעד. לדוגמא ניתן לקבוע את שאובייקט היעד יהיה בעצמו אובייקט המקור (כדי לסנכרן שתי תכונות של אותו אובייקט), עשינו זאת בדוגמא קודמת בעזרת **ElementName**. נראה כיצד ניתן לעשות זאת באמצעות **RelativeSource**.

קוד הXAML הנדרש לשם כך הוא כדלהלן:

<Window x:Class="RelativeSourceDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <StackPanel Orientation="Horizontal">  
    <TextBlock Text="Color: " />  
    <TextBox Background="{Binding RelativeSource={RelativeSource Mode=Self}, Path=Text}" />  
  </StackPanel>  
</Window>

בדוגמא זו קבענו את מוד העבודה של **RelativeSource**  להיות **Self**, כלומר אובייקט המקור הינו אובייקט היעד. ניתן להשתמש באפשרויות נוספות כגון  **FindAncestor**. אפשרות זו מאפשרת להגדיר כמה שלבים יש לטפל בעץ הפקדים כדי למצוא את אובייקט המקור המבוקש. הסבר מפורט יותר על התכונה  **RelativeSource**ניתן למצוא [בתיעוד](http://msdn.microsoft.com/en-us/library/system.windows.data.relativesource.aspx" \t "_blank).



# [**שימוש ב :Data Binding שימוש ב IValueConverter**](http://webmaster.org.il/articles/wpf-data-binding-ivalueconverter)

לשנה הבאה להעביר את נושא זה לעמוד 10.

ישנם פעמים בהם נרצה לעשות  **Binding**בין תכונת יעד ותכונת מקור מטיפוסים שונים. לדוגמא, נרצה לעשות תוכנית שבה סימון CheckBox מעלימה כפתור  ואילו הורדת הסימון מציגה אותו. לשם כך יש לבצע קישור בין התכונות **Visibility** של הכפתור מטיפוס Visibility לבין תכונת **IsChecked** של ה CheckBox מטיפוס Boolean.

בדוגמא הבאה אנו מנסים לבצע סנכרון שכזה:

<Window x:Class="ValueConverterDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <StackPanel>  
    <Button Content="button"  
           Visibility="{Binding ElementName=checkbox, Path=IsChecked}" />  
    <CheckBox x:Name="checkbox"  
             Content="hide button?" />  
  </StackPanel>  
</Window>

הרצה של דוגמא זו תדווח על שגיאה "שקטה" במהלך ביצוע ה **Binding** ( תופיע הודעה בחלון ה **Debug Output**), הבעיה היא שהטיפוסים שונים ולכן לא ברור כיצד לעשות את הסנכרון.

כדי לבצע את ההמרה בין הטיפוסים יש להגדיר מחלקה חדשה המממשת את הממשק  **IValueConverter**. מחלקה זו תאפשר לנו להוסיף לוגיקה משלנו בעת ביצוע ה **Binding**. הפונקציה המרכזית שיש לממשק זה נקראת  **Convert**. פונקציה זו נקראת כאשר מתבצע **Binding**  ממקור כלשהוא ליעד כלשהוא. פונקציה זו מקבלת מספר פרמטרים כאשר החשוב ביניהם נקרא Value מטיפוס object והוא הערך המקורי שיש להמיר. הערך המוחזר מהפונקציה  **Convert** הוא הערך לאחר שעבר המרה.

במקרה שלנו נרצה להמיר את הערך המתקבל מ Boolean לערך מטיפוס Visibility. נעשה זאת ע"י הוספת מחלקה חדשה שתממש **IValueConverter**  באופן הבא:

using System;  
using System.Globalization;  
using System.Windows;  
using System.Windows.Data;  
   
namespace ValueConverterDemo  
{  
  public class NotBooleanToVisibilityConverter : IValueConverter  
  {  
    public object Convert(  
      object value,  
      Type targetType,  
      object parameter,  
      CultureInfo culture)  
    {  
      bool boolValue = (bool)value;  
      if (boolValue)  
      {  
        return Visibility.Collapsed;  
      }  
      else  
      {  
        return Visibility.Visible;  
      }  
    }  
   
    public object ConvertBack(  
      object value,  
      Type targetType,  
      object parameter,  
      CultureInfo culture)  
    {  
      throw new NotImplementedException();  
    }  
  }  
}

הפונקציה השנייה שיש לממש בממשק  **IValueConverter** היא הפונקציה **ConvertBack**  שעובד באותה צורה, רק שהיא ממירה מערך היעד לערך מקור ולכן צריכה לעשות את הפעולה ההפוכה למה שהתבצע ב **Convert**. פונקציה זו רלוונטית רק כאשר מבצעים  **Binding** בעל Mode מטיפוס TwoWay או OneWayToSource. בשאר המקרים ניתן לוותר על מימוש פונקציה זו.

כעת לאחר שיצרנו את המימוש שלנו ל  **IValueConverter** נרצה להשתמש בו.   
ראשית יש לבצע מיפוי של .NET namespace ל XML namespace באופן הבא:

xmlns:local="clr-namespace:ValueConverterDemo"

וכעת ניצור מופע של המחלקה שלנו ב Resources של החלון:

<Window.Resources>  
  <local:NotBooleanToVisibilityConverter   
   x:Key="notBooleanToVisibilityConverter" />  
</Window.Resources>

נותר רק להוסיף את הפרמטר  **Converter** ל  **Binding** שהגדרנו מקודם:

  <StackPanel>  
    <Button Content="button"  
           Visibility="{Binding ElementName=checkbox,   
                                Path=IsChecked,   
                                Converter={StaticResource notBooleanToVisibilityConverter}}" />  
    <CheckBox x:Name="checkbox"  
             Content="hide button?" />  
  </StackPanel>

הקוד המלא של התוכנית מופיע להלן. מומלץ להריץ בעצמכם ולהתנסות בדוגמא:

<Window x:Class="ValueConverterDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       xmlns:local="clr-namespace:ValueConverterDemo"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <Window.Resources>  
    <local:NotBooleanToVisibilityConverter   
     x:Key="notBooleanToVisibilityConverter" />  
  </Window.Resources>  
  <StackPanel>  
    <Button Content="button"  
           Visibility="{Binding ElementName=checkbox,   
                                Path=IsChecked,   
                                Converter={StaticResource notBooleanToVisibilityConverter}}" />  
    <CheckBox x:Name="checkbox"  
             Content="hide button?" />  
  </StackPanel>  
</Window>

# [**שימוש ב:Data Binding שימוש ב INotifyPropertyChange**](http://webmaster.org.il/articles/wpf-data-binding-inotifypropertychanged)

בואו ננסה לחשוב לרגע על הסוגיה הבאה: כיצד מנגנון ה **Binding**יודע על כל שינוי באובייקט המקור?

כדי שזה יקרה אנחנו צריכים שיהיה איזשהו מנגנון של הודעות או אירועים כך שבכל פעם שיש שינוי באובייקט המקור, מנגנון ה**Binding** מקבל את האירוע ובעקבותיו מפעיל את תהליך הסנכרון לתוך היעד (ובאופן דומה לכיוון ההפוך).

**אז מי מספק את אותם אירועים בכל פעם תכונת מקור משתנה? זה תלוי.**

אם תכונת המקור היא מסוג **Dependency Property** אזי דיווח על שינויים הוא אחד הדברים שתכונה שכזו מוסיפה, כלומר מנגנון ה **Binding** משתמש בעובדה שתכונת המקור היא **Dependency Property.**

אם תכונת המקור היא תכונה רגילה אזי אין דרך למנגנון ה **Binding**לדעת על השינוי!

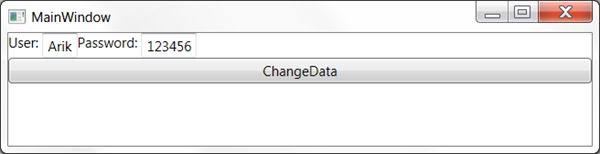
## דוגמא שלא עובדת

בדוגמא הבאה יש לנו מחלקת נתונים בשם MyData הממומשת באופן הבא:

namespace NotifyTargetBad  
{  
  public class MyData  
  {  
    public string User { get; set; }  
    public string Password { get; set; }  
  }  
}

וכן נגדיר ממשק משתמש המכיל שתי תיבות טקסט וכפתור שיגרום לשינוי המידע, להלן הגדרת החלון:

<Window x:Class="NotifyTargetBad.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <Grid>  
   
    <Grid.RowDefinitions>  
      <RowDefinition Height="Auto" />  
      <RowDefinition Height="Auto" />  
    </Grid.RowDefinitions>  
   
    <StackPanel x:Name="stackPanel"  
               Orientation="Horizontal"  
               Grid.Row="0">  
      <TextBlock Text="User: " />  
      <TextBox Text="{Binding Path=User}" />  
      <TextBlock Text="Password: " />  
      <TextBox Text="{Binding Path=Password}" />  
    </StackPanel>  
   
    <Button Grid.Row="1"  
           Content="ChangeData"  
           Click="Button\_Click" />  
   
  </Grid>  
</Window>



והנה הקוד שגורם לאתחול הנתונים הראשון ולשינוי הנתונים כאשר לוחצים על הכפתור:

using System.Windows;  
   
namespace NotifyTargetBad  
{  
    /// <summary>  
    /// Interaction logic for MainWindow.xaml  
    /// </summary>  
    public partial class MainWindow : Window  
    {  
        private MyData \_myData;  
   
        public MainWindow()  
        {  
            InitializeComponent();  
   
            \_myData = new MyData()  
            {  
                User = "Arik",  
                Password = "123456"  
            };  
            stackPanel.DataContext = \_myData;  
        }  
   
        private void Button\_Click(object sender, RoutedEventArgs e)  
        {  
            \_\_myData.Password = "555555";

**שינוי דרך ה GUI ישפיע על האובייקט MyData.**

**אך לא להיפך!!**

**ואם נרצה שגם האובייקט יודיע ל GUI?**

**השדות של MyData יצטרכו להיות מוגדרים כ DependencyProperties**

**או שהמחלקה MyData תממש ממשק INotifyPropertyChanged.**

**מיד נרחיב...**

        }  
    }  
}

הרצת הדוגמא הזו תראה לכם שלחיצה על הכפתור לא משנה שום דבר בפקדי ה UI למרות שהנתון של הסיסמא השתנה. הסיבה היא שהתכונות שיצרנו באובייקט MyData הן תכונות פשוטות (לא Dependency Properties) ולכן מנגנון ה Binding אינו יכול לדעת על השינוי!

## דוגמא עובדת באמצעות שימוש ב INotifyPropertyChanged

כדי לגרום למנגנון ה Binding לדעת על שינויים בתכונות של אובייקט מקור יש לדאוג לכך שאובייקט המקור יממש את הממשק  **INotifyPropertyChanged**. ממשק זה מגדיר אירוע אחד בלבד בשם **PropertyChanged** שלאובייקט המקור צריך להיות. כמובן שלא מספיק שיהיה אירוע אלא צריך לדאוג להפעיל אותו בכל פעם שתכונה של האובייקט משתנה.

להלן הקוד שמראה את הדרך הנכונה לכתוב את המחלקה MyData :

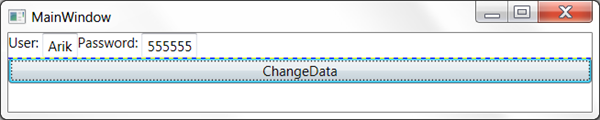
using System.ComponentModel;  
   
namespace NotifyTargetUsingInterface  
{  
  public class MyData : **INotifyPropertyChanged**  
  {  
    private string \_user;  
    public string User  
    {  
      get  
      {  
        return \_user;  
      }  
      set  
      {  
        \_user = value;  
**if (PropertyChanged != null)  
        {  
          PropertyChanged(this, new PropertyChangedEventArgs("User"));  
        }**      }  
    }  
   
    private string \_password;  
    public string Password  
    {  
      get  
      {  
        return \_password;  
      }  
      set  
      {  
        \_password = value;  
        **if (PropertyChanged != null)  
        {  
          PropertyChanged(this, new PropertyChangedEventArgs("Password"));  
        }**      }  
    }  
   
**public event PropertyChangedEventHandler PropertyChanged;**  }  
}

**שינוי דרך ה GUI ישפיע על האובייקט MyData.**

**וכעת גם להיפך!**

נעיר כי נאלצנו להחליף את ה Properties האוטומטיים מהדוגמא הקודמת בProperties מלאים, בעלי משתנה פרטי מפורש כדי שנוכל לכתוב קוד בתוך המימוש שלהם.

שאר הקוד בתוכנית נותר ללא שינוי אבל כעת לחיצה על הכפתור מעדכנת את השדות שהשתנו!



## דוגמא עובדת באמצעות שימוש ב Dependency Properties

כמובן שיכולנו לעשות גם שימוש בתכונות מסוג  **Dependency Properties**ואז להנות מדיווח אוטומטי על שינויים בתכונות. לשם כך יש לשנות את המחלקה MyData כך שתירש מ DependencyObject ותהיה בעלת שתי תכונות מתאימות מסוג Dependency Property :

using System.Windows;  
   
namespace NotifyTargetUsingDependecyProperties  
{  
  public class MyData : DependencyObject  
  {  
    public string User  
    {  
      get { return (string)GetValue(UserProperty); }  
      set { SetValue(UserProperty, value); }  
    }  
   
    public static readonly DependencyProperty UserProperty =  
        DependencyProperty.Register("User",   
        typeof(string), typeof(MyData), new UIPropertyMetadata(""));  
   
    public string Password  
    {  
      get { return (string)GetValue(PasswordProperty); }  
      set { SetValue(PasswordProperty, value); }  
    }  
   
    public static readonly DependencyProperty PasswordProperty =  
        DependencyProperty.Register("Password",   
        typeof(string), typeof(MyData), new UIPropertyMetadata(""));  
  }  
}

**שינוי דרך ה GUI ישפיע על האובייקט MyData.**

**וכעת גם להיפך!**

אמנם בשיטה זו מקבלים בחינם את הדיווח על שינויים (ועוד תכונות טובות של **Dependency** **Properties**) אבל אנו משלמים על כך שאנו חייבים לרשת מאובייקט **DependencyObject**  ישירות או בעקיפין. ישנם הרבה מקרים שבהם זה בלתי אפשרי (בגלל שניתן לרשת רק ממחלקה אחת) ואז הפתרון הוא להשתמש ב **INotifyPropertyChanged**.

# [**שימוש בData Binding: חיבור ל Collection**](http://webmaster.org.il/articles/wpf-data-binding-collection)

עד כה ראינו כיצד לחבר בין פקד אחד לפריט מידע אחד. בפרק זה נראה כיצד נוכל לחבר בין פקד שיודע להציג רשימות כמו **ListBox**  או  **ComboBox** לבין רשימה של אובייקטים.

הערה: דוגמת הקוד במודל מדגימה גם **ListBox**  וגם  **ComboBoxוממחישה טוב יותר מאשר הדוגמא במסך שמדגימה רק ListBox**   **.**

## התכונה ItemsSource

התכונה  **ItemsSource** משותפת לכל הפקדים שמסוגלים להתחבר לרשימה של אובייקטים. תכונה זו מקבלת את אובייקט הרשימה, ישירות או ע"י שימוש ב  **Binding**. לאחר מכן, הפקד שבשימוש יוצר אלמנט גרפי עבור כל איבר ברשימה.

לדוגמא, להראות חיבור של ListBox לרשימה של אובייקטים.

ראשית נוסיף מחלקה חדשה בשם MyData שתייצג ישות מידע בודדת ברשימה:

namespace CollectionDemo  
{  
  public class MyData  
  {  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
    public bool IsLecturer { get; set; }  
  }  
}

כעת ניצור פקד שיודע להציג רשימות, ונחבר אותו לרשימה שניצור ב code-behind. בנוסף נוסיף כפתור שתפקידו יהיה להוסיף איבר חדש לרשימה:

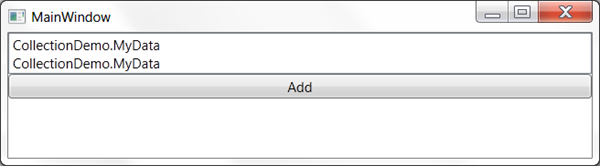
<Window x:Class="CollectionDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <StackPanel>  
    <ListBox ItemsSource="{Binding}" />  
    <Button Content="Add"  
           Click="Button\_Click" />  
  </StackPanel>  
</Window>

נשים לב לשימוש המיוחד ב  **Binding** ללא פרמטרים. מאחר ואין מקור מפורש (ElementName, Source, RelativeSource) אזי ברירת המחדל תהיה לחפש את מקור הנתונים ב **DataContext**  של האובייקט ListBox. בנוסף לא הוגדר **Path**  לכן במקום להתחבר תכונה מסוימת של אובייקט המקור, החיבור יהיה לכל האובייקט.

ההצבה של אובייקט ל  **DataContext** נעשית בקוד #C באופן הבא:

using System.Collections.Generic;  
using System.Windows;  
   
namespace CollectionDemo  
{  
  /// <summary>  
  /// Interaction logic for MainWindow.xaml  
  /// </summary>  
  public partial class MainWindow : Window  
  {  
    private List<MyData> \_myCollection = new List<MyData>();  
   
    public MainWindow()  
    {  
      InitializeComponent();  
   
      DataContext = \_myCollection;  
      \_myCollection.Add(  
        new MyData  
        {  
          FirstName = "Arik",  
          LastName = "Poznanski",  
          IsLecturer = true  
        });  
      \_myCollection.Add(  
        new MyData  
        {  
          FirstName = "John",  
          LastName = "Smith",  
          IsLecturer = false  
        });  
    }  
   
    private int counter = 0;  
    private void Button\_Click(object sender, RoutedEventArgs e)  
    {  
      ++counter;  
      \_myCollection.Add(  
          new MyData()  
          {  
            FirstName = "item " + counter,  
            LastName = "item " + counter,  
            IsLecturer = counter % 3 == 0  
          });  
    }  
  }  
}

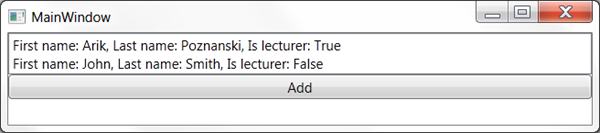
הרצה של תוכנית זו תציג את הפלט הבא:



הבעיה הראשונה שעולה מקוד זה היא שהאלמנטים מטיפוס MyData לא מוצגים בצורה טובה. מאחר והמחלקה MyData איננה אלמנט גרפי, מנגנון הציור של WPF קורא לפונקציה ToString של האובייקט ומציג את התוצאה בתוך TextBlock. נראה בהמשך המדריך כיצד לשפר משמעותית את ההצגה הגרפית של אובייקטים מטיפוס MyData. בינתיים נסתפק בהעמסה של האופרטור ToString של MyData:

public override string ToString()  
{  
  return string.Format("First name: {0}, Last name: {1}, Is lecturer: {2}",   
    FirstName, LastName, IsLecturer);  
}

כעת התוצאה נראית כך:



הבעיה השנייה נגלית רק כאשר לוחצים על כפתור Add ומגלים ששום דבר לא קורה. כלומר, הפונקציה אכן נקראת ומוסיפה איבר לרשימה, אבל ה ListBox לא מתעדכן עם המידע החדש!

הבעיה דומה לבעיה שכבר נתקלנו בה בפרק קודם. ה **ListBox** אינו יודע שה Collection שאליו אנו מחוברים השתנה ולכן אינו יודע ליצור אלמנט חדש עבור האיבר החדש שהצטרך לרשימה. הפתרון לבעיה זו הוא שה Collection שאיתו אנו עובדים יממש ממשק בשם **INotifyCollectionChange.** שימו לב, זהו לא אותו ממשק כמו שהיה מקודם INotifyPropertyChanged. המטרה פה היא לדווח על שינויים ב Collection ולא בתכונה כזו או אחרת.

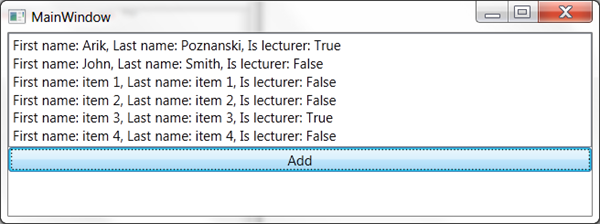
## שינוי Collection בצורה דינמית

כדי לגרום לכך ששינויים ב Collection שאליו אנו מחוברים יגיעו לפקד הרשימה יש לדאוג שה Collection יממש את הממשק INotifyCollectionChange. למזלנו, לא צריך לממש מחלקות של רשימות משלנו. מיקרוסופט מספקת לנו מימוש כזה בצורת המחלקה הגנרית: ObservableCollection. מחלקה זו מממשת כבר את הממשק הנ"ל ומדווחת על שינויים. כל שנותר לעשות הוא להשתמש במחלקה הזו במקום השימוש שלנו במחלקה הגנרית List.

להלן הקוד המתוקן:

using System.Collections.ObjectModel;  
using System.Windows;  
   
namespace CollectionDemo  
{  
  /// <summary>  
  /// Interaction logic for MainWindow.xaml  
  /// </summary>  
  public partial class MainWindow : Window  
  {  
    private ObservableCollection<MyData> \_myCollection =   
      new ObservableCollection<MyData>();  
   
    public MainWindow()  
    {  
      InitializeComponent();  
   
      DataContext = \_myCollection;  
      \_myCollection.Add(  
        new MyData  
        {  
          FirstName = "Arik",  
          LastName = "Poznanski",  
          IsLecturer = true  
        });  
      \_myCollection.Add(  
        new MyData  
        {  
          FirstName = "John",  
          LastName = "Smith",  
          IsLecturer = false  
        });  
    }  
   
    private int counter = 0;  
    private void Button\_Click(object sender, RoutedEventArgs e)  
    {  
      ++counter;  
      \_myCollection.Add(  
          new MyData()  
          {  
            FirstName = "item " + counter,  
            LastName = "item " + counter,  
            IsLecturer = counter % 3 == 0  
          });  
    }  
  }  
}

כעת לחיצה על כפתור Add  עובדת והתוצאה לאחר מספר לחיצות היא:



# [**שימוש ב Data Templates**](http://webmaster.org.il/articles/wpf-data-templates)

מה זה Data Templates?

**Data Template** הוא מנגנון המאפשר ליצור בזמן ריצה רכיבי UI מתוךData . לדוגמא, בפרק הקודם ראינו כיצד ניתן לחבר פקד ListBox ל Collection של איברים מטיפוס MyData. נחזור על הקוד של הדוגמא:

המחלקה MyData הוגדרה באופן הבא:

namespace CollectionDemo  
{  
  public class MyData  
  {  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
    public bool IsLecturer { get; set; }  
   
    public override string ToString()  
    {  
      return string.Format("First name: {0}, Last name: {1}, Is lecturer: {2}",  
        FirstName, LastName, IsLecturer);  
    }  
  }  
}

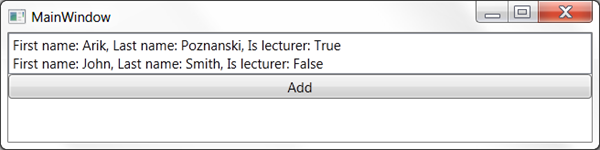
קוד ה XAML הוגדר להיות:

<Window x:Class="CollectionDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <StackPanel>  
    <ListBox ItemsSource="{Binding}" />  
    <Button Content="Add"  
           Click="Button\_Click" />  
  </StackPanel>  
</Window>

הקוד מאחור ב#C:

using System.Collections.ObjectModel;  
using System.Windows;  
   
namespace CollectionDemo  
{  
  /// <summary>  
  /// Interaction logic for MainWindow.xaml  
  /// </summary>  
  public partial class MainWindow : Window  
  {  
    private ObservableCollection<MyData> \_myCollection =  
      new ObservableCollection<MyData>();  
   
    public MainWindow()  
    {  
      InitializeComponent();  
   
      DataContext = \_myCollection;  
      \_myCollection.Add(  
        new MyData  
        {  
          FirstName = "Arik",  
          LastName = "Poznanski",  
          IsLecturer = true  
        });  
      \_myCollection.Add(  
        new MyData  
        {  
          FirstName = "John",  
          LastName = "Smith",  
          IsLecturer = false  
        });  
    }  
   
    private int counter = 0;  
    private void Button\_Click(object sender, RoutedEventArgs e)  
    {  
      ++counter;  
      \_myCollection.Add(  
          new MyData()  
          {  
            FirstName = "item " + counter,  
            LastName = "item " + counter,  
            IsLecturer = counter % 3 == 0  
          });  
    }  
  }  
}

תוצאת הרצת התוכנית הזו הייתה:



כעת נחזור לבעיה המקורית של התצוגה של אלמנט מטיפוס MyData . הבעיה בתצוגה של אלמנט זה הייתה שאלמנט זה אינו רכיב גרפי ולכן אינו יודע כיצד להציג את עצמו כראוי. כברירת מחדל, WPF קורא לפונקציה ToString של האובייקט MyData ומציג את התוצאה בתוך פקד TextBlock.

**Data Template** זוהי טכניקה המאפשרת לנו להגדיר כיצד אובייקט נתונים (לדוגמא, מטיפוס MyData) יראה בזצן ריצה בצורה גרפית.

כיצד נגדיר Data Template?

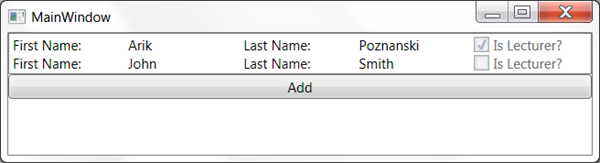
כדי להגדיר כיצד יש להציג אלמנטים מטיפוס MyData יש צורך להגדיר **Data Template** עבורו. להלן דוגמא:

<Window x:Class="DataTemplates.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       xmlns:local="clr-namespace:DataTemplates"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <Window.Resources>  
    **<DataTemplate x:Key="MyDataTemplate"  
                 DataType="local:MyData">**      <Grid>  
        <Grid.RowDefinitions>  
          <RowDefinition Height="Auto" />  
        </Grid.RowDefinitions>  
        <Grid.ColumnDefinitions>  
          <ColumnDefinition Width="\*" />  
          <ColumnDefinition Width="\*" />  
          <ColumnDefinition Width="\*" />  
          <ColumnDefinition Width="\*" />  
          <ColumnDefinition Width="\*" />  
        </Grid.ColumnDefinitions>  
   
        <TextBlock Grid.Column="0"  
                  Text="First Name: " />  
        <TextBlock Grid.Column="1"  
                  Text="{Binding Path=FirstName}" />  
        <TextBlock Grid.Column="2"  
                  Text="Last Name: " />  
        <TextBlock Grid.Column="3"  
                  Text="{Binding Path=LastName}" />  
        <CheckBox Grid.Column="4"  
                 Content="Is Lecturer?"  
                 IsEnabled="False"  
                 IsChecked="{Binding Path=IsLecturer}" />  
      </Grid>  
    **</DataTemplate>**  
  </Window.Resources>  
  <StackPanel>  
    <ListBox ItemsSource="{Binding}"  
**ItemTemplate="{StaticResource MyDataTemplate}"**            HorizontalContentAlignment="Stretch" />  
    <Button Content="Add"  
           Click="Button\_Click" />  
  </StackPanel>  
</Window>

בדוגמא זו אנו מגדירים שכל אלמנט מסוג MyData יוצג ע"י Grid שבתוכו מספר פקדים (מסוג TextBlock ו CheckBox) שיציגו בצורה גרפית את התכונות החשובות של האלמנט.

בנוסף נשים לב לשימוש בתכונה  **ItemTemplate**בפקד ListBox. תכונה זו מציינת באיזה  **Data Template** יש להשתמש בשביל לצייר את האלמנטים ברשימה. בזמן ריצה עבור כל אלמנט שברשימה ייוצר עותק מהתוכן הגרפי של ה **Data Template** לצורך הצגת האלמנט. בנוסף, האלמנט עצמו (מסוג MyData) יקבע באופן אוטומטי להיות ה **DataContext** של הפקדים הגרפיים שב **Data Template**. באופן זה כאשר משתמשים ב**Binding** בהגדרת ה **Data Template**, האובייקט מקור יהיה באופן אוטומטי האלמנט שגרם ליצירת הפקדים הנ"ל.

תוצאת הרצת התוכנית באמצעות הקוד הנ"ל נראית כך:



שימוש ב Implicit Data Templates

בדוגמא הקודמת הראינו כיצד להגדיר **Data Template** עבור אובייקט מטיפוס MyData ולתת לו מפתח מזהה, ואז להשתמש במפתח כדי להגדיר את סוג האובייקט בעזרת **ItemTemplate**.

ישנה אפשרות נוספת לעשות זאת, והיא באמצעות שימוש ב **Implicit Data Templates**.

ב **Implicit Data Templates** הכוונה היא יצירת **Data Template** ללא מפתח מפורש ( כלומר, ללא הגדרת**x:Key**), במקרה כזה ה **Data Template** יופעל בכל פעם שנרצה להציג למסך אובייקט מטיפוס MyData, גם ללא שימוש בתכונה **ItemTemplate**.

להלן הקוד של הדוגמא הקודמת תוך שימוש ב **Implicit Data Template**:

<Window x:Class="DataTemplates.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       xmlns:local="clr-namespace:DataTemplates"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <Window.Resources>  
**<DataTemplate DataType="local:MyData">**      <Grid>  
        <Grid.RowDefinitions>  
          <RowDefinition Height="Auto" />  
        </Grid.RowDefinitions>  
        <Grid.ColumnDefinitions>  
          <ColumnDefinition Width="\*" />  
          <ColumnDefinition Width="\*" />  
          <ColumnDefinition Width="\*" />  
          <ColumnDefinition Width="\*" />  
          <ColumnDefinition Width="\*" />  
        </Grid.ColumnDefinitions>  
   
        <TextBlock Grid.Column="0"  
                  Text="First Name: " />  
        <TextBlock Grid.Column="1"  
                  Text="{Binding Path=FirstName}" />  
        <TextBlock Grid.Column="2"  
                  Text="Last Name: " />  
        <TextBlock Grid.Column="3"  
                  Text="{Binding Path=LastName}" />  
        <CheckBox Grid.Column="4"  
                 Content="Is Lecturer?"  
                 IsEnabled="False"  
                 IsChecked="{Binding Path=IsLecturer}" />  
      </Grid>  
    </DataTemplate>  
  </Window.Resources>  
  <StackPanel>  
**<ListBox ItemsSource="{Binding}"**            HorizontalContentAlignment="Stretch" />  
    <Button Content="Add"  
           Click="Button\_Click" />  
  </StackPanel>  
</Window>

# [**שימוש ב Dependency Properties**](http://webmaster.org.il/articles/wpf-dependency-properties)

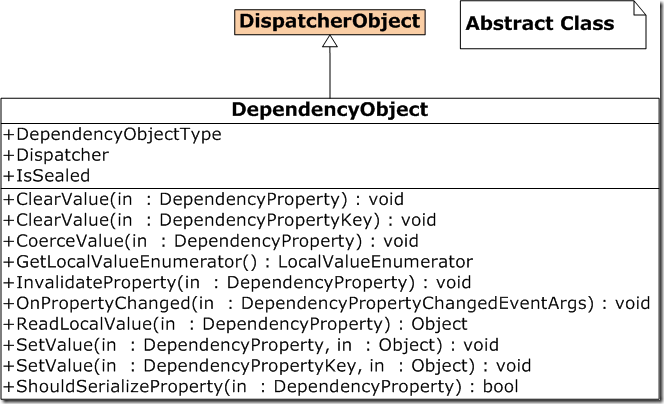
‏ ‏הקדמה

כפי שאתם יודעים, במחלקה בשפת #C ניתן להגדיר Properties (תכונות). כאשר פיתחו את WPF, עלה הצורך **בProperties משוכללים**, עם יכולות נוספות מעבר לProperties הרגילים.

ל Properties החדשים שפיתחו ב WPF קוראים **Dependency Properties**.

כמעט כל Property שנמצאת בשימוש ב WPF היא Dependency Property. בפרק זה נפרט את היתרונות בDependency Property. נעיר כי פרק זה הוא די תיאורטי, ברוב היתרונות אנחנו משתמשים רק בעקיפין ולא שמים לב שזה עובד רק בזכות Dependency Properties.

Dependency property הוא מאפיין שצריך להיות מוגדר בתוך מחלקה שיורשת מהמחלקה DependencyObject



הפונקציה **OnPropertyChanged**

נקראת אוטומטית כאשר הערך של המאפיין משתנה.

המנגנון של ה Binding למשל, רושם אותה לאירוע עדכון הערך

באופן כללי dependency properties מוגדרות בתוך **מילון (Dictionary) סטטי** עבור כל האובייקטים מאותו טיפוס.

וחלקם מוגדרים במילון שאינו סטטי עבור כל אובייקט בעצמו.

לכל DP יש **ערך ברירת מחדל** המוגדר בתוך המילון הסטטי של הטיפוס.

שינוי הערך נעשה ע"י הפונקציה SetValue וקבלת הערך ע"י GetValue של המחלקה DependencyObject.

כל הDP של מחלקה מסויימת, שיש להם ערך ברירת מחדל**, נשמרים בזכרון פעם אחת במילון** עבור כל האובייקטים. והם לא חלק מהזכרון של האובייקט עצמו.

רק אם משנים ערך של DP אז הוא נוצר כחלק מהזכרון של האובייקט. **זה חסכון עצום בזכרון.**

**למשל:** כל האובייקטים מטיפוס Button יש להם המון תכונות DP שמשותפות לכל הכפתורים. ורק נשנה את התכונות הספציפיות.

**Button : DependencyObject**

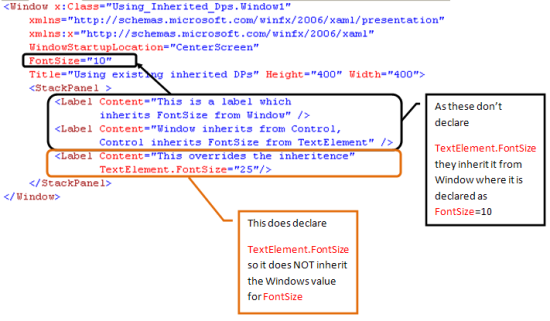
|  |  |
| --- | --- |
| **Value** | **Key** |
| true | IsEnable |
| 12 | FontSize |
| null | Content |
| … | … |

<Button>button</Button>

<Button FontSize="30">button</Button>

<Button FontSize="30" IsEnabled="False">button</Button>

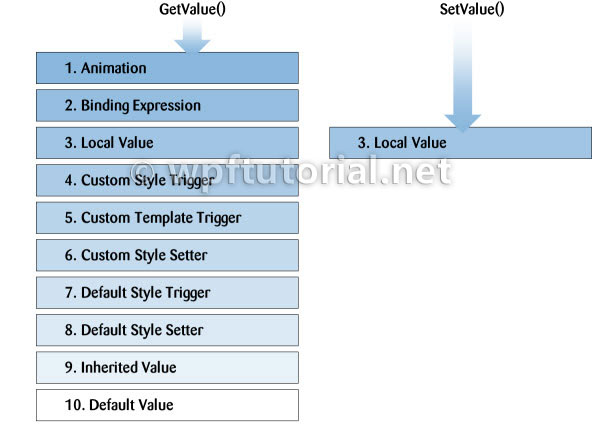
**יתרון נוסף**, ניתן להגדיר ערך לDP של אלמנט ראשי בXAML כך שערכו יחלחל לכל האלמנטים מתחתיו בהיררכיה.



המטרה העיקרית של  Dependency Property היא לספק דרך לחשב ערך של תכונה שמבוסס על מספר מקורות שונים. לדוגמא, לערך של תכונה  FontSize של כפתור יכולים להיות מספר מקורות, למשל:

* הגדרת גודל פונט במערכת ההפעלה
* הגדרת גודל פונט של החלון הראשי (שיכולה לחלחל אל כל הפקדים שמתחתיו)
* ערך שמגיע בעזרת Data Binding (למדנו כבר במדריך זה. (עמוד 19)
* ערך שמגיע בעזרת אנימציה
* ערך שמגיע בעזרת Style
* ועוד..

תהליך בחירת הערך לDP



יתרונות של Dependency Properties

* שימוש יעיל בזיכרון ביחס לתכונות רגילות כאשר יש הרבה פקדים ולכל פקד יש מאות תכונות
* **מאפשר הצבה של ערך באמצעות DynamicResource ( ילמד בהמשך המדריך)**
* **מאפשר הצבה של ערך באמצעות  Data Binding (ילמד בהמשך המדריך)**
* מאפשר הצבה של ערך באמצעות **Animation**
* **מאפשר קבלת ערך ע"י ירושת ערך מפקד אשר מכיל את הפקד הנוכחי**
* **מספק דרך לקבלת הודעה (אירוע) כאשר הערך של התכונה משתנה**

 זיהוי Dependency Property

נשים לב כי Dependency Property לרוב נראה כמו Property רגיל מבחינת השימוש בו. הדרך הטובה ביותר לזהות שProperty מסוים הוא Dependency Property היא ע"י קריאת התיעוד של הProperty.

 שימוש ב Dependency Property

כאמור, השימוש ב Dependency Property נעשה בצורה זהה ל Property רגיל, כלומר אם נרצה להציב ערך באמצעות XAML נוכל להשתמש בקוד מהצורה הבאה:

    <Button Width="100" />

שינוי ערך של Dependency Property באמצעות קוד #C נעשה גם כן כרגיל:

Button myButton = new Button();  
myButton.Width = 100;

הגדרת Dependency Property במחלקה שלנו

רוב הזמן אנו רק נשתמש בDependency Properties ולא ניצור חדשים. למרות זאת, בואו נראה דוגמא של יצירת Dependency Property בשם **Age** מטיפוס **int** במחלקה משלנו. נעיר כי פירוט מעמיק יותר על Dependency Properties הוא מעבר לגבולות מדריך זה.

ראשית יש לוודא שהמחלקה שלנו יורשת באופן ישיר או עקיף מ DependencyObject. בנוסף יש צורך לבצע רישום ברגיסטרי לתכונה החדשה שלנו, וכן לחשוף Property רגיל שמאחורי הקלעים מביא את הערך באמצעות שימוש במנגנונים שמסופקים ע"י מערכת ה Properties של WPF. להלן הדוגמא המלאה:

קוד זה נוצר אוטומטית באמצעות ה SNIPPET

**propdp**

using System.Windows;  
   
namespace DependencyPropertiesDemo  
{  
  class Person : DependencyObject  
  {  
    public int Age  
    {  
      get { return (int)GetValue(AgeProperty); }  
      set { SetValue(AgeProperty, value); }  
    }  
   
    public static readonly DependencyProperty AgeProperty =  
        DependencyProperty.Register("Age", typeof(int), typeof(Person), new UIPropertyMetadata(0));  
  }  
}

כאשר הDP היא של מחלקה רגילה הקוד שלעיל מספיק. אך כאשר נייצר DP ל USER CONTROL נצטרך גם להגדיר את הפונקציות האלו:

public static object **ValueCoerceValueCallback**(DependencyObject d, object baseValue)

{

float? value = baseValue as float?;

NumericUpDownControl THIS = d as NumericUpDownControl;

if (value > THIS.MaxValue)

return THIS.MaxValue;

else if (value < THIS.MinValue)

return THIS.MinValue;

else return value;

}

public static void **PropertyChangedCallback**

(DependencyObject d, DependencyPropertyChangedEventArgs e)

{

NumericUpDownControl THIS = d as NumericUpDownControl;

THIS.textNumber.Text = THIS.Value == null ? "" : THIS.Value.ToString();

}

ואז לעדכן את הDP בהתאם כך:

public static readonly DependencyProperty ValueProperty =

DependencyProperty.Register("Value", typeof(float?), typeof(NumericUpDownControl),

**new(PropertyMetadata, Nullable, PropertyChangedCallback)**);

**ראו הרחבה אצל אושרי כהן ב OSF תחת, WPF נושאים מתקדמים.**

מקור נוסף וחשוב לעיון:<https://www.wpftutorial.net/dependencyproperties.html>

# **WPF Dependency Property vs INotifyPropertyChanged**

When to use WPF Dependency Property vs INotifyPropertyChanged?

A little background:  
WPF databinding has two essential parts: Source and Target.  
Now Target has to be a Dependency Property.  
But source can be a simple property with INotifyPropertyChanged implemented. Or a Dependency property.  
In both cases, any change to the property will see the UI getting updated.  
So back to the question.  
Which one?

Well it is a matter of choice. Making the source property a dependency object does make the code very simple. But making it a Dependency Property is a bit involved. I keep the source property as a plain .NET property. And implement the INotifyPropertyChanged.

If I am writing a custom(uaser) control only then I create the dependency properties (mainly because I do not have a choice).

[לקוח מ:](http://webmaster.org.il/articles/wpf-resources) <http://ikeptwalking.com/wpf-dependency-property-vs-inotifypropertychanged/>

# **[שימוש בResources](http://webmaster.org.il/articles/wpf-resources)**

לפני שנתחיל נעיר כי ה **Resources**  שאנו מדברים עליהם בפרק זה אינם קשורים בכלל ל  **Project Resources** שאנו מכירים מעידן הקודם לWPF. . השימוש ב **Project Resources** מיועד בעיקר להוספת תמונות וטבלת מחרוזות (strings) שניתן לבצע להם לוקליזציה (תרגום לשפה מקומית). ה **Resources** שאנחנו נדבר עליהם בפרק זה הם תכונה ייחודית לטכנולוגיות מבוססות XAML.

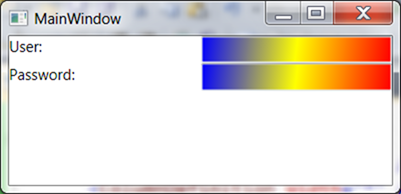
מה זה Resources ?

**Resource (משאב) בWPF זו אפשרות להגדיר ערכים במילון (Dictionary) שמזוהים ע"י מפתח (Key).**

**שימוש בResources  מאפשר לנו לעשות שימוש חוזר באובייקטים שיצרנו.** ע"י שימוש ב  **Resources** נוכל לחסוך בזיכרון (פחות אובייקטים ישבו בזיכרון), לחסוך בזמן (עצם היצירה של אובייקטים לוקחת זמן), וכן לחסוך בכתיבת קוד (הגדרה של אובייקטים יכולה להיות מורכבת וחבל לחזור עליה שוב ושוב).

נראה דוגמא שבה נרצה להשתמש ב  **Resources**. בקוד הבא אנו מגדירים טופס לקבלת נתוני User ו Password של משתמש, וקבענו רקע צבעוני במיוחד לשני ה TextBox בתוכנית. השימוש במחלקה**LinearGradientBrush** ליצירת רקע מתחלף נלמד במדריך WPF. כרגע מה שחשוב לשים לב הוא שאותה הגדרה מורכבת של האוביקט LinearGradientBrush מופיעה פעמיים בקוד הXAML:

<Window x:Class="ResourcesDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <Grid>  
    <Grid.RowDefinitions>  
      <RowDefinition Height="Auto" />  
      <RowDefinition Height="Auto" />  
    </Grid.RowDefinitions>  
    <Grid.ColumnDefinitions>  
      <ColumnDefinition Width="\*" />  
      <ColumnDefinition Width="\*" />  
    </Grid.ColumnDefinitions>  
    <TextBlock Text="User:"  
              Grid.Row="0"  
              Grid.Column="0" />  
    <TextBox Grid.Row="0"  
            Grid.Column="1">  
      <TextBox.Background>  
        **<LinearGradientBrush>  
          <GradientStop Color="Blue"  
                       Offset="0" />  
          <GradientStop Color="Yellow"  
                       Offset="0.5" />  
          <GradientStop Color="Red"  
                       Offset="1" />  
        </LinearGradientBrush>**      </TextBox.Background>  
    </TextBox>  
    <TextBlock Text="Password:"  
              Grid.Row="1"  
              Grid.Column="0" />  
    <TextBox Grid.Row="1"  
            Grid.Column="1">  
      <TextBox.Background>  
     **<LinearGradientBrush>  
          <GradientStop Color="Blue"  
                       Offset="0" />  
          <GradientStop Color="Yellow"  
                       Offset="0.5" />  
          <GradientStop Color="Red"  
                       Offset="1" />  
        </LinearGradientBrush>**      </TextBox.Background>  
    </TextBox>  
  </Grid>  
</Window>



בזמן ריצה של הקוד הנ"ל, ייווצרו שני אובייקטים מטיפוס **LinearGradientBrush**  שיהיו בעלי אותם ערכים. נרצה לראות כיצד להימנע מבזבוז זה. הפתרון יהיה להגדיר את האובייקט LinearGradientBrush פעם אחת בלבד במקום מסוים, ואז להשתמש בו בכל פעם שנצטרך ע"י הפניה לאותו מקום.

כיצד נגדיר Resource?

כמעט לכל פקד בWPF יש Property בשם **Resources**. הטיפוס של תכונה זו הוא **Dictionary**. כלומר ניתן להכניס לתכונה זו זוגות של מפתח (Key) וערך (Value).

כדי להוסיף אובייקט לResources של פקד נשתמש בקוד הבא:

  <**Window.Resources**>  
    <LinearGradientBrush **x:Key="myLinearGradientBrush"**>  
      <GradientStop Color="Blue"  
                   Offset="0" />  
      <GradientStop Color="Yellow"  
                   Offset="0.5" />  
      <GradientStop Color="Red"  
                   Offset="1" />  
    </LinearGradientBrush>  
  </**Window.Resources**>

הוספנו את האובייקט LinearGradientBrush שלנו תחת תכונת **Resources** של החלון **ובנוסף** הוספנו לו תגית **x:Key**כדי לתת לו מפתח שיזהה אותו. יותר מאוחר אנחנו נשתמש במפתח הזה כדי לפנות לאובייקט.

נעיר כי ניתן לשים אובייקטים בתכונת **Resources** של כל פקד שיש לו את התכונה הזו, אבל לרוב אנו נשים את האובייקטים במקום מרכזי כמו באובייקט החלון הראשי (**Window**) או באובייקט האפליקציה (**App**).

כיצד נשתמש ב Resource שהגדרנו?

ישנם שני דרכים להשתמש בResource שהגדרנו:

1. ע"י שימוש ב  **StaticResource** – דרך זו יותר יעילה מבחינת ביצועים, אך טובה רק כאשר ה Resource שאליו אנו פונים אינו משתנה.
2. ע"י שימוש ב  **DynamicResource** – בדרך זו ניתן לפנות גם ל Resource שמשתנה, אך נשלם על כך בביצועים.

לרוב מספיק לעבוד עם  **StaticResource** ורק כאשר נצטרך נשתמש ב **DynamicResource** .

כדי להשתמש בResource שהגדרנו נכתוב את הקוד הבא:

<TextBox Grid.Row="0"  
         Grid.Column="1"  
         Background="**{StaticResource myLinearGradientBrush}**">

במקום להגדיר את כל האובייקט LinearGradientResource תחת התכונה Background כמו מקודם, כל מה שצריך לעשות להשתמש בתגית המיוחדת  **StaticResource**ולאחריה לתת את המפתח שמזהה את האובייקט המבוקש. כמובן שניתן לפנות לאובייקט ממספר מקומות שונים (זאת הייתה המטרה מלכתחילה).

להלן הקוד המלא של הדוגמא המתוקנת:

<Window x:Class="ResourcesDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
**<Window.Resources>  
    <LinearGradientBrush x:Key="myLinearGradientBrush">  
      <GradientStop Color="Blue"  
                   Offset="0" />  
      <GradientStop Color="Yellow"  
                   Offset="0.5" />  
      <GradientStop Color="Red"  
                   Offset="1" />  
    </LinearGradientBrush>**

**</Window.Resources>**  
  <Grid>  
    <Grid.RowDefinitions>  
      <RowDefinition Height="Auto" />  
      <RowDefinition Height="Auto" />  
    </Grid.RowDefinitions>  
    <Grid.ColumnDefinitions>  
      <ColumnDefinition Width="\*" />  
      <ColumnDefinition Width="\*" />  
    </Grid.ColumnDefinitions>  
    <TextBlock Text="User:"  
              Grid.Row="0"  
              Grid.Column="0" />  
    <TextBox Grid.Row="0"  
            Grid.Column="1"  
**Background="{StaticResource myLinearGradientBrush}">**    </TextBox>  
    <TextBlock Text="Password:"  
              Grid.Row="1"  
              Grid.Column="0" />  
    <TextBox Grid.Row="1"  
            Grid.Column="1"  
**Background="{StaticResource myLinearGradientBrush}">**    </TextBox>  
  </Grid>  
</Window>

התגית **StaticResource** דואגת לחפש את האובייקט המבוקש ולהציב אותו בתור ערך לתכונה.

כיצד מתבצע תהליך החיפוש?

תהליך החיפוש של  **StaticResource**וגם  **DynamicResource**עובד באופן הבא:

החיפוש משתמש במפתח כדי למצוא את האובייקט באחד הDictionary במעלה העץ.

ראשית מחפשים את האובייקט המבוקש (בדוגמא LinearGradientBrush ) בתכונה **Resources** של הפקד הנוכחי (בדוגמא TextBox), אם מצאנו את האוביקט ב Dictionary שלו התהליך מסתיים, אחרת החיפוש עולה רמה אחת בעץ הפקדים (בדוגמא Grid) ושוב מחפש בתכונה **Resources**, וכך התהליך ממשיך. בדוגמא שלנו, לאחר שהאובייקט לא נמצא ב Grid אנו מחפשים ב **Resources**  של הWindow ושם האובייקט נמצא ומוחזר בהצלחה.

אם האובייקט לא נמצא בשורש העץ שהוא לרוב מטיפוס Window אזי מתבצע חיפוש נוסף בקובץ הXAML של האפליקציה: App.xaml.

 מתי מתבצע תהליך החיפוש?

כאשר אנו משתמשים **ב StaticResource** תהליך החיפוש מתבצע כאשר WPF זקוק לערך של התכונה פעם ראשונה (לרוב בטעינת התוכנית), ואז מתבצע החיפוש על עץ הפקדים. לאחר החיפוש התוצאה נשמרת ולא מתבצעים יותר חיפושים בהמשך.

לעומת זאת, כאשר נשתמש **בDynamicResource**  החיפוש יתבצע מחדש בכל פעם שWPF יזדקק לערך. באופן זה ניתן לקבל ערכים חדשים במקרה וה Dictionary שמכיל את האובייקט המבוקש משתנה. כמובן שזה עולה לנו בביצועים מאחר והחיפוש חוזר על עצמו בכל שימוש בתכונה.

נעיר כי כדי להשתמש ב **DynamicResource** על התכונה שאנו קובעים להיות מסוג **DependencyProperty**. מגבלה זו אינה חלה כאשר אנו עובדים עם **StaticResource**.

# **תוספות מאושרי כהן OSF – WPF נושאים מתקדמים**

ניתן למקם משאב במקומות שונים:

1. **תחת פקד:**

<Canvas.Resources>

<LinearGradientBrush

**x:Key="MyButtonBackground"**

EndPoint="0.504,1.5" StartPoint="0.504,0.03">

<GradientStop Color="#FFFFC934" Offset="0"/>

<GradientStop Color="#FFFFFFFF" Offset="0.567"/>

</LinearGradientBrush>

</Canvas.Resources>

1. **בקובץ הקיים App.xaml שנמצא אוטומטית בכל אפליקציה:**

<Application.Resources>

<LinearGradientBrush

**x:Key="MyButtonBackground"**

EndPoint="0.504,1.5" StartPoint="0.504,0.03">

<GradientStop Color="#FFFFC934" Offset="0"/>

<GradientStop Color="#FFFFFFFF" Offset="0.567"/>

</LinearGradientBrush>

</Application.Resources>

ואז להשתמש בו כמשאב סטטי או כמשאב דינאמי על פי רצוננו.

למשל נסתכל בXAML של הגדרת חלון כלשהו:

<Button Grid.Row="2"

Grid.Column="0"

Content="Push Me - Static"

**Background="{StaticResource MyButtonBackground}"** Margin="0,0,0,-36" >

</Button>

<Button Grid.Row="2"

Grid.Column="1"

Content="Push Me - Dynamic"

**Background="{DynamicResource MyButtonBackground}"**

Margin="0,0,0,-36"

Click="Button\_Click" >

</Button>

אם זה משאב דינאמי, אז לכל מרכיב UIיש גישה למשאבים שלו, וניתן לגשת לפי מפתח, ולשנות המשאב הדינאמי דרך ה C# קוד כך:

private void Button\_Click(object sender, RoutedEventArgs e)

{

LinearGradientBrush br =

new LinearGradientBrush

(Color.FromRgb(20, 227, 250), Color.FromRgb(200, 199, 238), 4.6);

**this.Resources["MyButtonBackground"] = br;**

}

1. **אפשר גם להגדיר משאבים בקובץ XAML חדש משלנו שניצור.**

ואז , לציין בקובץ הקיים App.xaml שהמקור של המשאבים מגיע מהקובץ שלנו:

<Application.Resources>

<ResourceDictionary>

<ResourceDictionary.MergedDictionaries>

**<ResourceDictionary Source="(@"/res/languages/AppStrings\_HE.xaml"/>**

</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

</Application.Resources>

או שניתן לטעון את קובץ המשאבים שלנו דרך הקוד:

public MainWindow()

{

InitializeComponent();

Uri dictUri =

new Uri(@"/res/languages/AppStrings\_HE.xaml", UriKind.Relative);

ResourceDictionary resourceDict =

Application.LoadComponent(dictUri) as ResourceDictionary;

Application.Current.Resources.MergedDictionaries.Clear(); Application.Current.Resources.MergedDictionaries.Add(resourceDict);

}

**ראו הרחבה אצל אושרי כהן ב OSF תחת, WPF נושאים מתקדמים.**

# [**שימוש ב Routed Events**](http://webmaster.org.il/articles/wpf-routed-events)

הקדמה

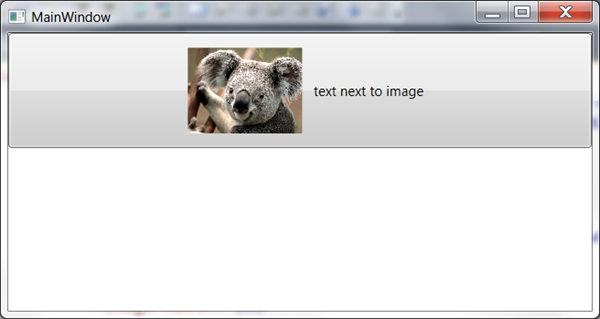
בפרק שעבר למדנו שבWPF החליטו לשפר את מערכת הProperties ולהוסיף Properties בעלי יתרונות חדשים שנקראים Dependency Properties. באותו אופן, גם Events (אירועים) היו זקוקים למקצה שיפורים בWPF, ולEvents מהסוג החדש קוראים **Routed Events**. אלו אירועים בעלי יתרונות ביחד לאירועים הרגילים בשפת #C. גם פרק זה יהיה בעיקר תיאורטי.

יתרונות Routed Events

הרעיון המרכזי ב **Routed Events** הוא שכאשר אירוע מופעל, הוא מופעל מספר פעמים, על פקדים נוספים בעץ הפקדים באפליקציה שלנו. לדוגמא, אם יש לנו את מבנה האפליקציה הבא:

<Window x:Class="EventsDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <StackPanel>  
    <Button>  
      <Button.Content>  
        <StackPanel Orientation="Horizontal">  
          <Image Width="100"  
                Source="/EventsDemo;component/Images/Koala.jpg"  
                Margin="10" />  
          <TextBlock Text="text next to image"  
                    VerticalAlignment="Center" />  
        </StackPanel>  
      </Button.Content>  
    </Button>  
  </StackPanel>  
</Window>

באפליקציה זו יש לנו חלון שבתוכו StackPanel ובתוכו כפתור, והכפתור מכיל תמונה וטקסט.



נניח שאירועים היו מתנהגים כרגיל, ולא כמו **ש Routed Events** עובדים. ונניח שנרצה להגיב לאירוע ללחיצה עם כפתור ימני של העכבר על הכפתור שלנו, אזי נצטרך להירשם לאירוע  **MouseRightButtonDown**. אבל מה היה קורה אם היינו מבצעים את הלחיצה על התמונה שנמצאת בתוך הכפתור? מי שיקבל את הלחיצה ויפעיל את האירוע הוא אותה תמונה ולא הכפתור! כלומר אם נרצה שיופעל הקוד שלנו בעת לחיצה ימנית על הכפתור (בתוך התמונה ומחוצה לה!), יש להירשם לאירוע גם של הכפתור, גם של התמונה, גם של הפקד TextBlock וגם של ה StackPanel! כמובן שמצב כזה הוא מאוד מסורבל ולא נוח. הקוד הדרוש לרישום לכל האירועים הללו יראה כך:

<Window x:Class="EventsDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <StackPanel>  
    <Button **MouseRightButtonDown="Button\_MouseRightButtonDown"**>  
      <Button.Content>  
        <StackPanel Orientation="Horizontal"  
                    **MouseRightButtonDown="Button\_MouseRightButtonDown"**>  
          <Image Width="100"  
                Source="/EventsDemo;component/Images/Koala.jpg"  
                Margin="10"  
                 **MouseRightButtonDown="Button\_MouseRightButtonDown"** />  
          <TextBlock Text="text next to image"  
                    VerticalAlignment="Center"  
                     **MouseRightButtonDown="Button\_MouseRightButtonDown"** />  
        </StackPanel>  
      </Button.Content>  
    </Button>  
  </StackPanel>  
</Window>

התוספת המרכזית **בRouted Events**  היא שלאחר שהאירוע שלנו מופעל על התמונה, האירוע "מטפס" במעלה עץ הפקדים, מגיע ל StackPanel ומופעל גם עליו, ואז ממשיך במעלה עץ הפקדים, ומופעל על ה Button, ואז תקרא הפונקציה שרשמנו אותה. באופן זה המקום היחיד שנצטרך להירשם אליו הוא הכפתור עצמו, כלומר הקוד הופך להיות קצר והגיוני:

<Window x:Class="EventsDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <StackPanel>  
    <Button **MouseRightButtonDown="Button\_MouseRightButtonDown"**>  
      <Button.Content>  
        <StackPanel Orientation="Horizontal">  
          <Image Width="100"  
                Source="/EventsDemo;component/Images/Koala.jpg"  
                Margin="10" />  
          <TextBlock Text="text next to image"  
                    VerticalAlignment="Center" />  
        </StackPanel>  
      </Button.Content>  
    </Button>  
  </StackPanel>  
</Window>

אם נזכור את העובדה שבWPF קל מאוד לשים פקדים בתוך פקדים, מאחר והתכונה Content יכולה להכיל אוסף פקדים אחרים, נבין שיתרון של Routed Events ביחס לאירועים רגילים הוא ממש הכרחי בWPF. מסיבה זו רוב האירועים בWPF הם **Routed Events**.

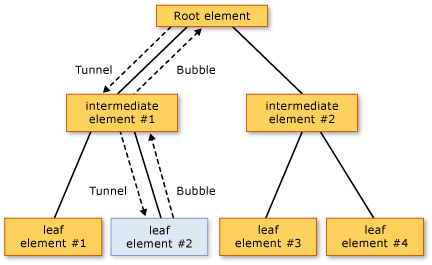
סוגים שונים של Routed Events

**קיימים שלושה סוגים של Routed Events. ההבדל בין הסוגים השונים הוא מה ההתנהגות של האירוע לאחר שהוא הופעל לראשונה**. כל אירוע ב WPF הוא בעל סוג **מוגדר** ומשתייך לאחד מהסוגים הנ"ל:

1. אירוע מסוג "פעפוע"  **Bubbling**– אירוע מסוג זה מתרחש **קודם על הפקד שיצר אותו** **ואז** **מטפס במעלה עץ הפקדים** ופועל שוב ושוב עד שהוא מגיע לשורש העץ, החלון הראשי. בדוגמא הקודמת האירוע **MouseRightButtonDown** הוא מסוג Bubbling.
2. אירוע מסוג "מנהור"  **Tunneling**– אירוע מסוג זה מתרחש **קודם על שורש העץ, החלון הראשי, ואזי הוא מחלחל במורד העץ**, ופועל שוב ושוב על הפקדים שבדרך, **עד שהוא מגיע לפקד שיצר אותו**. אירוע כזה לדוגמא נקרא **PreviewMouseRightButtonDown**. נעיר שהרבה סוגים של אירועים (כמו לחיצה עם העכבר) מגיעים בזוגות. אירוע אחד מסוג Bubbling ואירוע נוסף מסוג Tunneling בעל קידומת Preview בשם של האירוע.
3. אירוע מסוג **Direct**– אירוע מסוג זה הוא אירוע שפועל רק על הפקד שיצר אותו ואינו מחלחל על עץ הפקדים בשום כיוון. סוג זה דומה לאופן שבו האירועים הרגילים עובדים.

אירוע כזה לדוגמא נקרא**MouseEnter**

אם אינכם בטוחים מה סוג האירוע שאליו אתם נרשמים ניתן לבדוק זאת בתיעוד של האירוע.



הפקד שיצר את האירוע

ניתן כמובן ליצור  **Routed Events** משלכם, אבל זה חורג מגבולות המדריך.

<Window x:Class="Ex96\_RoutedEvent.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Ex96\_RoutedEvent"

mc:Ignorable="d"

Title="MainWindow" Height="687.973" Width="525">

<StackPanel>

<Button MouseRightButtonDown="Button\_MouseRightButtonDown">

<Button.Content>

<StackPanel Orientation="Horizontal">

<Image Width="100" Margin="10" Source="Images/20170724\_100331.jpg" />

<TextBlock Text="Mouse Right - Root Element" VerticalAlignment="Center" />

</StackPanel>

</Button.Content>

</Button>

<Button MouseRightButtonDown="Button\_MouseRightButtonDown">

<Button.Content>

<StackPanel Orientation="Horizontal">

<Image Width="100" Margin="10" Source="Images/20170724\_100331.jpg" />

<TextBlock Text="Mouse Right - Bubble" VerticalAlignment="Center"

MouseRightButtonDown="Button\_MouseRightButtonDown"/>

</StackPanel>

</Button.Content>

</Button>

<Button PreviewMouseRightButtonDown="Button\_PreviewMouseRightButtonDown">

<Button.Content>

<StackPanel Orientation="Horizontal">

<Image Width="100" Margin="10" Source="Images/20170724\_100331.jpg" />

<TextBlock Text="Preview Mouse Right - Tunnel"VerticalAlignment="Center"

PreviewMouseRightButtonDown="Button\_PreviewMouseRightButtonDown"/>

</StackPanel>

</Button.Content>

</Button>

<Button MouseEnter="Button\_MouseEnter">

<Button.Content>

<StackPanel Orientation="Horizontal">

<Image Width="100" Margin="10" Source="Images/20170724\_100331.jpg" />

<TextBlock x:Name="txt4" Text="Mouse Enter - Direct"

VerticalAlignment="Center" MouseEnter="Button\_MouseEnter"/>

 </StackPanel>

</Button.Content>

</Button>

</StackPanel>

</Window>

[**שימוש ב Styles**](http://webmaster.org.il/articles/wpf-styles)

בשביל מה צריך Style?

לפני שנדון במה זה **Style**  בדיוק, כדאי להבין את הבעיה שהוא בא לפתור.

תארו לכם אפליקציה שיש בה מספר פקדים מאותו סוג, לדוגמא כפתורים. היינו רוצים שהכפתורים באפליקציה שלנו לא יהיו משעממים לכן נרצה לשנות את המראה שלהם ע"י שינוי פונטים, צבעים גודל ועוד. הבעיה היא שצריך לעשות את השינויים הנ"ל על כל אחד מהכפתורים בנפרד! אם יש לנו הרבה כפתורים זה יכול להצטבר לכדי הרבה עבודה. ומה קורה אם נרצה לשנות את צבע הרקע בעתיד? נצטרך שוב לעבור על עשרות כפתורים ולבצע את השינוי בעשרות מקומות.

כאן נכנס **Style**.

**Style** מאפשר לנו להגדיר אוסף של זוגות, כל זוג מכיל שם של תכונה והערך שלה. לאחר שהגדרנו פעם אחת את ה**Style** שלנו ניתן להפעיל אותו על כל כפתור וכך לרכז במקום אחד את המראה של הכפתור.

הגדרה בסיסית של Style

כמעט לכל פקד יש תכונה בשם **Style** שמאפשרת הצבת אובייקט מסוג  **Style**בתוכה.   
מה שנהוג לעשות הוא להגדיר פעם אחת אובייקט **Style** במקום משותף כמו תכונת ה Resources של החלון ואז להפנות אליו מכל פקד שנרצה שישתמש בו.

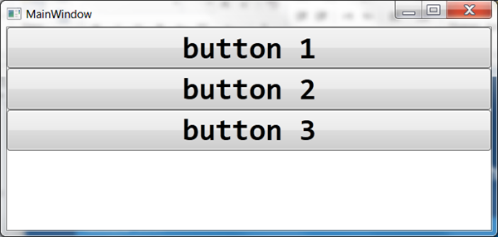
בדוגמא הבאה אנו מגדירים אובייקט **Style**  שפועל על כפתורים, המפתח המזהה של ה **Style**  יהיה myButtonStyle. האובייקט Style שלנו יגדיר ערכים לתכונות FontSize, FontFamily & FontWeight:

<Window x:Class="StylesDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <Window.Resources>  
 **<Style x:Key="myButtonStyle"  TargetType="Button">  
      <Setter Property="FontSize"  
             Value="32" />  
      <Setter Property="FontFamily"  
             Value="Consolas" />  
      <Setter Property="FontWeight"  
             Value="Bold" />  
    </Style>**  </Window.Resources>  
  <StackPanel>  
    <Button **Style="{StaticResource myButtonStyle}"**  
           Content="button 1" />  
    <Button **Style="{StaticResource myButtonStyle}"**           Content="button 2" />  
    <Button **Style="{StaticResource myButtonStyle}"**  
           Content="button 3" />  
  </StackPanel>  
</Window>

נשים לב לאופן ההגדרה של ה Style. יש להגדיר בתכונה  **TargetType** את סוג הפקדים שעליהם ה Style יופעל, בדוגמא אצלנו קבענו שיופעל על כפתורים. בנוסף בתוך ה Style יש אוסף של אובייקטים מסוג  **Setter**שבכל אחד מהם אנו מגדירים תכונה מבוקשת (**Property**) וערך המבוקש עבור התכונה (**Value**).

כדי להשתמש ב Style אנו קובעים בכל כפתור את התכונה**Style**  שתופנה לאותו אובייקט שהגדרנו ב Resources.

התוצאה של הרצת קוד:

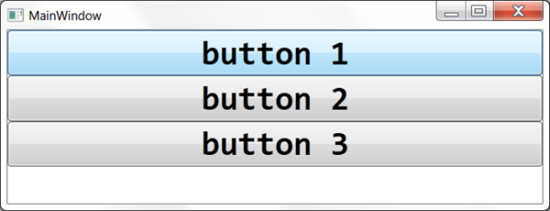


הגדרת של Implicit Style

באופן דומה לאופן שהגדרנו **Implicit Data Template** באחד הפקדים הקודמים, ניתן להגדיר גם **Implicit Style**. לשם כך, יש להשמיט מהגדרת ה Style את התכונה **x:Key**. כעת ה Style שהגדרנו יופעל באופן אוטומטי על כל אובייקט המתאים לסוג המופיע בתכונה **TargetType**. לפיכך נוכל לכתוב את הדוגמא הקודמת באופן הבא:

<Window x:Class="StylesDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow"  
       Height="350"  
       Width="525">  
  <Window.Resources>  
    <Style TargetType="Button">  
      <Setter Property="FontSize"  
             Value="32" />  
      <Setter Property="FontFamily"  
             Value="Consolas" />  
      <Setter Property="FontWeight"  
             Value="Bold" />  
    </Style>  
  </Window.Resources>  
  <StackPanel>  
    <Button Content="button 1" />  
    <Button Content="button 2" />  
    <Button Content="button 3" />  
  </StackPanel>  
</Window>

ולקבל את אותה התוצאה בדיוק:



נעיר כי במידה ונרצה להגדיר שני סוגים של כפתורים נהיה חייבים לעבוד בשיטה הקודמת שכן לא ניתן להגדיר שני Implicit Style לאותו סוג של אובייקט.

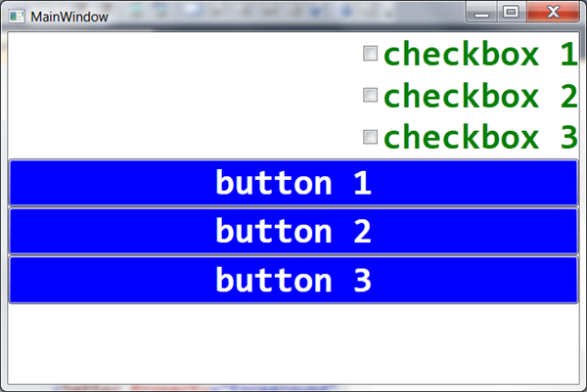
 ירושה של Style

ניתן להגדיר מ"ס אובייקטים של Style שירשו זה מזה באמצעות שימוש בתכונה **BasedOn**. זה יכול להיות שימושי במקרים שבהם יש לנו Style לכל סוג של פקד כגון Button ו CheckBox אבל יש לנו הרבה דברים שמשותפים לפקדים הללו ונרצה להגדיר זאת פעם אחת.

לדוגמא, בקטע הקוד הבא ניתן לראות לראות הגדרה של **Style**  כללי בשם commonStyle שמגדיר את מראה הפונטים, וכן הגדרה של **Implicit Style**  עבור Button ועבור CheckBox, שכל אחד מהם יורש את התכונות של commonStyle ומוסיף עליהם:

<Window x:Class="StylesDemo.MainWindow"  
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="MainWindow" Height="350" Width="525">  
  <Window.Resources>  
    <Style x:Key="commonStyle"  
          TargetType="Control">  
      <Setter Property="FontSize"  
             Value="32" />  
      <Setter Property="FontFamily"  
             Value="Consolas" />  
      <Setter Property="FontWeight"  
             Value="Bold" />  
    </Style>  
    <Style TargetType="Button"  
           **BasedOn="{StaticResource commonStyle}">**      <Setter Property="Background"  
             Value="Blue" />  
      <Setter Property="Foreground"  
             Value="White" />  
    </Style>  
    <Style TargetType="CheckBox"  
           **BasedOn="{StaticResource commonStyle}">**      <Setter Property="Foreground"  
             Value="Green" />  
      <Setter Property="HorizontalAlignment"  
             Value="Right" />  
    </Style>  
  </Window.Resources>  
  <StackPanel>  
    <CheckBox Content="checkbox 1" />  
    <CheckBox Content="checkbox 2" />  
    <CheckBox Content="checkbox 3" />  
    <Button Content="button 1" />  
    <Button Content="button 2" />  
    <Button Content="button 3" />  
  </StackPanel>  
</Window>

והתוצאה שנקבל:



# **Trigger, DataTrigger & EventTrigger**

**Trigger לא למבחן,**

**אך נחמד להכיר אותו...**

לקוח מהאתר:

<http://www.wpf-tutorial.com/styles/trigger-datatrigger-event-trigger/>

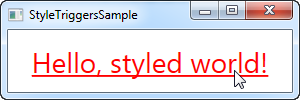
So far, we worked with styles by setting a static value for a specific property. However, using triggers, you can change the value of a given property, once a certain condition changes. Triggers come in multiple flavors: **Property triggers, event triggers and data triggers.** They allow you to do stuff that would normally be done in code-behind completely in markup instead, which is all a part of the ongoing process of separating style and code.

Property trigger

**The most common trigger is the property trigger**, which in markup is simply defined with a <Trigger> element. It watches a specific property on the **owner control** and when that property has a value that matches the specified value, properties can change. In theory this might sound a bit complicated, but it's actually quite simple once we turn theory into an example:

[Download sample](http://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_content=codebox&utm_campaign=wpf-tutorial)

<Window x:Class="WpfTutorialSamples.Styles.StyleTriggersSample"  
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
        Title="StyleTriggersSample" Height="100" Width="300">  
    <Grid>  
        <TextBlock Text="Hello, styled world!" FontSize="28" HorizontalAlignment="Center" VerticalAlignment="Center">  
            <TextBlock.Style>  
                <Style TargetType="TextBlock">  
                    <Setter Property="Foreground" Value="Blue"></Setter>  
                    <Style.Triggers>  
                        <Trigger Property="IsMouseOver" Value="True">  
                            <Setter Property="Foreground" Value="Red" />  
                            <Setter Property="TextDecorations" Value="Underline" />  
                        </Trigger>  
                    </Style.Triggers>  
                </Style>  
            </TextBlock.Style>  
        </TextBlock>  
    </Grid>  
</Window>



In this style, we set the **Foreground** property to blue, to make it look like a hyperlink. We then add a trigger, which listens to the **IsMouseOver** property - once this property changes to **True**, we apply two setters: We change the **Foreground** to red and then we make it underlined. This is a great example on how easy it is to use triggers to apply design changes, completely without any code-behind code.

We define a local style for this specific TextBlock, but as shown in the previous articles, the style could have been globally defined as well, if we wanted it to apply to all TextBlock controls in the application.

Data triggers

Data triggers, represented by the <DataTrigger> element, are used for properties that are **not necessarily dependency properties**. They work by creating a binding to a regular property, which is then monitored for changes. This also opens up for binding your trigger to a property **on a different control**. For instance, consider the following example:

[Download sample](http://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_content=codebox&utm_campaign=wpf-tutorial)

<Window x:Class="WpfTutorialSamples.Styles.StyleDataTriggerSample"  
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
        Title="StyleDataTriggerSample" Height="200" Width="200">  
    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">  
        <CheckBox Name="cbSample" Content="Hello, world?" />  
        <TextBlock HorizontalAlignment="Center" Margin="0,20,0,0" FontSize="48">  
            <TextBlock.Style>  
                <Style TargetType="TextBlock">  
                    <Setter Property="Text" Value="No" />  
                    <Setter Property="Foreground" Value="Red" />  
                    <Style.Triggers>  
                        <DataTrigger Binding="{Binding ElementName=cbSample, Path=IsChecked}" Value="True">  
                            <Setter Property="Text" Value="Yes!" />  
                            <Setter Property="Foreground" Value="Green" />  
                        </DataTrigger>  
                    </Style.Triggers>  
                </Style>  
            </TextBlock.Style>  
        </TextBlock>  
    </StackPanel>  
</Window>



In this example, we have a **CheckBox** and a **TextBlock**. Using a **DataTrigger**, we bind the TextBlock to the **IsChecked** property of the CheckBox. We then supply a default style, where the text is "No" and the foreground color is red, and then, using a DataTrigger, we supply a style for when the IsChecked property of the CheckBox is changed to True, in which case we make it green with a text saying "Yes!" (as seen on the screenshot).

Event triggers

Event triggers, represented by the <EventTrigger> element, are mostly used to trigger an animation, in response to an event being called. We haven't discussed animations yet, but to demonstrate how an event trigger works, we'll use them anyway. Have a look on the chapter about animations for more details. Here's the example:

[Download sample](http://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_content=codebox&utm_campaign=wpf-tutorial)

<Window x:Class="WpfTutorialSamples.Styles.StyleEventTriggerSample"  
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
        Title="StyleEventTriggerSample" Height="100" Width="300">  
    <Grid>  
        <TextBlock Name="lblStyled" Text="Hello, styled world!" FontSize="18" HorizontalAlignment="Center" VerticalAlignment="Center">  
            <TextBlock.Style>  
                <Style TargetType="TextBlock">  
                    <Style.Triggers>  
                        <EventTrigger RoutedEvent="MouseEnter">  
                            <EventTrigger.Actions>  
                                <BeginStoryboard>  
                                    <Storyboard>  
                                        <DoubleAnimation Duration="0:0:0.300" Storyboard.TargetProperty="FontSize" To="28" />  
                                    </Storyboard>  
                                </BeginStoryboard>  
                            </EventTrigger.Actions>  
                        </EventTrigger>  
                        <EventTrigger RoutedEvent="MouseLeave">  
                            <EventTrigger.Actions>  
                                <BeginStoryboard>  
                                    <Storyboard>  
                                        <DoubleAnimation Duration="0:0:0.800" Storyboard.TargetProperty="FontSize" To="18" />  
                                    </Storyboard>  
                                </BeginStoryboard>  
                            </EventTrigger.Actions>  
                        </EventTrigger>  
                    </Style.Triggers>  
                </Style>  
            </TextBlock.Style>  
        </TextBlock>  
    </Grid>  
</Window>



The markup might look a bit overwhelming, but if you run this sample and look at the result, you'll see that we've actually accomplished a pretty cool animation, going both ways, in ~20 lines of XAML. As you can see, I use an EventTrigger to subscribe to two events: **MouseEnter** and **MouseLeave**. When the mouse enters, I make a smooth and animated transition to a FontSize of 28 pixels in 300 milliseconds. When the mouse leaves, I change the FontSize back to 18 pixels but I do it a bit slower, just because it looks kind of cool.

Summary

WPF styles make it easy to get a consistent look, and with triggers, this look becomes dynamic. Styles are great in your application, but they're even better when used in control templates etc. You can read more about that elsewhere in this tutorial.

In the next article, we'll look at multi triggers, which allow us to apply styles based on multiple properties.

# **WPF MultiTrigger and MultiDataTrigger**

**MultiTrigger לא למבחן,**

**אך נחמד להכיר אותו...**

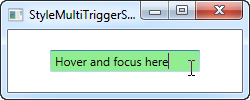
In the previous chapter, we worked with triggers to get dynamic styles. So far they have all been based on a single property, but WPF also supports multi triggers, which can monitor two or more property conditions and only trigger once all of them are satisfied.

There are two types of multi triggers: The **MultiTrigger**, which just like the regular Trigger works on dependency properties, and then the **MultiDataTrigger**, which works by binding to any kind of property. Let's start with a quick example on how to use the MultiTrigger.

MultiTrigger

[Download sample](http://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_content=codebox&utm_campaign=wpf-tutorial)

<Window x:Class="WpfTutorialSamples.Styles.StyleMultiTriggerSample"  
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
        Title="StyleMultiTriggerSample" Height="100" Width="250">  
    <Grid>  
        <TextBox VerticalAlignment="Center" HorizontalAlignment="Center" Text="Hover and focus here" Width="150">  
            <TextBox.Style>  
                <Style TargetType="TextBox">  
                    <Style.Triggers>  
                        <MultiTrigger>  
                            <MultiTrigger.Conditions>  
                                <Condition Property="IsKeyboardFocused" Value="True" />  
                                <Condition Property="IsMouseOver" Value="True" />  
                            </MultiTrigger.Conditions>  
                            <MultiTrigger.Setters>  
                                <Setter Property="Background" Value="LightGreen" />  
                            </MultiTrigger.Setters>  
                        </MultiTrigger>  
                    </Style.Triggers>  
                </Style>  
            </TextBox.Style>  
        </TextBox>  
    </Grid>  
</Window>



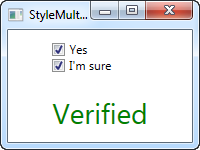
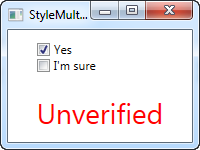
In this example, we use a trigger to change the background color of the TextBox once it has keyboard focus AND the mouse cursor is over it, as seen on the screenshot. This trigger has two conditions, but we could easily have added more if needed. In the Setters section, we define the properties we wish to change when all the conditions are met - in this case, just the one (background color).

MultiDataTrigger

Just like a regular DataTrigger, the MultiDataTrigger is cool because it uses bindings to monitor a property. This means that you can use all of the cool WPF binding techniques, including binding to the property of another control etc. Let me show you how easy it is:

[Download sample](http://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_content=codebox&utm_campaign=wpf-tutorial)

<Window x:Class="WpfTutorialSamples.Styles.StyleMultiDataTriggerSample"  
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
        Title="StyleMultiDataTriggerSample" Height="150" Width="200">  
    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">  
        <CheckBox Name="cbSampleYes" Content="Yes" />  
        <CheckBox Name="cbSampleSure" Content="I'm sure" />  
        <TextBlock HorizontalAlignment="Center" Margin="0,20,0,0" FontSize="28">  
            <TextBlock.Style>  
                <Style TargetType="TextBlock">  
                    <Setter Property="Text" Value="Unverified" />  
                    <Setter Property="Foreground" Value="Red" />  
                    <Style.Triggers>  
                        <MultiDataTrigger>  
                            <MultiDataTrigger.Conditions>  
                                <Condition Binding="{Binding ElementName=cbSampleYes, Path=IsChecked}" Value="True" />  
                                <Condition Binding="{Binding ElementName=cbSampleSure, Path=IsChecked}" Value="True" />  
                            </MultiDataTrigger.Conditions>  
                            <Setter Property="Text" Value="Verified" />  
                            <Setter Property="Foreground" Value="Green" />  
                        </MultiDataTrigger>  
                    </Style.Triggers>  
                </Style>  
            </TextBlock.Style>  
        </TextBlock>  
    </StackPanel>  
</Window>



In this example, I've re-created the example we used with the regular DataTrigger, but instead of binding to just one property, I bind to the same property (IsChecked) but on two different controls. This allows us to trigger the style only once both checkboxes are checked - if you remove a check from either one of them, the default style will be applied instead.

Summary

As you can see, multi triggers are pretty much just as easy to use as regular triggers and they can be extremely useful, especially when developing your own controls.