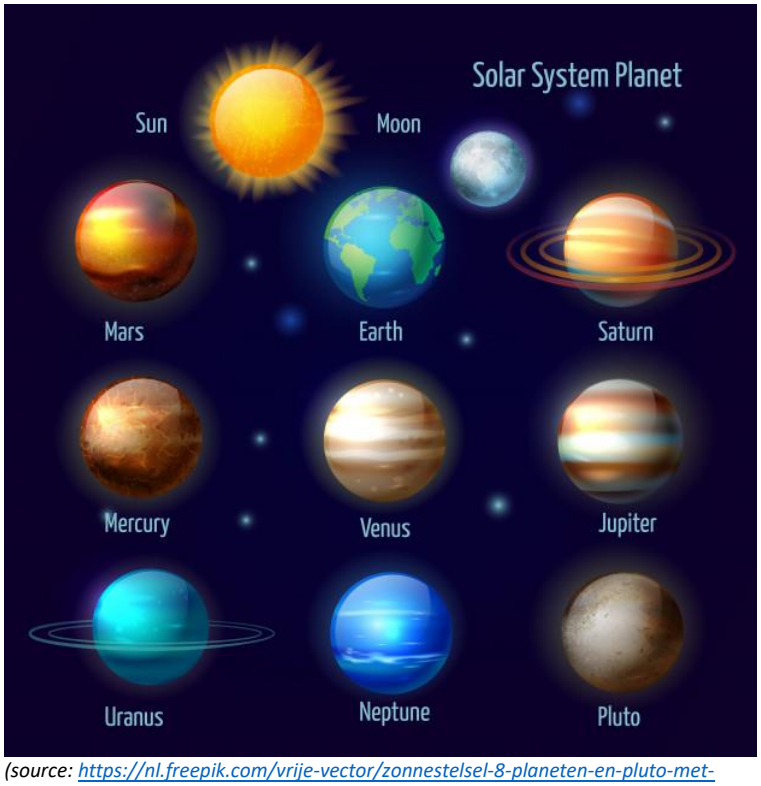
Exercise: Creating a Visual Solar System using WPF

version 20.07.16

Contents

Introduction

This exercise is meant to practise Windows Presentation Foundation (WPF) (Gradient) Brushes, Styles and Resource Dictionaries. We will create part of our solar system, using this picture as a drawing guide. As you can see, a lot of celestial bodies could be drawn using some kind of gradient coloring. We can make good use of this fact while practising our gradient brush skills. A UserControl will be used to depict the necessary celestial bodies so we can use two kinds of gradient brushes in one control.



(source: https://nl.freepik.com/vrije-vector/zonnestelsel-8-planeten-en-pluto-met-zonpictogrammen-instellen-astronomische-poster_2873121.htm)

Part 1: Preparing the Project

1. In Visual Studio, create a new **WPF App (.NET Framework)**, naming it *SolarSystem*.
2. Inside this project, create these two folders:
    - ResourceDictionaries
    - UserControls

Part 2: Adding a UserControl to the Project

Creating a *UserControl* is suitable if you want to build your control by adding *existing* elements to it, similar to how you build an application. The two *existing* elements that will be used as part of the UserControl are two same-shaped ellipses stacked on top of each other, each painted with different gradient brushes. The first ellipse will be allowed to be filled with a (linear gradient) brush by the designer/programmer. The ellipse on top will use a *RadialGradientBrush* with a *GradientOrigin* in the direction of the top left corner, giving it an *orb*-like appearance. This last ellipse will also be partly *transparent* so the bottom ellipse is visible, too.

1. Via *Solution Explorer*, select folder *UserControls* and *add* a *new item* of type **User Control (WPF)**. Enter **CelestialBody.xaml** for its name.

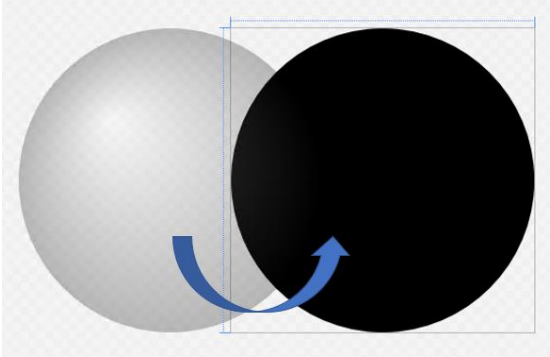
2. Make sure the XAML inside our added control looks as follows:

```
<UserControl ...
             MinWidth="50" MinHeight="50">
  <Grid>

    <Ellipse Fill="{Binding Foreground, RelativeSource={RelativeSource AncestorType=UserControl}}"/>

    <Ellipse Margin="-35,0,35,0">
      <Ellipse.Fill>
        <RadialGradientBrush GradientOrigin="0.3,0.3" Opacity="0.25">
          <GradientStop Color="Black" Offset="1"/>
          <GradientStop Color="White"/>
        </RadialGradientBrush>
      </Ellipse.Fill>
    </Ellipse>

  </Grid>
</UserControl>
```



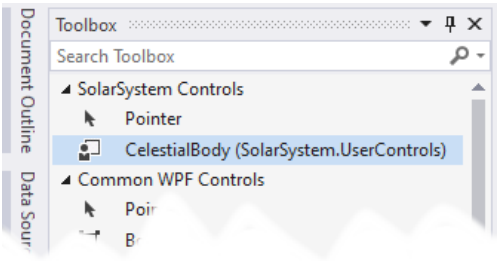
In the designer a black circle is visible now. This is because the `Binding` used in the first Ellipse uses the `Foreground` brush that is set to the UserControl 'from the outside', which defaults to *#FF000000* shown as *black*. For a better comprehension, the picture shows the two ellipses shifted more side-by-side to make the partly transparent radial gradient brush visible.

3. Build your solution (e.g. via menu *Build* > *Rebuild Solution*).

In general, if something does not look as expected while developing a *WPF* application, *rebuild*..!

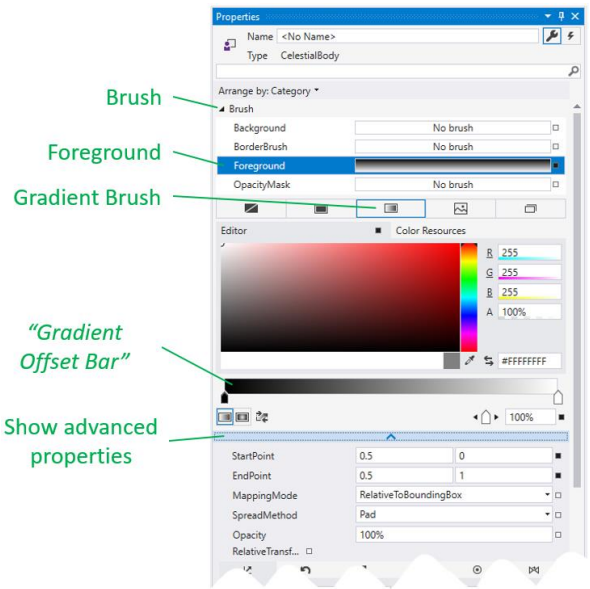
Part 3: Using the UserControl

Add the CelestialBody Control to the Window



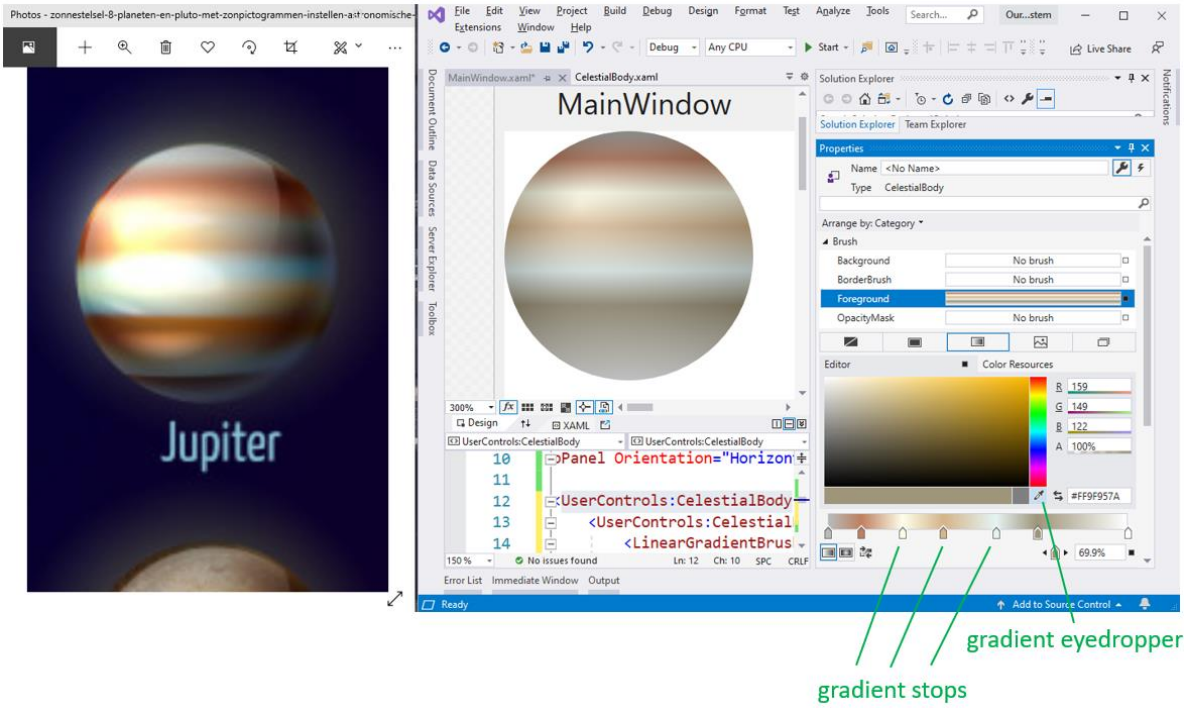
1. Open MainWindow.xaml and replace the `Grid` by a *horizontally oriented* `WrapPanel`.
2. After *building* the solution earlier, the UserControl should be visible in the *Toolbox*, as shown in the picture. Drag an instance of the CelestialBody control onto the WrapPanel. Rebuild the solution if an exclamation mark appears in the designer.

Create a Gradient Brush



1. In the MainWindow.xaml designer, zoom ellipse in such a way you can more easily design its gradient brush, and select the UserControl.
2. In the *Properties* Panel (if not visible, press the [F4] keyboard key) open category *Brush*, select *Foreground*, select the small *Gradient Brush icon* and *unfold* the *advanced properties*, as shown in the picture.
3. Now use the *Solar System Planet* picture from earlier in this exercise to pick a celestial body and try to mimic the horizontal stripes using the "Gradient Offset Bar", as follows (the next *bigger* image shows the position of the *gradient stops* and the *eyedropper* tool).
    a. select the first (now black) so-called *gradient stop* at the left end of the "Gradient Offset Bar"
    b. select the *eyedropper* and pick a color from the top of the 'real' planet picture
    c. To add a new *gradient stop*, click inside the "Gradient Offset Bar" a little to the right of the first *gradient stop* (they can be moved by dragging left/right and deleted using the [Delete] keyboard key if desired)

d. select the *eyedropper* again and pick the next desired color from the top of the 'real' planet picture
e. repeat the steps *c* and *d* until you reach the right end of the bar



Add a ResourceDictionary

1. In folder *ResourceDictionary*, add a new item of type **ResourceDictionary (WPF)** naming it *BrushDictionary.xaml*
2. In the *XAML* part of MainWindow.xaml a complete `LinearGradientBrush` was generated during the previous *color picking process*. Cut out that entire `<LinearGradientBrush>` element and paste it inside the BrushDictionary.xaml file. Add a `x:Key` attribute to it such as *jupiterBrush*.
3. From file MainWindow.xaml remove the lines that give you *scribbly warnings*. You could now turn the `<UserControl:CelestialBody>` element into an empty element.

Configure an Use a MergedDictionary

1. Although we have created only one separate ResourceDictionary so far, we still need to add and configure the `MergedDictionaries` element in the *App.xaml* file.
   Hint: use demo 🥸 *WPF101 > App.xaml* to see how to make our *BrushDictionary.xaml* available to the whole project.
2. In MainWindow.xaml remove both `Height` and `Width` from the `<UserControls:CelestialBody>` element.
3. Add the following attribute to that same UserControl to see our planet in its full glory:
   `Foreground="{StaticResource jupiterBrush}"`

Create more Celestial Body Brushes

- Repeat the steps of *Part 3* to create linear gradient brushes for other celestial bodies and add them to the *BrushDictionary.xaml* file, naming them after the planet they apply to.

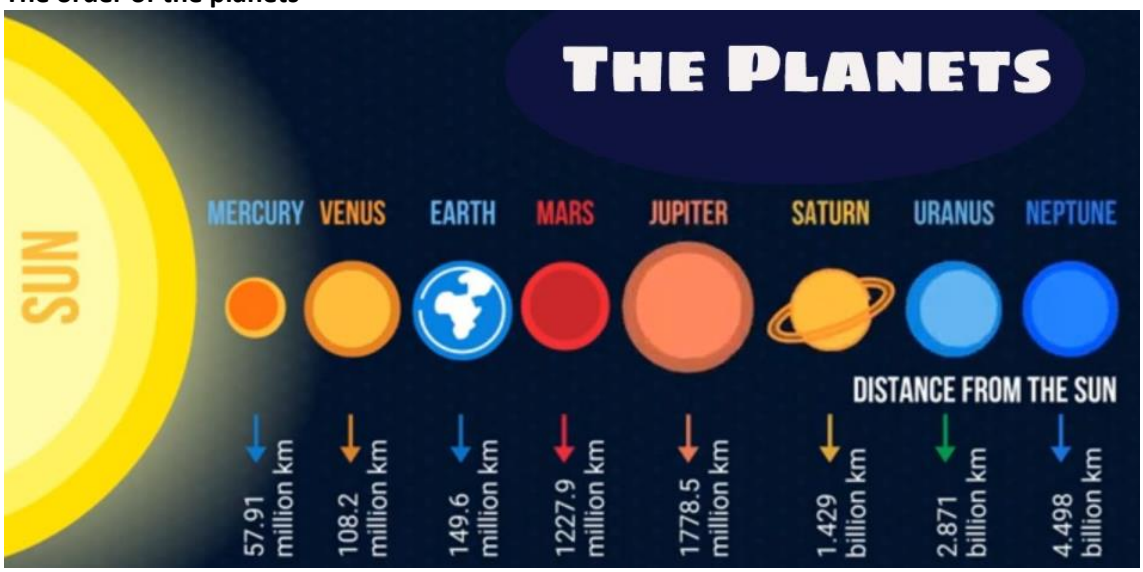
The following data was taken from https://www.universetoday.com/36649/planets-in-order-of-size/ and might be helpful to determine the size of the celestial bodies. Just as an indication, because our screens would not be large enough if all these values where used to the letter 😊

According to NASA, these are the estimated radii of the eight planets in our solar system, in *order of size*:

- Jupiter (69,911 km / 43,441 miles) – 1,120% the size of Earth
- Saturn (58,232 km / 36,184 miles) – 945% the size of Earth
- Uranus (25,362 km / 15,759 miles) – 400% the size of Earth
- Neptune (24,622 km / 15,299 miles) – 388% the size of Earth
- Earth (6,371 km / 3,959 miles)

- Venus (6,052 km / 3,761 miles) – 95% the size of Earth
- Mars (3,390 km / 2,460 miles) – 53% the size of Earth
- Mercury (2,440 km / 1,516 miles) – 38% the size of Earth

The order of the planets



(source: https://youtu.be/VczTjSF_Puw)

Part 4: Styling the UserControl

The celestial bodies in our solar system are not equal in size at all, neither is the spacing between them. We can use *WPF Styles* to make our solar system look a little more realistic, although it will never display the actual units…

1. In folder *ResourceDictionary*, add a new item of type **ResourceDictionary (WPF)** naming it *StyleDictionary.xaml*
2. Make sure to add this dictionary to the `MergedDictionaties` in file App.xaml
3. In file *StyleDictionary.xaml* add some kind of "baseStyle" that contains properties that are to be applied to all celestial bodies, like e.g. `Margin` if you choose to equalize the distance between them.
   Hint: use demo 🕶 *WPF101 > StylesDemo* if you need an example.
4. Still in file *StyleDictionary.xaml* add a specific style per celestial body, basing it on the appropriate *baseStyle*, if any was created in the previous step. These specific styles should set properties for `Width` and `Height` and for `Foreground`, using the brushes created earlier. Add `x:Key` attributes to it like *jupiterStyle*, etc.
5. Switch to MainWindow.xaml and change all `Foreground` attributes to be `Style` attributes referencing the *styles* just created.
   Hint: the page *Karel* suggested during the previous lesson could come in handy here: https://www.meziantou.net/visual-studio-tips-and-tricks-multi-line-and-multi-cursor-editing.htm

Part 5: Adding an Exceptional ToolTip (Bonus, If Time Permits)

Create a nice *tooltip* (not just a *one-liner*) that is displayed as the mouse arrow hovers over each of the celestial bodies, displaying lots of data about that item, maybe showing an image, etc. Inside *StyleDictionary.xaml* you can add the following per `Style` and go *crazy* using https://www.wpf-tutorial.com/control-concepts/tooltips/ as a starting point…

```
<Setter Property="ToolTip">
  <Setter.Value>
    ...
  </Setter.Value>
</Setter>
```

Part 6: Turning User Controls into Templated Buttons

In order to demonstrate some important WPF subjects like Control Templates, Commands, Routed Events and Data Binding, we will change our CelestialBody User Control into a Button Control Template, complete with some Triggers.

Add another ResourceDictionary

1. Add a new file of type *Resource Dictionary (WPF)* to folder *ResourceDictionaries*, naming it **ButtonStyleDictionary.xaml** and make sure to add this dictionary to the *App.xaml* file.
2. Add the following Style to the *ButtonStyleDictionary.xaml* file. The part declaring the two Ellipses was copied from *CelestialBody.xaml*. Only the *Template Binding* for the first Ellipse has changed, because it is more common to use our *Gradient Brushes* as *Background* for the Button now, leaving Foreground White in this case to show the planet name on the Button.
   Demo 👓 *WPF102 > ControlTemplatesDemo* contains an example of how to create `ControlTemplate.Triggers`

```
<Style TargetType="Button">
  <Setter Property="ToolTipService.ShowOnDisabled" Value="True"/>
  <Setter Property="Margin" Value="30"/>
  <Setter Property="RenderTransformOrigin" Value="0.5,0.5"/>
  <Setter Property="Foreground" Value="White"/>
  <Setter Property="FontSize" Value="20"/>
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="Button">
         <Grid MinWidth="30" Width="{TemplateBinding Width}"
               MinHeight="30" Height="{TemplateBinding Height}">

           <Ellipse Fill="{TemplateBinding Background}"/>

           <Ellipse x:Name="frontEllipse" Margin="0,0,0,0">
             <Ellipse.Fill>
               <RadialGradientBrush GradientOrigin="0.3,0.3" Opacity="0.25">
                 <GradientStop Color="Black" Offset="1"/>
                 <GradientStop Color="White"/>
               </RadialGradientBrush>
             </Ellipse.Fill>
           </Ellipse>

           <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center" />

         </Grid>

         <ControlTemplate.Triggers>
           <!-- create your own triggers here as seen in WPF102 > ControlTemplatesDemo -->
         </ControlTemplate.Triggers>
       </ControlTemplate>
     </Setter.Value>
  </Setter>
</Style>
```

Add another Window

1. Add a new Window (WPF) file to the project called **PlanetsAsButtons.xaml** and make it the `StartupUri`.
2. In this new Window, inside the `Grid` add a *horizontally oriented* `WrapPanel` and add a button for every planet, like so
   ```
   <Button Content="Mercury" />
   <Button Content="Venus" />
   ```
3. Because we did not put an `x:Key` attribute on our Button Style in *ButtonStyleDictionary.xaml* file yet, the following should be visible on our *PlanetsAsButtons.xaml* design window:

Getting our Style Back

1. To get our Styles back we need to copy and adjust large parts from file *StyleDictionary.xaml* to *ButtonStyleDictionary.xaml* as follows:
    a. Copy all entire planet styles that are named `x:Key="mercuryStyle"`, `x:Key="venusStyle"`, etc. until and including your last style and paste them below the one present in file *ButtonStyleDictionary.xaml*.
    b. Add an `x:Key="buttonBaseStyle"` to the first style present (the one with the `ControlTemplate`).
    c. Rename all planet styles from e.g. *mercuryStyle* to *buttonMercuryStyle*, etc. to avoid name clashes with our previously created UserControl Styles (like when using CSS in web development, the last style defined, closest to the control at hand wins).
    d. Change all planet styles to be based on this *buttonBaseStyle*.
    e. Make sure all planet styles target the *Button* type.
    f. Finally alter the `Setter Property="Foreground"` to be `Setter Property="Background"`.
2. In our *PlanetsAsButtons.xaml* window, we need to explicitly pick a style for all planet buttons to get our original look back, like so:
```
<Button Content="Mercury" Style="{StaticResource buttonMercuryStyle}" />
<Button Content="Venus" Style="{StaticResource buttonVenusStyle}" />
```
3. Rebuild if necessary to get the planets back as we were used to see them. The only difference is that they display content, being the name of that planet.

Part 7: Applying the Command Pattern

In the future we would like to be able to edit planet data after clicking on that planet, but we also want to offer that functionality via a menu and tool bar buttons. This is were the Command Pattern will help us.

Demo 🕶 *WPF102 > CommandPatternDemo* contains an example of how this can be achieved

Add a Custom Command

1. Add a folder named *Commands* and add a class named *CustomCommands.cs* to it. Call the RoutedUICommand *EditPlanet* and name the required text strings accordingly. See the demo mentioned before and https://www.wpf-tutorial.com/commands/implementing-custom-commands/ on how to create one.
2. In *PlanetsAsButtons.xaml* reference the *Commands* namespace just created using `xmlns:cmd="..."`

Add a Menu

1. In *PlanetsAsButtons.xaml* change the `Grid` to a `DockPanel`.
2. Add a Menu control as seen in the *CommandPatternDemo* having *menu items* with `Headers` File > Exit and Edit > Mercury, Venus, etc.

Add a Toolbar

1. In *PlanetsAsButtons.xaml* add a ToolbarTray with a Toolbar below the menu. See the demo and https://www.wpf-tutorial.com/common-interface-controls/toolbar-control/ on how to do that.
2. To be able to enable/disable menu items, toolbar buttons and buttons later on, add the following XAML to the toolbar just added:
```
<Grid>
  <CheckBox x:Name="enableEditingCheckBox" VerticalAlignment="Center" Margin="2">
    Enable Editing
  </CheckBox>
</Grid>
<Separator/>
<!-- rest of the toolbar -->
```

Configure the Command Bindings

1. Add the following `CommandBindings` to *PlanetsAsButtons.xaml*:
```
<Window.CommandBindings>

  <CommandBinding Command="ApplicationCommands.Close"
                  Executed="Close_Executed" />

  <CommandBinding Command="cmd:CustomCommands.EditPlanet"
                  Executed="EditPlanet_Executed"
                  CanExecute="EditPlanet_CanExecute"/>
```

```
</Window.CommandBindings>
```

2. Add these three belonging methods to the code behind file *PlanetsAsButtons.xaml.cs*:

```
private void Close_Executed(object sender, ExecutedRoutedEventArgs e) => Close();

private void EditPlanet_Executed(object sender, ExecutedRoutedEventArgs e)
{
  MessageBox.Show($"The data of planet {e.Parameter} could be edited here...");
}

private void EditPlanet_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
  e.CanExecute = (bool)enableEditingCheckBox.IsChecked;
}
```

Adjust the Controls to use the Configured Commands

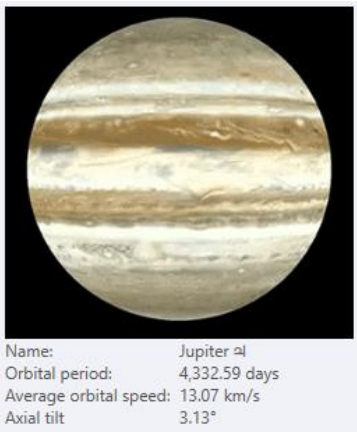
The only thing that needs to be done is letting the *Menu Items*, *Toolbar Buttons* and *Planet Buttons* use the `Command` and setting the correct `CommandParameter`.

1. Add the following `Command` attribute to each relevant *Menu Item*, *Toolbar Button* and *Planet Button*:
   `Command="cmd:CustomCommands.EditPlanet"`
2. To be able to determine which planet is meant, a `CommandParameter` will tell them apart. Without the need to enter the name of the planet multiple times in one control as an attribute, we can use this attribute on every *Menu Item* to get things working:
   `CommandParameter="{Binding RelativeSource={x:Static RelativeSource.Self}, Path=Header}"`
3. This XAML attribute is needed on all relevant *Buttons*:
   `CommandParameter="{Binding RelativeSource={x:Static RelativeSource.Self}, Path=Content}`
4. Build, run and test the application. Make sure the enabling/disabling of the controls is dependend on the ToolBar CheckBox choice.

Make Use of Style Once More (Bonus, If Time Permits)

- Instead of copying the above `Command` and `CommandParameter` attributes to all relevant controls as indicated above, try to make use of *Styling* to minimize duplicating similar attributes one more time.

Part 8: Binding the Celestial Body Data



Create a Planet Data Class

1. Create a *new folder* named **Models**.
2. Based on the information shown in your *ToolTip*, create a class named **CelestialBody.cs** in the *folder* just created. The class only needs to contain auto-implemented properties to hold the desired planet data. Make sure at least the properties integer `Id` and string `Name` are present.

Create a Test Repository

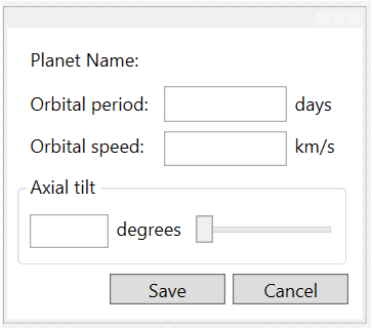
1. Create a *new folder* named **Repositories**.
2. Add a new `static` class to that folder named **CelestialBodyRepository.cs**.
3. Declate a `private List` of type CelestialBody inside the class.
4. In the class *constructor*, populate the `List` with new *CelestialBody* objects and fill their properties with the values used to fill the ToolTips earlier.
5. Add a method named `GetCelestialBodyByName` that does what the name promises 😊
6. Add a method names `SaveCelestialBody` that stores a passed *CelestialBody* object in its correct `List` slot.

Create a Planet ViewModel

1. Create a *new folder* named **ViewModels**.
2. Add a new class named **PlanetViewModel.cs** to that folder.
3. Make sure the class implements the **INotifyPropertyChanged** interface.
4. Add a `SetProperty` method as shown in Demo 🥸 *WPF103 > Models > CitizenINPC* present inside the `#region Setter Helper Method`

5. Add a so-called *Full Property* (i.e. getter/setter plus backing field) to the *PlanetViewModel* class for each propery present in the *CelestialBody* class. Call the `SetProperty` method from each of the property `set` parts.
6. Add a `private void` method called **PopulatePlanetVM** to the class, with a string parameter called **planetName**. Inside this method perform the following steps:
   a) Call the *GetCelestialBodyByName* method on the *CelestialBodyRepository* using the *planetName* parameter as argument.
   b) Copy every property of the planet *returned* by the repository to the belonging property of the current *PlanetViewModel* class.
7. Add a `public void` method called **SaveThePlanet** to the class. This method will transport the *ViewModel* property data back to the *repository*. Perform the following steps to get this task done:
   a) Create an instance of the CelestialBody class and copy each *ViewModel* property into the corresponding property of that instance.
   NOTE: This is in fact the *opposite* of what is being done in method *PopulatePlanetVM*.
   b) Call the *SaveCelestialBody* of the *CelestialBodyRepository* passing it the instance created at step a).
8. Finally add a *constructor* to the class that takes a `string` parameter called *planetName*. Add code to this constructor to call the *PopulatePlanetVM* method using *planetName* as argument if it contains a value.

Create an Edit Planet Info View



1. Create a *new folder* named **Views**.
2. Add a new **Window (WPF)** item to that folder, called **EditPlanetInfoView.xaml**.
3. Design a *user interface similar* to this picture, but by using the properties of your own **PlanetViewModel** class. Be creative 😊
4. Demo 🥸 *WPF103 > BindingToSingleObjectDemoINPC* contains XAML examples of how to *DataBind* controls in XAML to *ViewModel* properties. Make sure to type those bindings correctly 😐
   NOTE: the *DataContext* will be *configured* in a later step.
5. Make sure to `x:Name` the *Save* Button `saveButton` and the *Cancel* Button `cancelButton`. *Double-click* both in turn to generate an event handler for each in the code behind page.
6. Here are some convenient *UI* tips:
   - If a `Button` needs to respond to the [Enter] keyboard key, set its `IsDefault` property to `True`.
   - If a `Button` needs to respond to the [Esc] keyboard key, set its `IsCancel` property to `True`.
   - The `Title` property of a `Window` can be *databound*, too.
7. Make sure in file **EditPlanetInfoView.xaml.cs** under the *constructor* the code looks as follows. Bring classes *into scope* if necessary e.g. by using the light *bulb*.

```
public EditPlanetInfoView(string currentPlanetName) : this()
{
      DataContext = new PlanetViewModel(currentPlanetName);
}

private void saveButton_Click(object sender, RoutedEventArgs e)
{
  (DataContext as PlanetViewModel).SaveThePlanet();

  Close();
}

private void cancelButton_Click(object sender, RoutedEventArgs e)
{
   Close();
}
```

Activate and Connect the View

1. From *Solution Explorer* open file *PlanetsAsButtons.xaml.cs* and make sure the following code is the only code inside the `EditPlanet_Executed` method:

```
var dialog = new EditPlanetInfoView(e.Parameter.ToString());
dialog.ShowDialog();
```

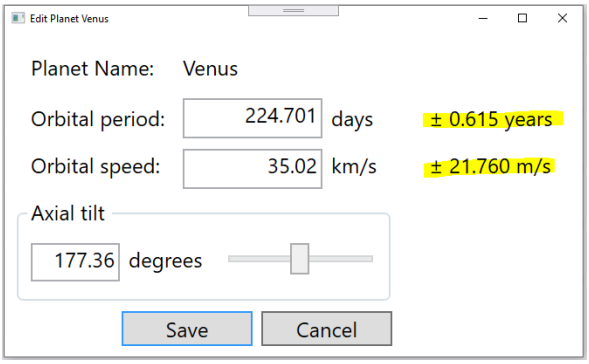
2. Build, run and test the application. Check the [Enable Editing] CheckBox and edit data of some of the planets. See whether they are stored correctly.

The *ToolTips* are not updated, though. This will be done later in the exercise, when we try to approach the Model–view–viewmodel (MVVM) software architectural pattern more closely…

Part 9: Creating and Applying a ValueConverter (Bonus, If Time Permits)

A couple of things could be improved using a *ValueConverter*:

- As you can see in this picture, the planet names all use the same font size. This makes some characters invisible while other names could be written bigger to use the space more efficiently. You could size the font relative to e.g. the `Width` of the Button.
- Some values on the View could be rewritten in another *unit of measure*, as marked in *yellow* in the screenshot.



Create a new folder *ValueConverters* with a new class called **MultiplierConverter.cs** implementing the `IValueConverter` interface. Demo 👓 *WPF103 > ValueConverterDemo* and the belonging *ValueConverter* classes in the *ValueConverters* folder could be used as an example.

Some hints:

- Use a `ConverterParameter` to pass in the *multiplier* value.
- Use a *convert* statement that takes the *decimal point* into account, such as:
  ```
  if (!double.TryParse(parameter?.ToString(), NumberStyles.Number, CultureInfo.InvariantCulture,
  out multiplier))
  ```
- Use a `TextBlock` that has a `Text` property which you can *StringFormat*, because a `Label` has a `Content` property on which *StringFormat* does not seem to be working.
- Information on how to use the `StringFormat` property can be found here: https://www.wpf-tutorial.com/data-binding/the-stringformat-property/
- The `Style` part for sizing the planet name font could look something like this:
  ```
  <Setter Property="FontSize" Value="{Binding RelativeSource={RelativeSource self}, Path=Width,
  Converter={StaticResource multiplierConverter}, ConverterParameter=0.25}"/>
  ```

Part 10: Introducing and Using BindableBase

In one of the next steps we will create another ViewModel that also needs to implement *INotifyPropertyChanged*. Common practise for MVVM is to create a base class for all ViewModels, called *ViewModelBase* or *BindableBase*.

1. In folder ViewModels add a class called *BindableBase*. Because this is pretty standard code, a nice example of this class can be found here https://www.danrigby.com/2015/09/12/inotifypropertychanged-the-net-4-6-way/
2. Copy the entire `public abstract class BindableBase` implementation (situated under the line *"Using these features, our implementation of the BindableBase class now looks like this:"*).
3. In file *PlanetViewModel.cs*, make sure class *PlanetViewModel* derives from BindableBase, so remove *INotifyPropertyChanged*.
4. Also comment/remove the *INotifyPropertyChanged* implementation, so both the `event` declaration and the `SetProperty` method.

Part 11: Adding and Removing certain Parts

Maintenance on the Repository

1. Our *CelestialBodyRepository* class needs a method returning the *entire planet list*. This could look as follows:
   `public static List<CelestialBody> GetCelestialBodies() => _celestialBodies;`
2. The repository method `GetCelestialBodyByName` is no longer necessary and may be commented/removed.

Maintenance on the ViewModel

1. From class *PlanetViewModel* the overloaded constructor `public PlanetViewModel(string planetName)` may be commented/removed.
2. Instead add these lines of code to enable the *yet to be created* second ViewModel to pass a reference of itself to this ViewModel. NOTE: these lines *will* give errors until *Part 12* has been completed.

```
private readonly PlanetCollectionViewModel _planetCollectionViewModel;

public PlanetViewModel(PlanetCollectionViewModel planetCollectionViewModel)
{
  _planetCollectionViewModel = planetCollectionViewModel;
}
```

3. Still in class *PlanetViewModel* method *PopulatePlanetVM* may be commented/removed as well.
4. The following property needs to be added to class *PlanetViewModel* to return the `x:Key` of the style belonging to the planet. Depending on the naming of your planet styles, this could look as follows:
   `public string PlanetStyle { get => $"button{Name}Style"; }`

Maintenance on the Views

Because we try to solve all *UI matters* in XAML and in the ViewModels, we can tidy up the *code-behind* files of our Views in an orderly fashion, like so:

1. In file *EditPlanetInfoView.xaml.cs* comment/remove all methods except the constructor containing the call to `InitializeComponents`.
2. In file *EditPlanetInfoView.xaml* from XAML remove the `Click` attributes from both the *saveButton* and *cancelButton*.
3. In file *PlanetsAsButtons.xaml.cs* comment/remove all methods except the constructor containing the call to `InitializeComponents`.
4. In file *PlanetsAsButtons.xaml.cs* comment/remove the complete `<Window.CommandBindings>` element. This part will be taken over by the *RelayCommand* added later.

Part 12: Introducing a ViewModel for the Planet Collection

1. In folder *ViewModels* add a class called **PlanetCollectionViewModel**. Make sure it is declared `public` and inherits from *BindableBase*.
2. Add a public propery of type `ObservableCollection` that will hold `PlanetViewModel` instances and name it **PlanetCollectionVM**. Instantiate that collection in one go.
3. Add the following method to populate this *ObservableCollection*:

```
private void PopulateVM()
{
   PlanetCollectionVM.Clear();

   List<CelestialBody> list = CelestialBodyRepository.GetCelestialBodies();

   foreach (CelestialBody c in list)
   {
     PlanetCollectionVM.Add(
       new PlanetViewModel(this)
       {
         Id = c.Id,
         Name = c.Name,
         /* do this for all properties concerned */
       }
     );
   }
```

```
    }
```

4. Call this *PopulateVM* method from the *default* constructor.

Part 13: Modifying the Current Main View

Instead of the *hard-coded* Buttons representing planets, we will generate these Buttons based on the properties exposed by our ViewModels. These are the steps to be taken to *remodel* our View:

1. In file PlanetsAsButtons.xaml add a DataContext and bring the namespace in scope using the *light bulb*.

```
<Window.DataContext>
  <vm:PlanetCollectionViewModel/>
</Window.DataContext>
```

In the same file comment/remove the entire `<WrapPanel>` element and replace that code using this piece of XAML:

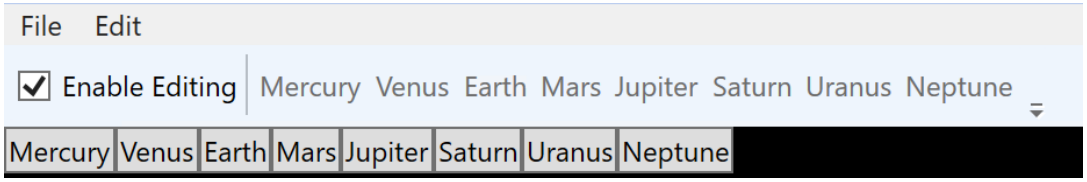
```
<ItemsControl ItemsSource="{Binding PlanetCollectionVM}">

  <!-- sets the template that defines the panel that controls the layout of items -->
  <ItemsControl.ItemsPanel>
    <!-- specifies the panel that the ItemsPresenter creates for laying out ItemsControl items -->
    <ItemsPanelTemplate>
      <WrapPanel Orientation="Horizontal"/>
    </ItemsPanelTemplate>
  </ItemsControl.ItemsPanel>

  <!-- sets the DataTemplate used to display each item -->
  <ItemsControl.ItemTemplate>
    <DataTemplate>
      <Button Content="{Binding Name}"
              Command="{Binding EditPlanetCommand}"
              CommandParameter="{Binding Name}" />
    </DataTemplate>
  </ItemsControl.ItemTemplate>

</ItemsControl>
```

At this moment the following should be visible in the designer (after a *rebuild*). It may look disappointing, but this time the buttons are placed on the View in a *data-driven* way. The missing styles will be added soon 😊



Part 14: Adding a ValueConverter to bring Back our Styles

Unfortunately we are not able to bind our styles directly via the *ResourceKey* property of a *Static (or Dynamic) Resource*. This is because a binding can only be set to a *DependencyProperty* of a *DependencyObject* and *Static/DynamicResource properties* do *not* derive from *DependencyObject*. So we need to create a simple ValueConverter to perform this.

1. In folder *ValueConverters* add a class called **ResourceKeyToResourceConverter**.
2. Make sure to implement the *IValueConverter* as e.g. shown in Demo 👓 *WPF103 > ValueConverterDemo* and the belonging *ValueConverter* classes in the *ValueConverters* folder.
3. The *ultimate purpose* of the `Convert` method is to return the sought after resource value using this piece of code: `System.Windows.Application.Current.FindResource(/* resourceKey value */);`

4. The `ConvertBack` method may just return `DependencyProperty.UnsetValue`.
5. In file *PlanetsAsButtons.xaml* bring this new converter into scope and instantiate it in the resources part of XAML.

6. Now we can add this line of code to the `Button` in the `DataTemplate`:
   ```
   Style="{Binding PlanetStyle, Converter={StaticResource resourceKeyToResourceConverter}}"
   ```

7. After a *rebuild* the desired look-and-feel of our planets should be back now..!

Part 15: Implementing the RelayCommand

Earlier we commented/deleted the <Window.CommandBindings> element because this way of working is not suitable when following the MVVM pattern, because of its dependency of direct events and their belonging event handlers being present in the *code-behind* file of the View at hand. From now on we will be using the so-called *RelayCommand* to define and handle our *Commands* which is a common practice when using MVVM. More on this can be found here: https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern#relaying-command-logic

Add the RelayCommand Class

1. In folder *Commands* add a class called **RelayCommand**.
2. The contents of this class this is pretty standard again and can be found via the URL mentioned above. So copy and paste the entire class situated under header *"Figure 3 The RelayCommand Class"*.

Add Properties that return ICommand

1. In our *PlanetViewModel* class we need to add some *plumbing* code to be able to use the *RelayCommand*. Now we can add a `#region Commands` and make sure these lines are present:
   a) Backing fields for the desired commands:
   ```
   private RelayCommand _editPlanetCommand;
   private RelayCommand _savePlanetCommand;
   private RelayCommand _cancelPlanetCommand;
   ```

   b) The properties the View can bind to:
   ```
   public ICommand EditPlanetCommand { get => _editPlanetCommand ?? (_editPlanetCommand = new
     RelayCommand(EditPlanetExecute, _planetCollectionViewModel.EditPlanetCanExecute)); }

   public ICommand SavePlanetCommand { get => _savePlanetCommand ?? (_savePlanetCommand = new
     RelayCommand(SavePlanetExecute)); }

   public ICommand CancelPlanetCommand { get => _cancelPlanetCommand ?? (_cancelPlanetCommand =
     new RelayCommand(CancelPlanetExecute)); }
   ```

   c) And the beloning implementation methods:
   ```
   private void EditPlanetExecute(object parameter)
   {
     var dialog = new EditPlanetInfoView();
     dialog.DataContext = this;
     dialog.ShowDialog();
   }

   private void SavePlanetExecute(object parameter)
   {
     SaveThePlanet();

     (parameter as Window)?.Close();
   }

   private void CancelPlanetExecute(object parameter) => (parameter as Window)?.Close();
   ```

2. The *EditPlanetCanExecute* method is still missing as you can see. This needs to be added to the *PlanetCollectionViewModel*. First we need to add a property for our *CheckBox* entitled *Enable Editing* to bind to. So open file *PlanetCollectionViewModel.cs* and add the following property and belonging *backing field*:

   ```
   private bool _enableEditing = true;
   ```

```
public bool EnableEditing
{
  get => _enableEditing;
  set => SetProperty(ref _enableEditing, value);
}
```

3. And finally the missing method:
```
public bool EditPlanetCanExecute(object parameter) => EnableEditing;
```

Add or Change the Command Bindings in XAML

In order to use the *ICommand* returning properties exposed by our ViewModels, we need to change or add *Command* attributes in XAML View windows to become data bindings.

1. Open file EditPlanetInfoView.xaml and add these attributes to the *save/cancel* Buttons respectively:
```
Command="{Binding SavePlanetCommand}" CommandParameter="{Binding
ElementName=EditPlanetInfoViewWindow}"

Command="{Binding CancelPlanetCommand}" CommandParameter="{Binding
ElementName=EditPlanetInfoViewWindow}"
```

2. The `x:Name` attributes could be removed from both buttons, but the `Window` itself needs a name, as seen in the `ElementName` above. So add this to the `Window` root element: `x:Name="EditPlanetInfoViewWindow"`
3. In order to get the *Enable Editing CheckBox* working again, open file *PlanetsAsButtons.xaml* and change the `IsChecked` property of the CheckBox to: `<CheckBox IsChecked="{Binding EnableEditing}" ...`
4. Build, run and test the application. Check the workings of the *[Enable Editing*] CheckBox and edit data of some of the planets.

The *planet-related* Menus and Toolbar Buttons do not work yet. That might be dealt with at some point in the future.

Hook up the Exit Menu

The menu File > Exit working will be fixed now. TIP: The steps to take are the same as for the *Cancel* Button on the *EditPlanetInfoView.xaml* file, so cheating is allowed 😊
Here are the global steps with which you can make the menu item work again:

1. Give the whole *PlanetsAsButtons.xaml* window an `x:Name` via XAML
2. In file *PlanetCollectionViewModel.cs* add a *backing field* for the desired Exit *RelayCommand*.
3. In that same file add a *Getter Property* instantiating *and/or* returning the *RelayCommand*.
4. Still in that file add the belonging *…Execute* method to finally close the window.
5. In file *PlanetsAsButtons.xaml* change `MenuItem` *Command* attribute in order to use the property just created. Do NOT forget to add a the belonging **CommandParameter** 😐
6. Build, run and test the application. Make sure the *Exit* menu works.

Part 16: Making the ToolTips Data Aware

After the many steps we took to move towards the MVVM pattern, we can now reap the benefits of our labour. We can do a decent *cleanup* in our *ButtonStyleDictionary* when we are *databinding* our *ToolTip* info.

1. Open file *ButtonStyleDictionary.xaml* and copy the entire `<Setter Property="ToolTip">` element of one of the planets to the *buttonBaseStyle*.
2. Remove/comment all `<Setter Property="ToolTip">` element for the particular planets.
3. Adjust the *hard-coded* planet info to be databound to their belonging properties (the same as used to bind to in file *EditPlanetInfoView.xaml*) and use `StringFormat` if desired to display units of measure.
4. In case of other *bindable* info *candidates*, like e.g. *AxialTilt* to perform a `RotateTransform`, copy one `Setter Property` to the *buttonBaseStyle*, databind it and comment/remove all other occurrences from the planet styles.
5. After these steps there are only a few `Setter Property` lines per planet left.

I discovered too late that although a *Cancel* Button is present, these is nothing to be cancelled since we are directly editing the planet objects in the *ObservableCollection* ☹
Again, that might be dealt with at some point in the future.

Part 17. Making sure no Negative Numbers are Entered (Bonus, If Time Permits)

Add a folder *ValidationRules* and add a class called **PositiveNumberValidationRule**. This small article may be of some help: https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/how-to-implement-binding-validation. A working example can be found in Demo 🥸 *WPF104 > BindingValidationDemo* and the belonging *RegExValidationRule* as an example. Make sure only positive numeric values can be entered. Create your own *validation* `ControlTemplate` if you have the time 😊